

Xcell journal

THE AUTHORITATIVE JOURNAL FOR PROGRAMMABLE LOGIC USERS

ESL Expands the Reach of FPGAs

INSIDE

ESL Tools Make
FPGAs Nearly
Invisible to Designers

Evaluating Hardware
Acceleration Strategies
Using C-to-Hardware Tools

The Benefits of
FPGA Coprocessing

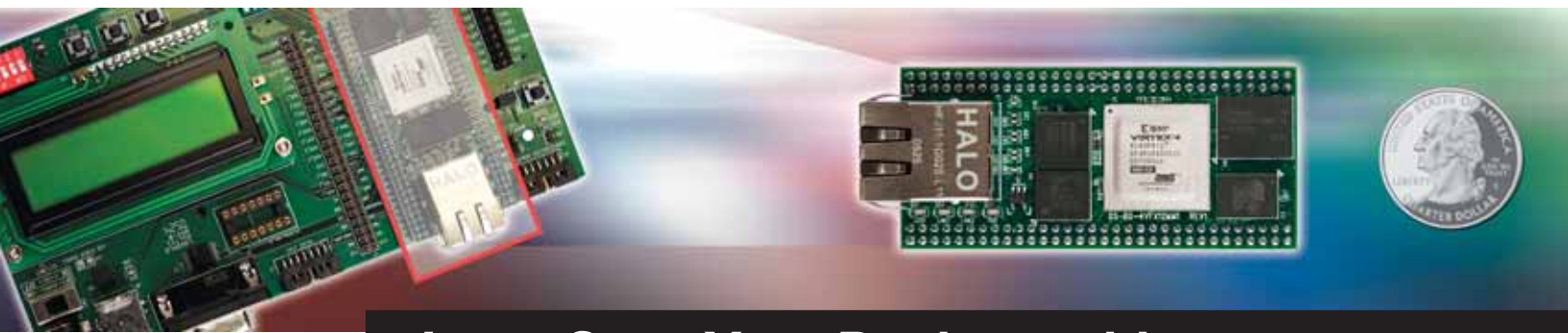
Accelerating System
Performance Using ESL
Design Tools and FPGAs

Tune Multicore Hardware
for Software



www.xilinx.com/xcell/

Support Across The Board.™



Jump Start Your Designs with Mini-Module Development Platforms

Mini-Module Features

- Small footprint (30 mm x 65.5 mm)
- Complete system-on-module solution with SRAM, Flash, Ethernet port and configuration memory
- Supports MicroBlaze™ or PowerPC® processors
- Based on the Virtex™-4 FX12 and 400K Gate Spartan™-3 FPGAs
- 76 User I/O

Baseboard Features

- Supports mini-module family
- Module socket allows easy connection
- Provides 3.3 V, 2.5 V and 1.2 V
- USB and RS232 ports
- 2 x 16 character LCD

Upcoming SpeedWay Design Workshops

- Developing with MicroBlaze
- Developing with PowerPC
- Xilinx Embedded Software Development
- Embedded Networking with Xilinx FPGAs

The Xilinx® Spartan™-3 and Virtex™-4 FX 12 Mini-Modules are off-the-shelf, fully configurable, complete system-on-module (SOM) embedded solutions capable of supporting MicroBlaze™ or PowerPC® embedded processors from Xilinx.

Offered as stand-alone modules and as development kits with companion baseboards, power supplies, software and reference designs, these modules can be easily integrated into custom applications and can facilitate design migration.

Gain hands-on experience with these development kits and other Xilinx tools by participating in an Avnet SpeedWay Design Workshop™ this spring.

Learn more about these mini-modules and upcoming SpeedWay Design Workshops at:

www.em.avnet.com/spartan3mini

www.em.avnet.com/virtex4mini

www.em.avnet.com/speedway



Enabling success from the center of technology™

1 800 332 8638
em.avnet.com

Xcell journal

PUBLISHER	Forrest Couch forrest.couch@xilinx.com 408-879-5270
EDITOR	Charmaine Cooper Hussain
ART DIRECTOR	Scott Blair
ADVERTISING SALES	Dan Teie 1-800-493-5551
TECHNICAL COORDINATOR	Milan Saini
INTERNATIONAL	Dickson Seow, Asia Pacific dickson.seow@xilinx.com Andrea Barnard, Europe/ Middle East/Africa andrea.barnard@xilinx.com Yumi Homura, Japan yumi.homura@xilinx.com
SUBSCRIPTIONS	All Inquiries www.xcellpublications.com
REPRINT ORDERS	1-800-493-5551



www.xilinx.com/xcell/

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

© 2006 Xilinx, Inc. All rights reserved. XILINX, the Xilinx Logo, and other designated brands included herein are trademarks of Xilinx, Inc. PowerPC is a trademark of IBM, Inc. All other trademarks are the property of their respective owners.

The articles, information, and other materials included in this issue are provided solely for the convenience of our readers. Xilinx makes no warranties, express, implied, statutory, or otherwise, and accepts no liability with respect to any such articles, information, or other materials or their use, and any use thereof is solely at the risk of the user. Any person or entity using such information in any way releases and waives any claim it might have against Xilinx for any loss, damage, or expense caused thereby.

A Shortcut to Success

This issue of the *Xcell Journal* features a very cohesive collection of articles about electronic system level (ESL) design. ESL is an umbrella term for tools and methods that allow designers with software programming skills to easily implement their ideas in programmable hardware (like FPGAs) without having to learn traditional hardware design techniques. The proliferation of these tools will make it easier for designers to use programmable devices for algorithm acceleration, high-performance computing, high-speed packet processing, and rapid prototyping.

In an effort to organize those vendors developing ESL products, in March 2006 Xilinx launched the ESL Initiative, inviting companies such as Celoxica and Impulse Accelerated Technologies to optimize support for Xilinx® embedded, DSP, and logic platforms and to establish common standards for ESL tool interoperability, among other goals.

Xilinx also set its own standards for participation in the ESL Initiative. To qualify as a partner, a company's ESL methodologies must be at a higher level of abstraction than RTL. They must also demonstrate a working flow for FPGAs and position their tools as an FPGA solution rather than an ASIC solution that also happens to work with FPGAs. Participants were additionally invited to write an article for this issue.

We also invited *FPGA Journal* Editor Kevin Morris to share his thoughts about ESL, and as you'll see he offers quite a interesting perspective. Depending on who you talk to, ESL is either all hype or the real deal. At Xilinx, we've committed to this promising technology – working with a variety of vendors (12 and growing) whose solutions are designed for a variety of applications.

I invite you to learn more about ESL and particularly each company's respective products by delving further into this issue of the *Xcell Journal*, or by visiting www.xilinx.com/esl.

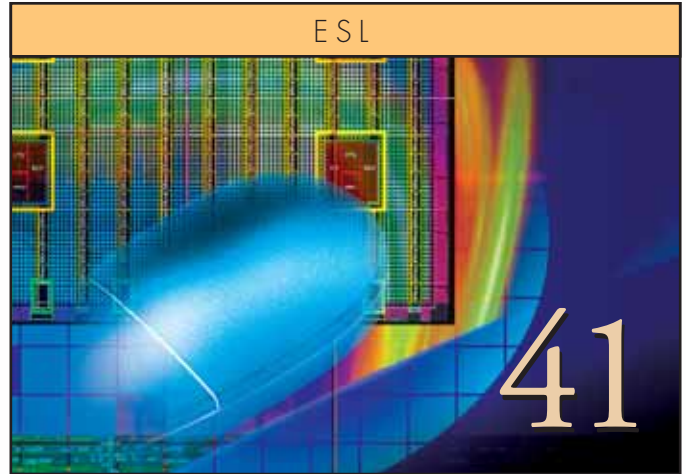


Forrest Couch

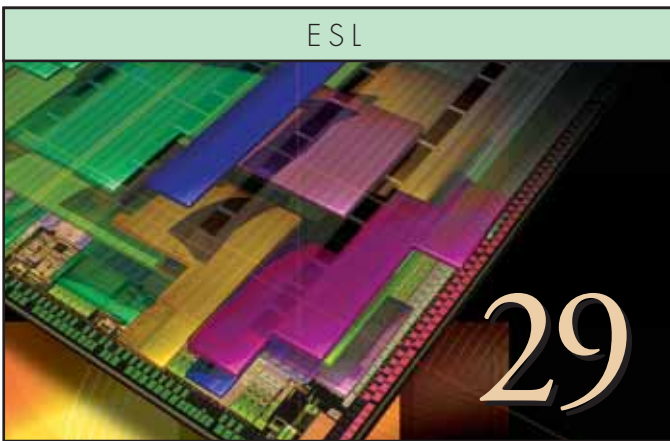
Forrest Couch
Publisher



Evaluating Hardware Acceleration Strategies Using C-to-Hardware Tools
Software-based methods enable iterative design and optimization for performance-critical applications.



Accelerating System Performance Using ESL Design Tools and FPGAs
ESL design tools provide the keys for opening more applications to algorithm acceleration.



The Benefits of FPGA Coprocessing
The Xilinx ESL Initiative brings the horsepower of an FPGA coprocessor closer to the reach of traditional DSP system designers.

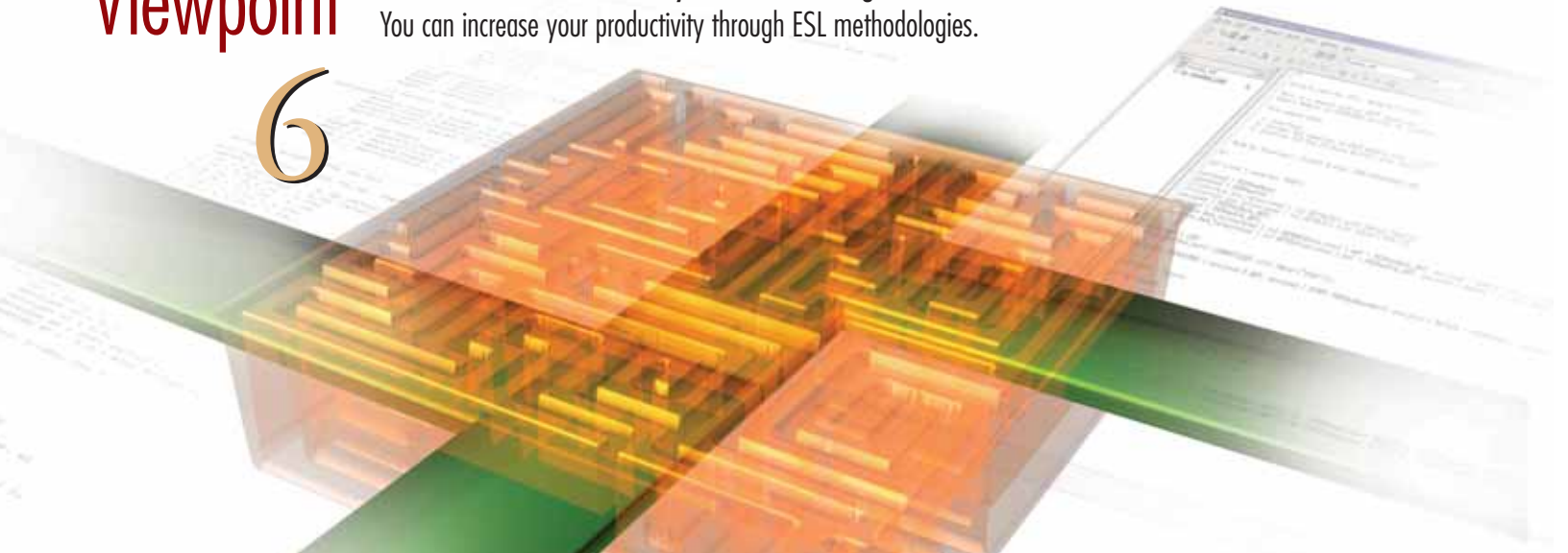


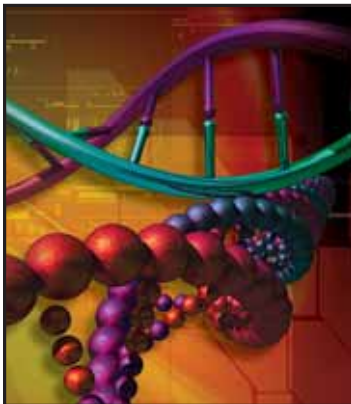
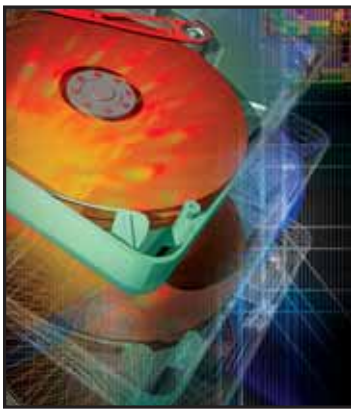
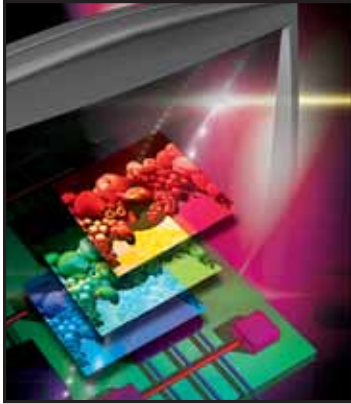
Tune Multicore Hardware for Software
Teja FP and Xilinx FPGAs give you more control over hardware configuration.

Viewpoint

ESL Tools Make FPGAs Nearly Invisible to Designers
You can increase your productivity through ESL methodologies.

6





Xcell journal

VIEWPOINTS

ESL Tools Make FPGAs Nearly Invisible to Designers6
Join the “Cool” Club9

ESL

Bringing Imaging to the System Level with PixelStreams12
Evaluating Hardware Acceleration Strategies Using C-to-Hardware Tools16
Accelerating Architecture Exploration for FPGA Selection and System Design19
Rapid Development of Control Systems with Floating Point22
Hardware/Software Partitioning from Application Binaries26
The Benefits of FPGA Coprocessing29
Using SystemC and SystemCrafter to Implement Data Encryption32
Scalable Cluster-Based FPGA HPC System Solutions35
Accelerate Image Applications with Poseidon ESL Tools38
Accelerating System Performance Using ESL Design Tools and FPGAs41
Transforming Software to Silicon46
A Novel Processor Architecture for FPGA Supercomputing49
Turbocharging Your CPU with an FPGA-Programmable Coprocessor52
Tune Multicore Hardware for Software55
Automatically Identifying and Creating Accelerators Directly from C Code58

GENERAL

ExploreAhead Extends the PlanAhead Performance Advantage62
Designing Flexible, High-Performance Embedded Systems66
Tracing Your Way to Better Embedded Software70
To Interleave, or Not to Interleave: That is the Question74
Using FPGA-Based Hybrid Computers for Bioinformatics Applications80
II DRR Delivers High-Performance SDR84
Eliminate Packet Buffering Busywork88

REFERENCE92

ESL Tools Make FPGAs Nearly Invisible to Designers

You can increase your productivity through ESL methodologies.



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

FPGA business dynamics have been rather consistent over the past decade. Price per gate has continued to fall annually by an average of 25%, while device densities have continued to climb by an average of 56%. Concurrent with advances in silicon, design methodologies have also continued to evolve.

In particular, a new paradigm known as electronic system level (ESL) design is promising to usher in a new era in FPGA design. Although the term ESL is broad and its definition subject to different interpretations, here at Xilinx it refers to tools and methodologies that raise design abstraction to levels above the current mainstream register transfer level (RTL) language (Figure 1).

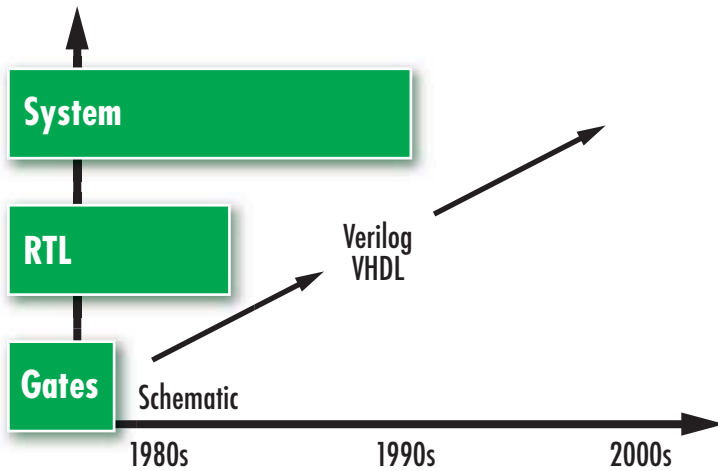


Figure 1 – Design methodologies are evolving to HLL-based system-level design.

Put another way, it is now possible for you to capture designs in high-level languages (HLLs) such as ANSI C and implement them in an optimized manner on FPGA hardware. These advances in software methodologies – when combined with the increasing affordability of FPGA silicon – are helping to make programmable logic the hardware platform of choice for a wide and rapidly expanding set of target applications.

Defining the Problem

You might wonder what problem ESL is trying to solve. Why is it needed? To answer this, let’s consider the following three scenarios. First, many of today’s design problems originate as a software algorithm in C. The traditional flow to hardware entails a manual conversion of the C source to equivalent HDL. ESL adds value by providing an automatic conversion from HLL to RTL or gates. For those wishing to extract the most performance, you can optionally hand-edit the intermediate RTL.

Second, for both power and performance reasons, it is clear that traditional processor architectures are no longer sufficient to handle the computational complexity of the current and future generation of end applications. ESL is a logical solution that helps overcome the challenges of processor bottlenecks by making an easy

and automated path to hardware-based acceleration possible.

Third, as the types of appliances that can now be deployed on FPGAs become more sophisticated, traditional simulation methods are often not fast enough. ESL methodologies enable faster system simulations, utilizing very high speed transaction-based models that allow you to verify functionality and perform hardware/software trade-off analysis much earlier in the design cycle.

Value Proposition for FPGAs

ESL technologies are a unique fit for programmable hardware. Together, ESL tools and FPGAs enable a desktop-based development environment that allows applications to target hardware using standard software-development flows. You can design, debug, and download applications developed using HLLs to an FPGA board much the same way that you can develop, debug, and download code to a CPU board.

Additionally, with powerful 32-bit embedded processors now a common feature in FPGAs, you can implement a complete system – hardware and software – on a single piece of silicon. Not only does this provide a high level of component integration, but with the help of ESL tools, it is now easy and convenient to dynamically partition design algorithms between the appropriate hardware (FPGA fabric) and software (embedded CPU) resources on the chip (Figure 2).

ESL brings a productivity advantage to both current FPGA hardware designers as well as to a potentially large number of software programmers. Hardware engineers are using HLL-based methods for rapid design prototyping and to better manage the complexity of their designs. Software developers, interested in accelerating their CPU bottlenecks, are using ESL

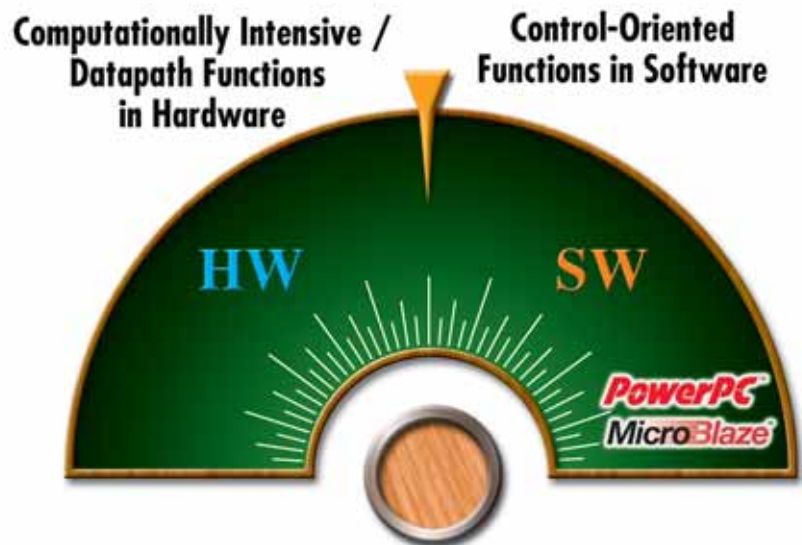


Figure 2 – ESL facilitates dynamic hardware/software partitioning conveniently possible in FPGAs.

ESL adds value by being able to abstract away the low-level implementation details associated with hardware design. By doing so, the tools simplify hardware design to such an extent so as to render the hardware practically “invisible” to designers.

flows to export computationally critical functions to programmable hardware.

ESL adds value by being able to abstract away the low-level implementation details associated with hardware design. By doing so, the tools simplify hardware design to such an extent so as to render the hardware practically “invisible” to designers. This helps expand the reach of FPGAs from the traditional base


If you remain skeptical about the role of HLLs in hardware design, you are not alone. Your concerns might range from compiler quality of results and tool ease-of-use issues to the lack of language standardization and appropriate training on these emerging technologies. The good news is that ESL suppliers are aware of these challenges and are making substantial and sustained investments to advance

methods are being used with FPGAs to solve practical problems.

You may also be wondering if ESL will signal the end of current RTL-based design methodologies. Although there is little doubt that ESL should be regarded as a disruptive technology, ESL flows today are positioned to complement, rather than compete, with HDL methods. In fact, several ESL tools write out RTL descriptions that are synthesized to gate level using current RTL synthesis. And there will always be scenarios where design blocks may need to be hand-coded in RTL for maximum performance. By plugging into the existing well-developed infrastructure, ESL can help solve those design challenges not addressed by current methodologies.

Next Steps

If you are looking for a place to begin your orientation on ESL and FPGAs, a good place to start would be the ESL knowledge center at www.xilinx.com/esl. In March 2006, Xilinx launched the ESL Initiative – a collaborative alliance with the industry’s leading tool providers to promote the value, relevance, and benefits of ESL tools for FPGAs (Table 1). The website aims to empower you with knowledge and information about what is available and how to get started, including information on low-cost evaluation platforms that bundle FPGA boards with ESL software and design examples.

This issue of the *Xcell Journal* features a range of articles on ESL and FPGAs written by engineers at Xilinx as well as our ESL partners. These articles are designed to give you better insights into emerging ESL concepts. You may find that some of these technologies do indeed present a better approach to solving your design challenges. We hope that some of you will feel inspired to take the next step and become part of the growing number of early adopters of this unique and revolutionary new methodology. 

Partner	From HLL To FPGA	Xilinx CPU Support	Description (Key Value)
Bluespec	✓		SystemVerilog-Like to FPGA (High QoR)
Binachip	✓	✓	Processor Acceleration (Accelerates Binary Software Code in FPGA)
Celoxica	✓	✓	Handel-C, SystemC to FPGA (High QoR)
Cebatech	✓		ANSI C to FPGA (Time to Market)
Critical Blue	✓		Co-Processor Synthesis (Accelerates Binary Software Code in FPGA)
Codetronix	✓	✓	Multi-Threaded HLL for Hardware/Software Design (Time to Market)
Impulse	✓	✓	Impulse-C to FPGA (Low Cost)
Mimosys	✓	✓	PowerPC APU-Based Acceleration (Ease of Use)
Mitron	✓		Mitron-C to FPGA (Supercomputing Applications)
Mirabilis		✓	High-Speed Virtual Simulation (Perform Hardware/Software Trade-Off Analysis)
Nallatech	✓		Dime-C to FPGA (High-Performance Computing Applications)
Poseidon	✓	✓	MicroBlaze/PowerPC Acceleration (Analysis, Profiling, and C Synthesis)
SystemCrafter	✓		SystemC to FPGA (Rapid Prototyping)
Teja	✓	✓	Distributed Processing Using MicroBlaze/PowerPC (High-Speed Packet Processing)

Table 1 – Xilinx partners provide a wide spectrum of system-level design solutions.

of hardware designers to a new and potentially larger group of software designers, who until now were exclusively targeting their applications to processors. As there are many more software developers in the world than hardware designers, this equates to a large new opportunity for FPGAs.

the state of the art. These efforts are beginning to yield results, as evidenced by an increasing level of interest from end users. For instance, more than two-thirds of the engineers in a Xilinx® survey expressed a keen interest in learning about the capabilities of ESL. Xilinx estimates that hundreds of active design seats exist where ESL

Join the "Cool" Club

You're not doing ESL yet? Loser!



by Kevin Morris
Editor – *FPGA Journal*
Techfocus Media, Inc.
kevin@techfocusmedia.com

When the *Xcell Journal* asked me to write a viewpoint on ESL, I thought, “Hey, why not? They’ve got ‘Journal’ in their name and I know all about ESL.” ESL is the design methodology of the future. ESL will revolutionize the electronics industry. ESL will generate billions of dollars worth of engineering productivity while the people and companies that develop the technology fight tooth and nail to capture infinitesimal percentages of that sum.

ESL (electronic system level [design]) brings with it a pretentious name and the potential for manic marketing misrepresentation. Although there are excellent ESL products and efforts underway, the early market is a good time for the buyer to beware and for savvy design teams to understand that they are likely to be experimenting with cutting-edge, nascent tech-

nology. ESL is also currently an over-broad term, encapsulating a number of interesting but diverse tools and technologies.

In order to understand ESL, we need to hack through the typical layers of marketing hype and confusion to get down to the fundamentals. ESL is not about languages or layers of abstraction. ESL is about productivity. Electronic hardware design (this is important because ESL is a hardware designer’s term) is far too inefficient. Forget the fact that we have boosted productivity an order of magnitude or two over the past 20 years because of technology advances in electronic design automation. Moore’s Law is a much harsher mistress than that. Do the math. In two decades, the gate count of the average digital design platform has risen by a factor of something like 2K. Even if EDA has managed a 200x improvement in gates per digital designer day, they are falling behind at a rate of more than 5x per decade.

This effect was observed ad-infinitum in practically every EDA PowerPoint presentation given in the 1990s. Logarithmic graphs of gate counts were shown skyrocketing

upward exponentially while designer productivity lounged along linearly, leaving what the EDA marketers proudly promoted as “The Gap.” This gap (skillfully highlighted and animated by black-belt marketing ninjas) was their battle cry – their call to arms. Without the help of their reasonably priced power tools, you and your poor, helpless electronic system design company would be gobbled up by The Gap, digested by your competitors who were, of course, immune to Moore’s Law because of the protective superpower shield of their benevolent EDA suppliers.

This decade, however, The Gap has gone into public retirement. Not because it isn’t still there – it is, growing as fast as ever. This pause is because we found a killer-app use for all of those un-designable gap gates, giving us a temporary, one-time reprieve from the doom of design tool deficiency. Our benefactor? Programmability. It has been widely documented that devices like FPGAs pay a steep price in transistor count for the privilege of programmability. Estimates run as high as 10x for the average area penalty imposed by programmable logic when compared with custom ASIC technologies. Dedicate 90% of



your logic fabric transistors to programmability, throw in a heaping helping of RAM for good measure, and you've chewed up enough superfluous silicon to leave a manageable design problem behind.

By filling several process nodes worth of new transistors with stuff you don't have to engineer, you've stayed on the productivity pace without having to reinvent yourself. Now, unfortunately, the bill is coming due again. FPGAs have become as complex as ASICs were not that long ago, and the delta is diminishing. Design teams around the world are discovering that designing millions of gates worth of actual FPGA logic using conventional RTL methodologies can be a huge task.

In an ASIC, however, this deficiency in design efficiency was masked by the mask. The cost and effort required to generate and verify ASIC mask sets far exceeded the engineering expense of designing an ASIC with RTL methodologies. Boosting the productivity of the digital designer would be a bonus for an ASIC company, but not a game-changing process improvement. Even if digital design time and cost went to zero, cutting-edge ASIC development would still be slow and expensive.

In FPGA design, there is no such problem to hide behind, however. Digital design is paramount on the critical path of most FPGA projects. Are you an FPGA designer? Try this test. Look outside your office door up and down the corridor. Do you see a bunch of offices or cubes filled with engineers using expensive design-for-manufacturing (DFM) software, correcting for optical proximity, talking about "rule decks" and sweating bullets about the prospect of their design going to "tapeout"? Not there, are they? Now look in a mirror. There is your company's critical path.

With ESL the buzzword-du-jour in design automation, everyone in the EDA industry is maneuvering – trying to find a way to claim that what they were already doing is somehow ESL so they can be in the "cool" club. As a result, everything from bubble and block diagram editors to transaction-level simulation to almost-high-level hardware synthesis has been branded "ESL" and rushed to the show floor at the Design

Automation Conference, complete with blinking lights, theme music, and special, hush-hush secret sneak-preview demos served with shots of espresso in exclusive private suites.

FPGAs and ESL are a natural marriage. If we believe our premise that ESL technology delivers productivity – translating into a substantial reduction in time to market – we see a 100% alignment with FPGA's key value proposition. People need ESL techniques to design FPGAs for the same reason they turned to FPGAs in the first place –

FPGA platforms and ESL tools deliver dramatically increased flexibility.

they want their design done now. Second-order values are aligned as well, as both FPGA platforms and ESL tools deliver dramatically increased flexibility. When design changes come along late in your product development cycle, both technologies stand ready to help you respond rapidly.


With all of this marketing and uncertainty, how do you know if what you're doing is actually ESL? You certainly do not want all of the other designers pointing and laughing at engineering recess if you show up on the playground with some lame tool that isn't the real thing. Although there is no easy answer to that question, we can provide some general guidelines. First, if you are describing your design in any language in terms of the structure of the hardware, what you are doing probably isn't ESL. Most folks agree that the path to productivity involves raising the abstraction layer.

If we are upping our level of abstraction, how will the well-heeled hardware engineer be describing his design circa 2010? Not in RTL. Above that level of abstraction, however, a disquieting discontinuity occurs. Design becomes domain-specific. The natural abstraction for describing a digital signal

processing or video compression algorithm probably bears no resemblance to the native design tongue of the engineer developing packet-switching algorithms, for example. As a result, we are likely to see two different schools of thought in design description. One group will continue to pursue general-purpose specification techniques, while the other charts a course of domain-specificity.

In the software world, there is ample precedent for both ways of thinking. Even though software applications encompass a huge gamut from business to entertainment to scientific, most software engineers manage to develop their applications using one of the popular general-purpose languages like C/C++ or Java. On the other hand, some specialized application types such as database manipulation and user interface design have popularized the use of languages specific to those tasks. Software is a good exemplar for us, because programmers have been working at the algorithmic level of abstraction for years. Hardware design is only just now heading to that space.

The early ESL tools on the market now address specific hot spots in design. Some provide large-scale simulation of complex, multi-module systems. Others offer domain-specific synthesis to hardware from languages like C/C++ or MATLAB's M, aimed at everything from accelerating software algorithms and offloading embedded processors to creating super-high-performance digital signal processing engines for algorithms that demand more than what conventional Von Neumann processors can currently deliver.

These tools are available from a variety of suppliers, ranging from FPGA vendors themselves to major EDA companies to high-energy startups banking on hitting it big based on the upcoming discontinuity in design methodology. You should be trying them out now. If not, plan on being left behind by more productive competitors over the next five years. At the same time, you should realize that today's ESL technologies are still in the formative stage. These are nascent engineering tools with a long way to go before they fulfill the ultimate promise of ESL – a productivity leap in digital design methodology that will ultimately allow us to keep pace with Moore's Law. 

WHAT'S NEW



To complement our flagship publication *Xcell Journal*, we've recently launched three new technology magazines:

- *Embedded Magazine*, focusing on the use of embedded processors in Xilinx® programmable logic devices.
- *DSP Magazine*, focusing on the high-performance capabilities of our FPGA-based reconfigurable DSPs.
- *I/O Magazine*, focusing on the wide range of serial and parallel connectivity options available in Xilinx devices.

In addition to these new magazines, we've created a family of Solution Guides, designed to provide useful information on a wide range of hot topics such as *Broadcast Engineering*, *Power Management*, and *Signal Integrity*. Others are planned throughout the year.

Bringing Imaging to the System Level with PixelStreams

You can develop FPGA-based HDTV and machine vision systems in minutes using Celoxica's suite of ESL tools and the PixelStreams video library.

by Matt Aubury
Vice President, Engineering
Celoxica
matt.aubury@celoxica.com

The demand for high-performance imaging systems continues to grow. In broadcast and display applications, the worldwide introduction of HDTV has driven data rates higher and created a need to enhance legacy standard definition (SD) content for new displays. Simultaneously, rapid changes in display technology include novel 3D displays and new types of emissive flat panels.

Concurrently, machine vision techniques that were once confined to manufacturing and biomedical applications are finding uses in new fields. Automotive applications include lane departure warning, drowsy driver detection, and infrared sensor fusion. Security systems currently in development can automatically track people in closed-circuit TV images and analyze their movements for certain patterns of behavior. In homes, video-enabled robots will serve as entertainment and labor-saving devices.

Cost, processing power, and time to market are critical issues for these new applications. With flexible data paths, high-performance arithmetic, and the ability to reconfigure on the fly, FPGAs are increasingly seen as the preferred solution. However, designers of reconfigurable imaging systems face some big hurdles:

- Getting access to a rich library of reusable IP blocks
- Integrating standard IP blocks with customized IP
- Integrating the imaging part of the system with analysis, control, and networking algorithms running on a CPU such as a PowerPC™ or Xilinx® MicroBlaze™ processor

In this article, I'll introduce PixelStreams from Celoxica, an open framework for video processing that has applicability in both broadcast and machine vision applications. PixelStreams leverages the expressive power of the Celoxica DK Design Suite to develop efficient and high-performance FPGA-based solutions in a fraction of the time of traditional methods.

Filters and Streams

A PixelStreams application is built from a network of filters connected together by streams. Filters can generate streams (for example, a sync generator), transform streams (for example, a colorspace converter), or absorb streams (for example, video output).

Streams comprise flow control, data transport, and high-level type information. The data component in turn contains:

- An active flag (indicating that this pixel is in the current region-of-interest)
- Optional pixel data (in 1-bit, 8-bit, or signed 16-bit monochrome, 8-bit YCbCr, or RGB color)
- An optional (x, y) coordinate
- Optional video sync pulses (horizontal and vertical sync, blanking, and field information for interlaced formats)

Combining all of these components into a single entity gives you great flexibility.

Filters can perform purely geometric transforms by modifying only the coordinates, or create image overlays just by modifying the pixel data.

Filters use the additional type information that a stream provides in two ways: to ensure that they are compatible with the stream they are given (for example, a sync generator cannot be attached directly to a video output because it does not contain any pixel data), and to automatically parameterize themselves. Filters are polymorphic; a single PxsConvert() filter handles colorspace conversion between each of the pixel data formats (which includes 20 different operations).

Flow Control

Flow control is handled by a downstream "valid" flag (indicating the validity of the data component on a given clock cycle) and an upstream "halt" signal. This combination makes it easy to assemble multi-rate designs, with pixel rates varying (dynamically if necessary) from zero up to the clock rate. Simple filters are still easy to design. For example, a filter that modifies only the pixel components would use a utility function to copy the remaining components (valid, active, coordinates, and sync pulses) from its inputs to outputs while passing the halt signal upstream. More complex filters that need to block typically require buffering at their inputs, which is easily handled by using the generic PxsFIFO() filter.

Sophisticated designs will require filter networks in multiple clock domains. These are easy to implement using the DK Design Suite, which builds efficient and reliable channels between clock domains and sends and receives filters that convert streams to channel communications. The first domain simply declares the channel (in this case, by building a short FIFO to maximize throughput) and uses the PxsSend() filter to transmit the stream:

```
// domain 1
chan X with { fifolength = 16 };
...
PxsSend (&S0, &X);
```

The second domain imports the channel using extern and receives the stream from it with a PxsReceive() filter:

```
// domain 2
extern chan X;
...
PxsReceive (&S1, &X);
```

Custom Filters

PixelStreams comes "batteries included," with more than 140 filters, many of which can perform multiple functions. A sample of this range is shown in Table 1. Every PixelStreams filter is provided with complete source code. Creating custom filters is often just a matter of modifying one of the standard filters. New filters are accessible from either the graphical editor or from code.

- Image analysis and connected component ("blob") labeling
- Image arithmetic and blending
- Clipping and region-of-interest
- Colorspace conversion and dithering
- Coordinate transforms, scaling, and warping
- Flow control, synchronization, multiplexing, FIFOs, and cross-clock domain transport
- Convolutions, edge detection, and non-linear filtering
- Frame buffering and de-interlacing
- Color look-up-tables (LUTs)
- Grayscale morphology
- Noise generators
- Video overlays, text and cursor generators, line plotter
- Sync generators for VGA, TV, HDTV
- Video I/O

Table 1 – An overview of the provided source-level IP

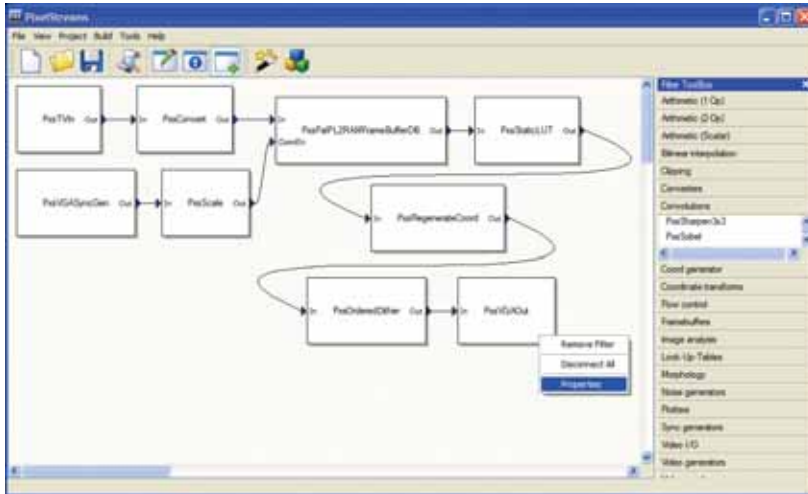


Figure 1 – Example application built using the PixelStreams graphical editor

```
#define ClockRate 65000000
#include "pxs.hch"

void main (void)
{
    /* Streams */
    PXS_I_S (Stream0, PXS_RGB_U8);
    PXS_I_S (Stream1, PXS_YCbCr_U8);
    PXS_PV_S (Stream2, PXS_EMPTY);
    PXS_PV_A (Stream3, PXS_EMPTY);
    PXS_PV_A (Stream4, PXS_RGB_U8);
    PXS_PV_S (Stream6, PXS_RGB_U8);
    PXS_PV_A (Stream7, PXS_RGB_U8);
    PXS_PV_S (Stream8, PXS_RGB_U8);

    /* Filters */
    par
    {
        PxsTVIn (&Stream1, 0, 0, ClockRate);
        PxsConvert (&Stream1, &Stream0);
        PxsPalPL2RAMFrameBufferDB (&Stream0, &Stream3, &Stream4, Width, PXS_BOB,
                                   PalPL2RAMCT(0), PalPL2RAMCT(1), ClockRate);
        PxsVGASyncGen (&Stream2, Mode);
        PxsScale (&Stream2, &Stream3, 704, 576, 1024, 768);
        PxsStaticLUT (&Stream4, &Stream7, PxsLUT8Square);
        PxsOrderedDither (&Stream8, &Stream6, 5);
        PxsVGAOut (&Stream6, 0, 0, ClockRate);
        PxsRegenerateCoord (&Stream7, &Stream8);
    }
}
```

Figure 2 – Generated source code for example application

Concept to FPGA Hardware in 10 Minutes

Let's take a simple HDTV application as an example. We have an SD video stream that we want to de-interlace, upscale, apply gamma correction to, then dither and output to an LCD.

We can assemble the blocks for this very quickly using the PixelStreams graphical editor. The TV (SD) input undergoes color-space conversion from YCbCr to RGB and is stored in a frame buffer that implements simple bob de-interlacing. Pixels are fetched from the frame buffer, with the upscaling achieved by modifying the lookup coordinates. The output stream is gamma-corrected using a color LUT, dithered, and output.

Assembling this network takes only a few minutes. The editor uses heuristics to set the stream parameters automatically. The final design is shown in Figure 1.

One click generates code for this network, with a corresponding project and workspace, and launches the DK Design Suite. The generated code is shown in Figure 2. For this simple application each filter is a single process running in a parallel "par" block. More sophisticated designs will have C-based sequential code running in parallel to the filters.

One more click starts the build process for our target platform (in this case, Celoxica's Virtex™-4 based RC340 imaging board). The build process automatically runs ISE™ software place and route and generates a bit file, taking about five minutes. One last click and the design is downloaded to the board over USB, as shown in Figure 3.

As designs become more complex, it becomes easier to code them directly rather than use the graphical editor. This enables a higher level of design abstraction, with the ability to construct reusable hierarchies of filters and add application control logic.

Simulation

Simulation is handled by the DK Design Suite's built-in cycle-based simulator and utilizes its virtual platform technology to show video input and output directly on screen. Celoxica's co-simulation manager can simulate designs incorporating CPUs and external RTL components.

Future versions of PixelStreams will extend both its flexibility as a framework and the richness of the base library of IP.

Synthesis

PixelStreams capitalizes on the strengths of the DK Design Suite's built-in synthesis to achieve high-quality results on Xilinx FPGAs. For example:

- When building long pipelines, it is common that certain components of streams will not be modified. These will be efficiently packed into SRL16 components.
- When doing convolutions or coordinate transforms, MULT18 or DSP48 components will be automatically targeted (or if the coefficients are constant, efficient constant multipliers are built).
- Line buffers and FIFOs will use either distributed or pipelined block RAM resources, which are automatically tiled.
- The data components of a stream are always registered, to maximize the timing isolation between adjacent filters.
- DK's register retiming can move logic between the pipeline stages of a filter to maximize clock rate (or minimize area).

In addition, DK can generate VHDL or Verilog for use with third-party synthesis tools such as XST or Synplicity and simulation tools such as Aldec's Active HDL or Mentor Graphics's ModelSim.

System Integration

A complete imaging system typically comprises:

- A PixelStreams filter network, with additional C-based control logic
- A MicroBlaze or PowerPC processor managing control, user interface, or high-level image interpretation functions
- VHDL or Verilog modules (possibly as the top level)

To deal with CPU integration, PixelStreams provides a simple "PxsBus" that can be readily bridged to CPU buses such as AMBA or OPB or controlled remotely (over PCI or USB). This is purely a control bus, allowing filters to be controlled by a CPU (for adding menus or changing filter coefficients) or to provide real-time data back from

a filter to the controlling application (such as the result of blob analysis).

To support integration with RTL flows, PixelStreams offers a PxsExport() filter that packages a filter network into a module that can be instantiated from VHDL or Verilog. Alternatively, an RTL module can be instantiated within a PixelStreams top level using PxsImport(). Used together, pre-synthesized filter networks can be rapidly instantiated and reduce synthesis time.

Conclusion

Combining the two main elements of ESL tools for Xilinx FPGAs – C- and model-based design – PixelStreams offers a uniquely powerful framework for implementing a variety of imaging applications. The provision of a wide range of standard filters combined with features for integration into RTL flows and hardware/software co-design makes it easy to add imaging features to your system-level designs.

Future versions of PixelStreams will extend both its flexibility as a framework and the richness of the base library of IP. We plan to add additional pixel formats (such as Porter-Duff based RGBA) and color depths and increase the available performance by introducing streams that transfer multiple pixels per clock cycle. We also intend to introduce the ability to easily transmit streams over other transports, such as USB, Ethernet, and high-speed serial links, add more filters for machine vision features such as corner detection and tracking, and offer additional features for dynamically generating user interfaces.

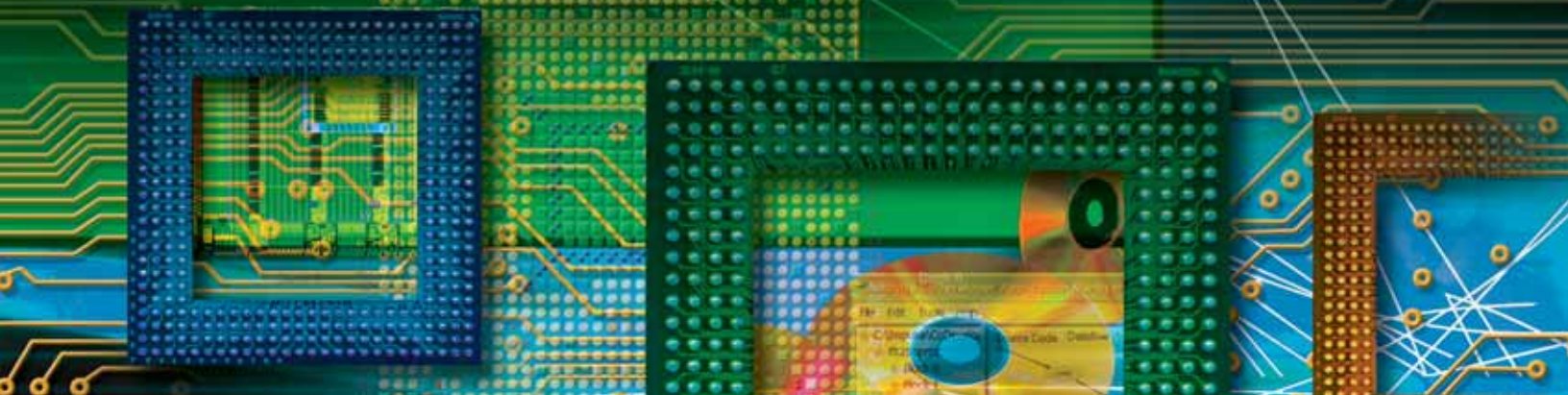
You can use PixelStreams to target any prototyping or production board identified for a project with the addition of simple I/O filters. It can immediately target Celoxica's range of RC series platforms and is fully compatible with the ESL Starter Kit for Xilinx FPGAs. For more information, visit www.celoxica.com/xilinx, www.celoxica.com/pxs, and www.celoxica.com/rc340.



Figure 3 – Application running in hardware

Evaluating Hardware Acceleration Strategies Using C-to-Hardware Tools

Software-based methods enable iterative design and optimization for performance-critical applications.



by David Pellerin
CTO

Impulse Accelerated Technologies, Inc.
david.pellerin@impulsec.com

Scott Thibault, Ph.D.
President

Green Mountain Computing Systems, Inc.
thibault@gmvhdl.com

Language-based ESL tools for FPGAs have proven themselves viable alternatives to traditional hardware design methods, bringing FPGAs within the reach of software application developers. By using software-to-hardware compilation, software developers now have greater access to FPGAs as computational resources.

What has often been overlooked by traditional hardware designers, however, is the increased potential for design exploration, iterative performance optimization, and higher performance when ESL tools are used in combination with traditional FPGA design methods.

In this article, we'll describe the role of C-to-hardware ESL tools for iterative design exploration and interactive optimization. We'll present techniques for evaluating alternative implementations of C-language accel-

erators in Xilinx® FPGAs and explore the relative performance of fixed- and floating-point FPGA algorithms.

Crossing the Abstraction Gap

Before ESL tools for FPGAs existed, it was necessary to describe all aspects of an FPGA-based application using relatively low-level methods such as VHDL, Verilog, or even schematics. These design methods are still adequate, and in many cases preferable, for traditional FPGA-based hardware applications. However, when using traditional hardware design methods for creating complex control or computationally intense applications, a significant gap in abstraction (Figure 1) can exist between the original software algorithm and its corresponding synthesizable hardware implementation, as expressed in VHDL or Verilog. Crossing this gap may require days or weeks of tedious design conversion, making iterative design methods difficult or impossible to manage.

Tools providing C compilation and optimization for FPGAs can help software and hardware developers cross this gap by providing behavioral algorithm-level methods of design. Even for the most experienced FPGA designers, the potential exists for design improvements using such tools.

Although it may seem counterintuitive to an experienced hardware engineer, using higher level tools can actually result in higher performance applications because of the dramatically increased potential for design experimentation and rapid prototyping.

Iterative Optimization Is the Key

To understand how higher level tools can actually result in higher performance applications, let's review the role of software compilers for more traditional non-FPGA processors.

Modern software compilers (for C, C++, Java, and other languages) perform much more than simple language-to-instruction conversions. Modern processors, computing platforms, and operating systems have a diverse set of architectural characteristics, but today's compilers are built in such a way that you can (to a great extent) ignore these many architectural features. Advanced optimizing compilers take advantage of low-level processor and platform features, resulting in faster, more efficient applications.

Nonetheless, for the highest possible performance, you must still make programming decisions based on a general understanding of the target. Will threading an application help or hurt performance?

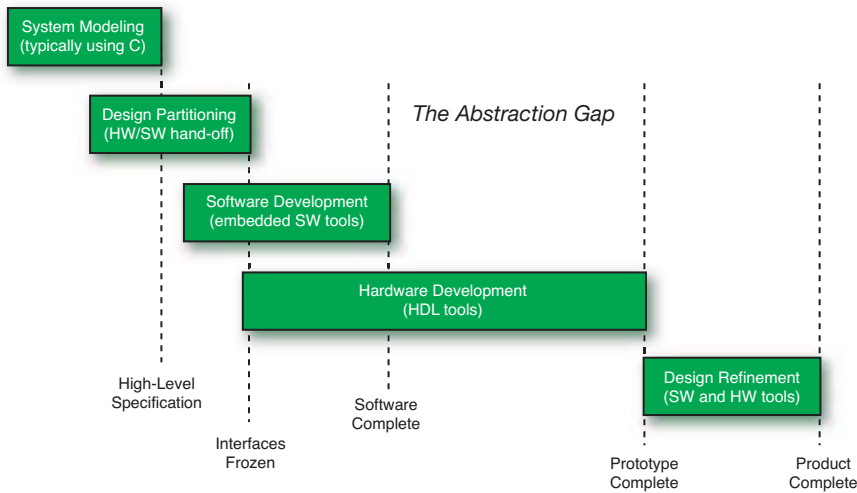


Figure 1 – Crossing the abstraction gap: in system-level design, hardware development can be the bottleneck when creating a working prototype.

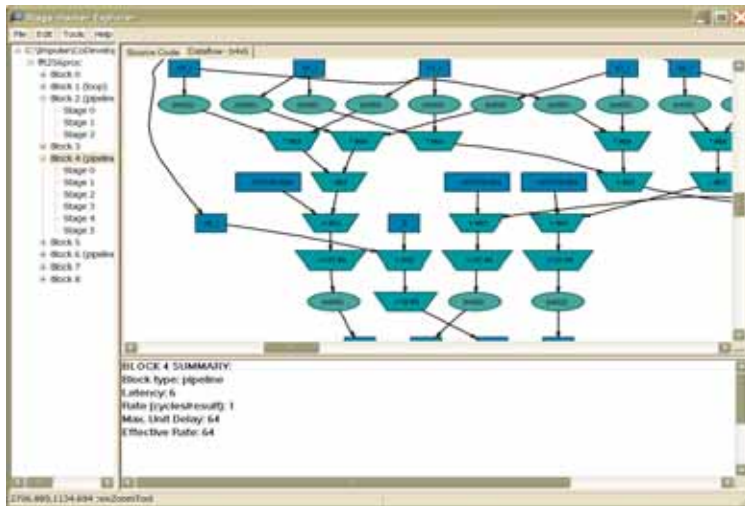


Figure 2 – A dataflow graph allows C programmers to analyze the generated hardware and perform explorative optimizations to balance trade-offs between size and speed. Illustrated in this graph is the final stage of a six-stage pipelined loop. This graph also helps C programmers understand how sequential C statements are parallelized and optimized.

Should various types of data be stored on disk, maintained in heap memory, or accessed from a local array? Is there a library function available to perform a certain I/O task, or should you write a custom driver? These questions, and others like them, are a standard and expected part of the application development process.

For embedded, real-time, and DSP application developers there are even more decisions to be made – and more dramatic performance penalties for ignoring the realities of the target hardware. For these platforms, the ability to quickly experiment with new algorithmic approaches is an

important enabler. It is the primary reason that C programming has become an important part of every embedded and DSP programmer's knowledge base. Although most embedded and DSP programmers understand that higher performance is theoretically possible using assembly language, few programmers wish to use such a low level of abstraction when designing complex applications.

What does all of this mean for FPGA programmers? For software engineers considering FPGAs, it means that some understanding of the features and constraints of FPGA platforms is critical to

success; it is not yet possible to simply ignore the nature of the target, nor is it practical to consider legacy application porting to be a simple recompilation of existing source code. For hardware engineers, it means that software-to-hardware tools should be viewed as complements to – not replacements for – traditional hardware development methods. For both software and hardware designers, the use of higher level tools presents more opportunities for increasing performance through experimentation and fast prototyping.

Practically speaking, the initial results of software-to-hardware compilation from C language descriptions are not likely to equal the performance of hand-coded VHDL, but the turnaround time to get those first results working may be an order of magnitude better. Performance improvements can then occur iteratively, through an analysis of how the application is being compiled to the hardware and through the experimentation that C-language programming allows.

Graphical tools like those shown in Figure 2 can help provide initial estimates of algorithm performance such as loop latencies and pipeline throughput. Using such tools, you can interactively change optimization options or iteratively modify and recompile C code to obtain higher performance. Such design iterations may take only a matter of minutes when using C, whereas the same iterations may require hours or even days when using VHDL or Verilog.

Analyzing Algorithms

To illustrate how design iteration and C-language programming can help when prototyping algorithms, consider a DSP function such as a fast Fourier transform (FFT) or an image-processing function such as a filter that must accept sample data on its inputs and generate the resulting filtered values on its outputs. By using C-to-hardware tools, we can easily try a variety of different implementation strategies, including the use of different pipeline depths, data widths, clock rates, and numeric formats, to find a combination of size (measured as FPGA slice count) and speed (measured both as clock speed and data throughput) that meets the specific requirements of a larger hardware/software system.

To demonstrate these ideas, we conducted a series of tests with a 32-bit implementation of the FFT algorithm. The FFT we chose includes a 32-bit FIFO input, a 32-bit FIFO output, and two clocks, allowing the FFT to be clocked at a different rate than the embedded processor with which it communicates. The algorithm itself is described using relatively straightforward, hardware-independent C code. This implementation of the FFT uses a main loop that performs the required butterfly operations on the incoming data to generate the resulting filtered output.

Because this FFT algorithm is implemented as a single inner loop representing a radix-4 butterfly, we can use the automatic pipelining capabilities of the Impulse C compiler to try a variety of pipeline strategies. Pipelining can introduce a high degree of parallelism in the generated logic, allowing us to achieve higher throughput at the expense of additional hardware. This pipeline itself may be implemented in a variety of ways, depending on how fast we wish to clock the FFT.

Another aspect to this algorithm (and others like it) is that it can be implemented using either fixed- or floating-point math. For a high-end processor such as an Intel Pentium or AMD Opteron, the choice of floating point is obvious. But for an FPGA, floating point may or may not make sense given the lack of dedicated floating-point units and the corresponding expense (in terms of FPGA slice count) of generating additional hardware.

Fortunately, the Impulse C tools allow the use of either fixed- or floating-point operations, and Xilinx tools are capable of instantiating floating-point FPGA cores from the generated logic. This makes performance comparisons relatively easy. In addition, the Impulse tools allow pipelining for loops to be selectively enabled, and pipeline size and depth to be indirectly controlled through a delay constraint.

Table 1 shows the results of two different optimization and pipelining strategies for the 32-bit FFT, for both the fixed- and floating-point versions. We generated results using the Impulse C version 2.10 tools in combination with Xilinx ISE™

FFT Data Type	Pipelining Enabled?	Slices Used	FFs Used	LUTs Used	Clock Frequency	Total FFT Cycles
Fixed Point	No	3,207	2,118	5,954	80 MHz	1,536
Fixed Point	Yes	2,810	2,347	5,091	81 MHz	261
Floating Point	No	10,917	7,298	20,153	88 MHz	10,496
Floating Point	Yes	10,866	7,610	19,855	74 MHz	269

Table 1 – 32-bit FFT optimization results for fixed- and floating-point versions, with and without pipelining enabled

Stage Delay Constraint	Slices Used	FFs Used	LUTs Used	Clock Frequency	Pipeline Stages
300	1,360	1,331	2,186	60 MHz	5
200	1,377	1,602	2,209	85 MHz	7
150	1,579	2,049	2,246	99 MHz	9
100	1,795	2,470	2,392	118 MHz	11
75	2,305	3,334	2,469	139 MHz	15

Table 2 – 16-bit image filter optimization results for various pipelining and stage delay combinations

software version 8.1, and targeting a Xilinx Virtex™-4 LX-25 device. Many more choices are actually possible during iterative optimization, depending on the goals of the algorithm and the clocking and size constraints of the overall system. In this case, there is a clear benefit from enabling pipelining in the generated logic. There is also, as expected, a significant area penalty for making use of floating-point operations and a significant difference in performance when pipelining is enabled.

Table 2 shows a similar test performed using a 5x5 kernel image filter. This filter, which has been described using two parallel C-language processes, demonstrates how a variety of different pipeline optimization strategies (again specified using a delay constraint provided by Impulse C) can quickly evaluate size and speed trade-offs for a complex algorithm. For all cases shown, the image filter data rate (the rate at which pixels are processed) is exactly one half the clock rate.


For software applications targeting FPGAs, the ability to exploit parallelism (through instruction scheduling, pipelining, unrolling, and other automated or manual techniques) is critical to achieving

quantifiable improvements over traditional processors.

But parallelism is not the whole story; data movement can actually become a more significant bottleneck when using FPGAs. For this reason, you must balance the acceleration of critical computations and inner-code loops against the expense of moving data between hardware and software.

Fortunately, modern tools for FPGA compilation include various types of analysis features that can help you more clearly understand and respond to these issues. In addition, the ability to rapidly prototype alternative algorithms – perhaps using very different approaches to data management such as data streaming, message passing, or shared memory – can help you more quickly converge on a practical implementation.

Conclusion

With tools providing C-to-hardware compilation and optimization for Xilinx FPGAs, software and hardware developers can cross the abstraction gap and create faster, more efficient prototypes and end products. To discover what C-to-hardware technologies can do for you, visit www.xilinx.com/esl or www.impulsec.com/xilinx. 

Accelerating Architecture Exploration for FPGA Selection and System Design

You can create an optimized specification that achieves performance, reliability, and cost goals for use in production systems.

by Deepak Shankar
President and CEO
Mirabilis Design Inc.
dshankar@mirabilisdesign.com

Performance analysis and early architecture exploration ensures that you will select the right FPGA platform and achieve optimal partitioning of the application onto the fabric and software. This early exploration is referred to as rapid visual prototyping. Mirabilis Design's VisualSim software simulates the FPGA and board using models that are developed quickly using pre-built, parameterized modeling libraries in a graphical environment.

These library models resemble the elements available on Xilinx® FPGAs, including PowerPC™, MicroBlaze™, and PicoBlaze™ processors; CoreConnect; DMA; interrupt controllers; DDR; block RAM; LUTs; DSP48E; logic operators; and fabric devices. The components are connected to describe a given Xilinx Virtex platform and simulated for different operating conditions such as traffic, user activity, and operating environment.

More than 200 standard analysis outputs include latency, utilization, throughput, hit ratio, state activity, context

switching, power consumption, and processor stalls. VisualSim accelerates architecture exploration by reducing typical model development time from months to days.

I can illustrate the advantages of early architecture exploration with an example from one of our customers, who was experiencing difficulty with a streaming media processor implemented using a Virtex™-4 device. The design could not achieve the required performance and was dropping every third frame. Utilization at all of the individual devices was below 50%. A visual simulation that combined both the peripheral and the FPGA identified that the video frames were being transferred at the same clock sync as the audio frames along a shared internal bus.

As the project was in the final stages of development, making architecture changes to address the problem would have delayed shipment by an additional six months. Further refinement of the VisualSim model found that by giving the audio frames a higher priority, the design could achieve the desired performance, as the audio frames would also be available for processing. The project schedule was delayed by approximately 1.5 months.

If the architecture had been modeled early in the design cycle, the design cycle could have been reduced by 3 months, eliminating the 1.5-month re-spin to get to market approximately 5 months sooner. Moreover, with a utilization of 50%, control processing could have been moved to the same FPGA. This modification might have saved one external processor, a DDR controller, and one less memory board.

Rapid Visual Prototyping

Rapid visual prototyping can help you make better partitioning decisions. Evaluations with performance and architectural models can help eliminate clearly inferior choices, point out major problem areas, and evaluate hardware/software trade-offs. Simulation is cheaper and faster than building hardware prototypes and can also help with software development, debugging, testing, documentation, and maintenance. Furthermore, early partnership with customers using visual prototypes improves feedback on design decisions, reducing time to market and increasing the likelihood of product success (Figure 1).

A design-level specification captures a new or incremental approach to improve system throughput, power, latency, utiliza-

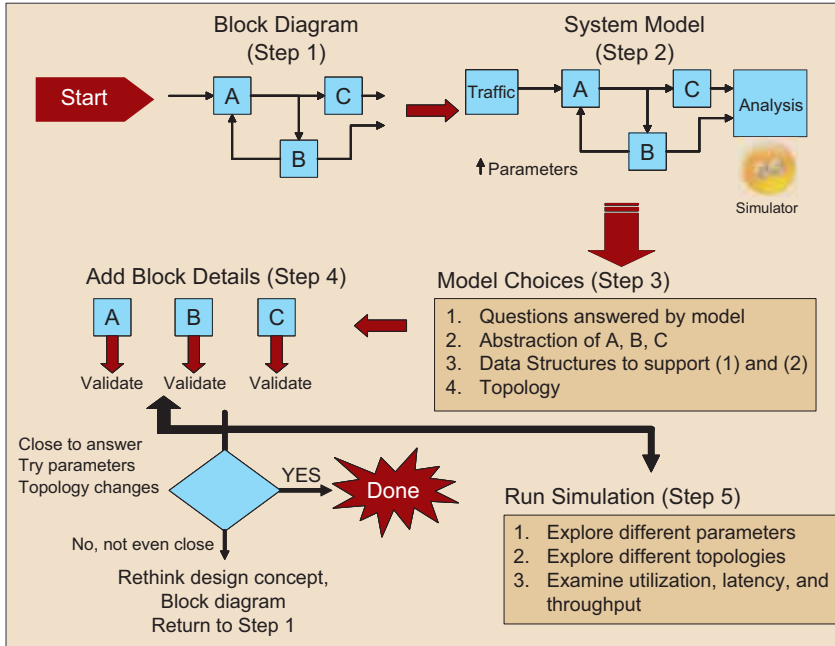


Figure 1 – Translating a system concept into rapid visual prototyping

tion, and cost; these improvements are typically referred to as price/power/performance trade-offs. At each step in the evolution of a design specification, well-intentioned modifications or improvements may significantly alter the system requirements. The time required to evaluate a design modification before or after the system design process has started can vary dramatically, and a visual prototype will reduce evaluation time.

To illustrate the use of the rapid visual prototype, let's consider a Layer 3 switch implemented using a Virtex FPGA. The Layer 3 switch is a non-blocking switch and the primary consideration is to maintain total utilization across the switch.

Current Situation

In product design, three factors are certain: specifications change, non-deterministic traffic creates performance uncertainty, and Xilinx FPGAs get faster. Products operate in environments where the processing and resource consumption are a function of the incoming data and user operations. FPGA-based systems used for production must meet quality, reliability, and performance metrics to address customer requirements. What is the optimal distribution of tasks into hardware acceleration and software on FPGAs and other

board devices? How can you determine the best FPGA platform to meet your product requirements and attain the highest performance at the lowest cost?

Early Exploration Solution

VisualSim provides pre-built components that are graphically instantiated to describe hardware and software architectures. The applications and use cases are described as flow charts and simulated on the VisualSim model of the architecture using multiple traffic profiles. This approach reduces the model construction burden and allows you to focus on analysis and interpretation of results. It also helps you optimize product architectures by running simulations with application profiles to explore FPGA selection; hardware versus software decisions; peripheral devices versus performance; and partitioning of behavior on target architectures.

Design Optimization

You can use architecture exploration (Figure 2) to optimize every aspect of an FPGA specification, including:

- Task distribution on MicroBlaze and PowerPC processors
- Sizing the processors

- Selecting functions requiring a co-processor
- Determining optimal interface speeds and pins required
- Exploring block RAM allocation schemes, cache and RAM speeds, off-chip buffering, and impact of redundant operators

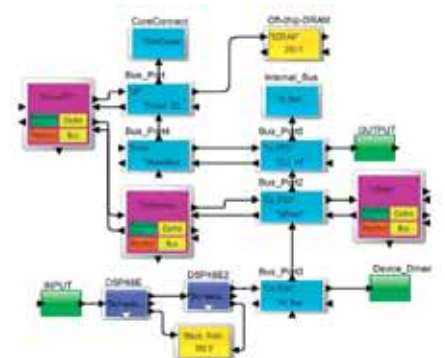


Figure 2 – Architecture model of the FPGA platform and peripheral using VisualSim FPGA components

An analysis conducted using VisualSim includes packet size versus latency, protocol overhead versus effective bandwidth, and resource utilization.

In reference to the Layer 3 example, your decisions would include using:

- The on-chip PowerPC or external processor for routing operations
- Encryption algorithms using the DSP function blocks or fabric multipliers and adders
- A dedicated MicroBlaze processor for traffic management or fabric
- PowerPC for control or proxy rules processing
- TCP offload using an external co-processor or MicroBlaze processor

Can a set of parallel PicoBlaze processors with external SDRAM support in-line spyware detection? What will the performance be when the packet size changes from 256 bytes to 1,512 bytes? How can you plan for future applications such as mobile IP?

You can extend the exploration to consider the interfaces between the FPGA and board peripherals, such as SDRAM. As the

PowerPC will be sharing the bus with the MicroBlaze processor, the effective bus throughput is a function of the number of data requests and the size of the local block RAM buffers. For example, you could enhance the MicroBlaze processor with a co-processor to do encryption at the bit level in the data path. You could also use the CoreConnect bus to connect the peripheral SDRAM to the PowerPC while the DDR2 is used for the MicroBlaze processor.

You can reuse the VisualSim architecture model for exploration of the software design, identifying high-resource consumption threads, balancing load across multiple MicroBlaze processors, and splitting operations into smaller threads. If a new software task or thread has data-dependent priorities, exploration of the priorities and task-arrival time on the overall processing is a primary modeling question. If you change the priority on a critical task, will this be sufficient to improve throughput and reduce task latency? In most cases, this will be true, but there may be a relative time aspect to a critical task that can reduce latencies on lower priority tasks such that both benefit from the new ordering. If peak processing is above 80% for a system processing element, then the system may be vulnerable to last-minute tasks added, or to future growth of the system itself.

Model Construction

System modeling of the Layer 3 switch (Figure 3) starts by compiling the list of functions (independent of implementation), expected processing time, resource consumption, and system performance metrics. The next step is to capture a flow diagram in VisualSim using a graphical block diagram editor (Figure 3). The flow diagrams are UML diagrams annotated with timing information. The functions in the flow are represented as delays; timed queues represent contention; and algorithms handle the data movement. The flow diagram comprises data processing, control, and any dependencies.

Data flow includes flow and traffic management, encryption, compression, routing, proxy rules, and TCP protocol

handling. The control path contains the controller algorithm, branch decision trees, and weighted polling policies. VisualSim builds scenarios to simulate the model and generate statistics. The scenarios are multiple concurrent data flows such as connection establishment (slow

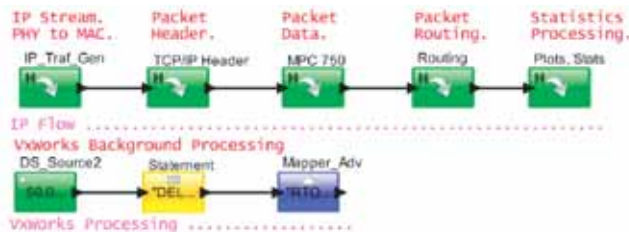


Figure 3 – Flow chart describing the application flow diagram in VisualSim

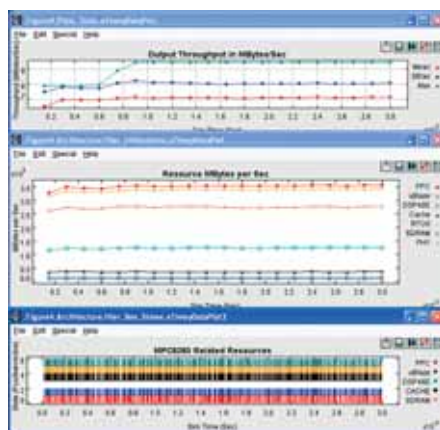


Figure 4 – Analysis output for the Layer 3 switch design

path); in-line data transfer after setup of secure channel (fast path); and protocol- and data-specific operation sequences based on data type identification.

You can use this model of the timed flow diagram for functional correctness and validation of the flows. VisualSim uses random traffic sequences to trigger the model. The traffic sequences are defined data structures in VisualSim; a traffic generator emulates application-specific traffic. This timed flow diagram selects the FPGA platform and conducts the initial hardware and software partitioning. The flow diagram model defines the FPGA components and peripheral hardware using the FPGA Modeling Toolkit.

The functions of the flow diagram are mapped to these architecture components. For each function, VisualSim automatically collects the end-to-end delay and number of packets processed in a time period. For the architecture, VisualSim plots the average processing time, utilization, and effective throughput (Figure 4). These metrics are matched against the requirements. Exploration of the mapping and architecture is possible by varying the link and replacing the selected FPGA with other FPGAs.

The outcome of this effort will be selecting the right FPGA family, correctly sizing the peripherals and the right number of block RAMs, DSP blocks, and MicroBlaze processors. You can add overhead to the models to capture growth requirements and ensure adequate performance.

Conclusion

Early architecture exploration ensures a highly optimized product for quality, reliability, performance, and cost. This provides direction for implementation plans, reduces the amount of tests you need to conduct, and has the ability to shrink the development cycle by almost 30%.

VisualSim libraries of standard FPGA components, flow charts defining the behavior, traffic models, and pre-built analysis probes ensure that system design is no longer time consuming, difficult to perform, and providing questionable results. The reduction in system modeling time and availability of standard component models provides a single environment for designers to explore both hardware and software architectures.

For a free 21-day trial of the FPGA Modeling Toolkit, including the MicroBlaze and PowerPC models, register at www.mirabilisdesign.com/webpages/evaluation/mdi_evaluation.htm. To learn more about VisualSim, visit www.mirabilisdesign.com, where there are models embedded in the HTML pages. You can modify parameters and execute from within your web browser without having to download custom software.

Rapid Development of Control Systems with Floating Point

Generate high-quality synthesizable HDL with the Mobius compiler.

```

procedure ir(m: real; a: real; b: real; c: real; d: real; e: real; f: real; g: real; h: real; i: real; j: real; k: real; l: real; m: real; n: real; o: real; p: real; q: real; r: real; s: real; t: real; u: real; v: real; w: real; x: real; y: real; z: real; aa: real; ab: real; ac: real; ad: real; ae: real; af: real; ag: real; ah: real; ai: real; aj: real; ak: real; al: real; am: real; an: real; ao: real; ap: real; aq: real; ar: real; as: real; at: real; au: real; av: real; aw: real; ax: real; ay: real; az: real; ba: real; bb: real; bc: real; bd: real; be: real; bf: real; bg: real; bh: real; bi: real; bj: real; bk: real; bl: real; bm: real; bn: real; bo: real; bp: real; bq: real; br: real; bs: real; bt: real; bu: real; bv: real; bw: real; bx: real; by: real; bz: real; ca: real; cb: real; cc: real; cd: real; ce: real; cf: real; cg: real; ch: real; ci: real; cj: real; ck: real; cl: real; cm: real; cn: real; co: real; cp: real; cq: real; cr: real; cs: real; ct: real; cu: real; cv: real; cw: real; cx: real; cy: real; cz: real; da: real; db: real; dc: real; dd: real; de: real; df: real; dg: real; dh: real; di: real; dj: real; dk: real; dl: real; dm: real; dn: real; do: real; dp: real; dq: real; dr: real; ds: real; dt: real; du: real; dv: real; dw: real; dx: real; dy: real; dz: real; ea: real; eb: real; ec: real; ed: real; ee: real; ef: real; eg: real; eh: real; ei: real; ej: real; ek: real; el: real; em: real; en: real; eo: real; ep: real; eq: real; er: real; es: real; et: real; eu: real; ev: real; ew: real; ex: real; ey: real; ez: real; fa: real; fb: real; fc: real; fd: real; fe: real; ff: real; fg: real; fh: real; fi: real; fj: real; fk: real; fl: real; fm: real; fn: real; fo: real; fp: real; fq: real; fr: real; fs: real; ft: real; fu: real; fv: real; fw: real; fx: real; fy: real; fz: real; ga: real; gb: real; gc: real; gd: real; ge: real; gf: real; gg: real; gh: real; gi: real; gj: real; gk: real; gl: real; gm: real; gn: real; go: real; gp: real; gq: real; gr: real; gs: real; gt: real; gu: real; gv: real; gw: real; gx: real; gy: real; gz: real; ha: real; hb: real; hc: real; hd: real; he: real; hf: real; hg: real; hh: real; hi: real; hj: real; hk: real; hl: real; hm: real; hn: real; ho: real; hp: real; hq: real; hr: real; hs: real; ht: real; hu: real; hv: real; hw: real; hx: real; hy: real; hz: real; ia: real; ib: real; ic: real; id: real; ie: real; if: real; ig: real; ih: real; ii: real; ij: real; ik: real; il: real; im: real; in: real; io: real; ip: real; iq: real; ir: real; is: real; it: real; iu: real; iv: real; iw: real; ix: real; iy: real; iz: real; ja: real; jb: real; jc: real; jd: real; je: real; jf: real; jg: real; jh: real; ji: real; jj: real; jk: real; jl: real; jm: real; jn: real; jo: real; jp: real; jq: real; jr: real; js: real; jt: real; ju: real; jv: real; jw: real; jx: real; jy: real; jz: real; ka: real; kb: real; kc: real; kd: real; ke: real; kf: real; kg: real; kh: real; ki: real; kj: real; kk: real; kl: real; km: real; kn: real; ko: real; kp: real; kq: real; kr: real; ks: real; kt: real; ku: real; kv: real; kw: real; kx: real; ky: real; kz: real; la: real; lb: real; lc: real; ld: real; le: real; lf: real; lg: real; lh: real; li: real; lj: real; lk: real; ll: real; lm: real; ln: real; lo: real; lp: real; lq: real; lr: real; ls: real; lt: real; lu: real; lv: real; lw: real; lx: real; ly: real; lz: real; ma: real; mb: real; mc: real; md: real; me: real; mf: real; mg: real; mh: real; mi: real; mj: real; mk: real; ml: real; mm: real; mn: real; mo: real; mp: real; mq: real; mr: real; ms: real; mt: real; mu: real; mv: real; mw: real; mx: real; my: real; mz: real; na: real; nb: real; nc: real; nd: real; ne: real; nf: real; ng: real; nh: real; ni: real; nj: real; nk: real; nl: real; nm: real; nn: real; no: real; np: real; nq: real; nr: real; ns: real; nt: real; nu: real; nv: real; nw: real; nx: real; ny: real; nz: real; oa: real; ob: real; oc: real; od: real; oe: real; of: real; og: real; oh: real; oi: real; oj: real; ok: real; ol: real; om: real; on: real; oo: real; op: real; oq: real; or: real; os: real; ot: real; ou: real; ov: real; ow: real; ox: real; oy: real; oz: real; pa: real; pb: real; pc: real; pd: real; pe: real; pf: real; pg: real; ph: real; pi: real; pj: real; pk: real; pl: real; pm: real; pn: real; po: real; pp: real; pq: real; pr: real; ps: real; pt: real; pu: real; pv: real; pw: real; px: real; py: real; pz: real; qa: real; qb: real; qc: real; qd: real; qe: real; qf: real; qg: real; qh: real; qi: real; qj: real; qk: real; ql: real; qm: real; qn: real; qo: real; qp: real; qq: real; qr: real; qs: real; qt: real; qu: real; qv: real; qw: real; qx: real; qy: real; qz: real; ra: real; rb: real; rc: real; rd: real; re: real; rf: real; rg: real; rh: real; ri: real; rj: real; rk: real; rl: real; rm: real; rn: real; ro: real; rp: real; rq: real; rr: real; rs: real; rt: real; ru: real; rv: real; rw: real; rx: real; ry: real; rz: real; sa: real; sb: real; sc: real; sd: real; se: real; sf: real; sg: real; sh: real; si: real; sj: real; sk: real; sl: real; sm: real; sn: real; so: real; sp: real; sq: real; sr: real; ss: real; st: real; su: real; sv: real; sw: real; sx: real; sy: real; sz: real; ta: real; tb: real; tc: real; td: real; te: real; tf: real; tg: real; th: real; ti: real; tj: real; tk: real; tl: real; tm: real; tn: real; to: real; tp: real; tq: real; tr: real; ts: real; tt: real; tu: real; tv: real; tw: real; tx: real; ty: real; tz: real; ua: real; ub: real; uc: real; ud: real; ue: real; uf: real; ug: real; uh: real; ui: real; uj: real; uk: real; ul: real; um: real; un: real; uo: real; up: real; uq: real; ur: real; us: real; ut: real; uu: real; uv: real; uw: real; ux: real; uy: real; uz: real; va: real; vb: real; vc: real; vd: real; ve: real; vf: real; vg: real; vh: real; vi: real; vj: real; vk: real; vl: real; vm: real; vn: real; vo: real; vp: real; vq: real; vr: real; vs: real; vt: real; vu: real; vv: real; vw: real; vx: real; vy: real; vz: real; wa: real; wb: real; wc: real; wd: real; we: real; wf: real; wg: real; wh: real; wi: real; wj: real; wk: real; wl: real; wm: real; wn: real; wo: real; wp: real; wq: real; wr: real; ws: real; wt: real; wu: real; wv: real; ww: real; wx: real; wy: real; wz: real; xa: real; xb: real; xc: real; xd: real; xe: real; xf: real; xg: real; xh: real; xi: real; xj: real; xk: real; xl: real; xm: real; xn: real; xo: real; xp: real; xq: real; xr: real; xs: real; xt: real; xu: real; xv: real; xw: real; xx: real; xy: real; xz: real; ya: real; yb: real; yc: real; yd: real; ye: real; yf: real; yg: real; yh: real; yi: real; yj: real; yk: real; yl: real; ym: real; yn: real; yo: real; yp: real; yq: real; yr: real; ys: real; yt: real; yu: real; yv: real; yw: real; yx: real; yy: real; yz: real; za: real; zb: real; zc: real; zd: real; ze: real; zf: real; zg: real; zh: real; zi: real; zj: real; zk: real; zl: real; zm: real; zn: real; zo: real; zp: real; zq: real; zr: real; zs: real; zt: real; zu: real; zv: real; zw: real; zx: real; zy: real; zz: real;
const a2: t = 0.01;
const a3: t = 0.12;
const b3: t = 0.5;
var u, x1, x2, x3: t;
seq
x1:=0.0; x2:=0.0; x3:=0.0;
while true do
seq
cu ? u; (* read input *)
x1,x2,x3 := x2,x3, u-(t(a3*x1)+(a2*x2)+x3); (* calc states *)
cy ! t(b3*x1)+x2; (* write output *)
end
end;
par (* testbench *)
iir(c1,c2);
while true do seq c1 ! 1.0; c2 ? y; write(" y=" & y) end (* get step response of filter *)
end;

```

by Per Ljung
 President
 Codetronix LLC
ljung@codetronix.com

With the advent of ever-larger FPGAs, both software algorithm developers and hardware designers need to be more productive. Without an increase in abstraction level, today's complex software systems cannot be realized, both because of a lack of resources (time, money, developers) and an inability to manage the complexity. Electronic system level (ESL) design offers a solution where an abstraction level higher than HDL will result in higher productivity, higher performance, and higher quality.

In this article, I'll show you how to use the high-level multi-threaded language Mobius to quickly and efficiently implement embedded control systems on FPGAs. In particular, I'll demonstrate the implementation of a floating-point embedded control system for a Xilinx® Virtex™-II FPGA in a few lines of Mobius source code.

Mobius lets software engineers create efficient and robust hardware/software systems, while hardware engineers become much more productive.

Mobius

Mobius is a tiny high-level multi-threaded language and compiler designed for the rapid development of hardware/software embedded systems. A higher abstraction level and ease of use lets you achieve greater design productivity compared to traditional approaches. At the same time you are not compromising on quality of results, since benchmarks show that Mobius-generated circuits match the best hand designs in terms of throughput and resources for both compact and single-cycle pipelined implementations.

Developing embedded systems with Mobius is much more like software development using a high-level language than hardware development using assembly-like HDL. Mobius has a fast transaction-level simulator so that the code/test/debugging cycle is much faster than traditional HDL iterations. The Mobius compiler translates all test benches into HDL, so for a quick verification simply compare the Mobius simulation with the HDL simulation using the same test vectors. The compiler-generated HDL is assembled from a set of Lego-like primitives connected using handshaking channels. As a result of handshaking, the circuit is robust and correct by construction.

Mobius lets software engineers create efficient and robust hardware/software systems, while hardware engineers become much more productive. With less than 200 lines of Mobius source, users have implemented FIR filters, FFT transforms, JPEG encoding, and DES and AES encryption, as well as hardware/software systems with PowerPC™ and MicroBlaze™ processors.

Parallelism is the key to obtaining high performance on FPGAs. Mobius enables both compact sequential circuits, latency-intolerant inelastic pipelined circuits, and parallel latency-tolerant elastic pipelined circuits. Using the keywords “seq” and “par,” the Mobius language allows basic blocks to run in sequence or in parallel, respectively. Parallel threads communicate with message passing channels, where the keywords “?”

and “!” are used to read and write over a channel that waits until both the reader and writer are ready before proceeding. Message passing obviates the need for low-level locks, mutexes, and semaphores. The Mobius compiler also identifies common parallel programming errors such as illegal parallel read/write or write/write statements.

The Mobius compiler generates synthesizable HDL using Xilinx-recommended structures, letting the Xilinx synthesizer efficiently infer circuits. For instance, variables are mapped to flip-flops and arrays are mapped to block RAM. Because Mobius is a front end to the Xilinx design flow, Mobius supports all current Xilinx targets, including the PowerPC and DSP units found on the Virtex-II Pro, Virtex-4, and Virtex-5 device families. The generated HDL is readable and graphically documented, showing hierarchical control and dataflow relationships. Application notes show how to create hardware/software systems communicating over the on-chip peripheral and Fast Simplex Link buses.

Handshaking allows you to ignore low-level timing, as handshaking determines the control and dataflow. However, it is very easy to understand the timing model of Mobius-generated HDL. Every Mobius signal has a request, acknowledge, and data component where the req/ack bits are used for handshaking. For a scalar variable instantiated as a flip-flop, a read takes zero cycles and a scalar write takes one cycle. An assignment statement takes as many cycles as the sum of its right-hand-side (RHS) and left-hand-side (LHS) expressions. An assignment with scalar RHS and LHS expressions therefore takes one cycle to execute. Channel communications take an unknown number of cycles (it waits until both reader and write are ready). A sequential block of statements takes as many cycles as the sum of its children, and a parallel block of statements takes the maximum number of cycles of its children.

Mobius has native support for parameterized signed and unsigned integers, fixed

point and floating point. The low-latency math libraries are supplied as Mobius source code. The fixed point add, sub, and mul operators are zero-cycle combinatorial functions. The floating-point add and sub operators take four cycles, the mul operators take two cycles, and the div operator is iterative-dependent on the size of the operands.

Infinite Impulse Response Filter

As an example of an embedded control system, let's look at how you can use Mobius to quickly implement an infinite impulse response (IIR) filter. I will investigate several different architectures for throughput, resources, and quantization using parameterized fixed- and floating-point math.

The IIR is commonly found in both control systems and signal processing. A discrete time proportional-integral-derivative (PID) controller and lead-lag controller can be expressed as an IIR filter. In addition, many common signal processing filters such as Elliptic, Butterworth, Chebychev, and Bessel are implemented as IIR filters. Numerically, an IIR comprises an impulse transfer function $H(z)$ with q poles and p zeros:

$$H(z) = \frac{\sum_{i=0}^P b_i z^{-i}}{1 - \sum_{k=1}^Q a_k z^{-k}}$$

This can also be expressed as a difference equation:

$$y(n) = \sum_{i=0}^P b_i x(n-i) + \sum_{k=1}^Q a_k y(n-k)$$

The IIR has several possible implementations. The direct form I is often used by fixed-point IIR filters because a larger single adder can prevent saturation. The direct form II is often used by floating-point IIR filters because this uses fewer states and the adders are not as sensitive to saturation. The cascade canonical form has the lowest quantization sensitivity, but at the cost of additional resources. For example, if $p = q$ and p is even, then the direct form I and II

```

procedure testbench();

type t = sfixed(6,8); (* type t = float(8,24); *)
var c1,c2:chan of t;
var y:t;

procedure iir(in cu:chan of t; out cy:chan of t);
const a2: t = 0.01;
const a3: t = 0.12;
const b3: t = 0.5;
var u,x1,x2,x3:t;
seq
x1:=0.0; x2:=0.0; x3:=0.0;
while true do
seq
cu ? u; (* read input *)
x1,x2,x3 := x2,x3, u-(t(a3*x1)+t(a2*x2)-x3); (* calc states *)
cy ! t(b3*x1)+x2; (* write output *)
end
end;

par (* testbench *)
iir(c1,c2);
while true do seq c1 ! 1.0; c2 ? y; write(" y=",y) end (* get step response of filter *)
end;

```

Figure 1 – Mobius source code and test bench for IIR filter

implementations only require $2p + 1$ constant multipliers, while the cascade requires $5p/2$ constant multipliers.

Because of the many trade-offs, it is clear that numerical experiments are necessary to determine a suitable implementation.

IIR Implementation in Mobius

Let's select a third-order IIR filter where $b_0 = 0$, $b_1 = 0$, $b_2 = 1$, $b_3 = 0.5$, $a_1 = -1$, $a_2 = 0.01$, and $a_3 = 0.12$. The difference equation can be easily written as the multi-threaded Mobius program shown in Figure 1.

The Mobius source defines the IIR filter as a procedure and a test bench to exercise it. Inside `iir()`, a perpetual loop continuously reads a new reference value `u`, updates the states `x1`, `x2`, `x3`, and then calculates and writes the output. The IIR uses four (fixed- or floating-point) adders and three (fixed- or floating-point) multipliers. Note how the

values of all states are simultaneously updated. The test bench simply sends a constant reference value to the filter input, reads the filter output, and writes that resulting step value to `stdout`. By separating the IIR filter from the test bench, just the IIR filter can be synthesized.

Fixed Point or Floating Point?

Floating-point math allows for large dynamic range but typically requires considerable resources. Using fixed point instead can result in a much smaller and faster design, but with trade-offs in stability, precision, and range.

Both fixed- and floating-point math are fully integrated into the Mobius language, making things easy for you to mix and match various-sized fixed- and floating-point operators. In the Mobius source, the type "t" defines a parameter-

```

y=0.000000
y=1.000000
y=2.500000
y=4.000000
y=5.492188
y=6.855469
y=8.031250
y=9.023438
y=9.832031
y=10.472656
y=10.964844
y=11.335938
y=11.609375
y=11.800781
y=11.933594
y=12.019531
y=12.078125
y=12.109375
y=12.121094
y=12.121094
y=12.117188
y=12.109375
y=12.097656
y=12.085938
y=12.074219
y=12.062500
y=12.054688
y=12.046875
y=12.042969
y=12.035156
y=12.031250
y=12.027344
y=12.027344
y=12.027344
y=12.027344
y=12.027344

```

Figure 2 – HDL simulation of step response running test bench

ized fixed- or floating-point size. By changing this single definition, the compiler will automatically use the selected parameterization of operands and math operators in the entire application. For instance, the signed fixed-point type definition `sfixed(6,8)` uses 14 bits, where 6 bits describe the whole number and 8 bits the fraction. The floating-point type definition `float(6,8)` uses 15 bits, with 1 sign bit, 6 exponent bits, and 8 mantissa bits.

Parallel or Sequential Architecture?

Using parallel execution of multiple threads, an FPGA can achieve tremendous speedups over a sequential implementation. Using a pipelined architecture requires more resources, while a sequential implementation can share resources using time-multiplexing.

The `iir()` procedure computes all expressions in a maximally parallel manner and does not utilize any resource sharing. You can also create alternate architectures that use pipelined operators and statements for higher speed, or use resource sharing for smaller resources and slower performance.

Mobius-Generated VHDL/Verilog

Using the Mobius compiler to generate HDL from the Mobius source (Figure 1), ModelSim (from Mentor Graphics) can simulate the step response (Figure 2).

I synthesized several Mobius implementations of the IIR filter for a variety of math types and parameterizations (Table 1) using Xilinx ISE™ software version 7.1i (service pack 4) targeting a Virtex-II Pro XC2VP7 FPGA. The Xilinx ISE synthesizer efficiently maps the HDL behavioral structures onto combinatorial logic and uses dedicated hardware (for example, multipliers) where appropriate. A constraint file was not used. As you can see, the fixed-point implementations are considerably smaller and faster than the floating-point implementations.

An alternate architecture using pipelining reduces the cycle time to 1 cycle for the fixed-point and 18 cycles for the floating-point

implementations, with similar resources and clock speeds.

You can also use an alternate architecture with time-multiplexed resource sharing to make the design smaller (but slower). Sharing multipliers and adders in the larger floating-point IIR design results in a design needing only 1,000 slices and 4 multipliers at 60 MHz, but the IIR now has a latency of 48 cycles. This is a resource reduction of 3x and a throughput reduction of 2x.

Conclusion

In this article, I've shown how Mobius users can achieve greater design productivity, as exemplified by the rapid development of several fixed- and floating-point IIR filters. I've also shown how you can quickly design, simulate, and synthesize several architectures using compact, pipelined, and time-multiplexed resource sharing to quantitatively investigate resources, throughput, and quantization effects.

The Mobius math libraries for fixed- and floating-point math enable you to quickly implement complex control and signal-processing algorithms. The Mobius source for the basic IIR filter is about 25 lines of code, whereas the generated HDL is about 3,000-8,000 lines, depending on whether fixed point or floating point is generated. The increase in productivity using Mobius instead of hand-coded HDL is significant.

Using Mobius allows you to rapidly develop high-quality solutions. For more information about using Mobius in your next design, visit www.codetronix.com.

	sfixed(6,8)	sfixed(8,24)	float(6,8)	float(8,24)
Resources	66 slices 66 FFs 109 LUTs 38 IOBs 3 mult18 x 18s	287 slices 135 FFs 517 LUTs 74 IOBs 12 mult18 x 18s	798 slices 625 FFs 1,329 LUTs 40 IOBs 3 mult18 x 18s	2,770 slices 1,345 FFs 4,958 LUTs 76 IOBs 12 mult18 x 18s
Clock	89 MHz	56 MHz	103 MHz	61 MHz
Cycles	2	2	26	26

Table 1 – Synthesis results of `iir()`

Supporting Your Future
HUNT ENGINEERING
USB connected Programmable FPGA systems

V-II Pro PowerPC

- Virtex-II Pro XC2VP7
- 256 Mbytes DDR Memory
- Configurable digital I/Os
- PowerPC boot FLASH
- USB 2 or Standalone

Software Defined Radio

- Virtex-II FPGA 1M gates
- 2 ch 125Msps A/D and D/A
- TI C6203 DSP
- 32Mbytes SDRAM
- Configurable Digital I/O
- USB 2 or Standalone

Imaging with Virtex-4FX

- Virtex-4 FPGA FX12
- 128Mbytes DDR Memory
- CameraLink connection
- VHDL and PowerPC Imaging Libs
- USB 2 or Standalone

Programmable hardware with cables, device drivers, loading tools, examples and Power Supply. Systems can be used connected to a PC using USB, or can function standalone (without USB) using the initialisation PROMs.

sales@hunteng.co.uk
+44 (0)1278 760188
www.hunt-rtg.com

Hardware/Software Partitioning from Application Binaries

A unique approach to hardware/software co-design empowers embedded application developers.

by Susheel Chandra, Ph.D.
President/CEO
Binachip, Inc.
schandra@binachip.com

Computationally intense real-time applications such as Voice over IP, video over IP, 3G and 4G wireless communications, MP3 players, and JPEG and MPEG encoding/decoding require an integrated hardware/software platform for optimal performance. Parts of the application run in software on a general-purpose processor, while other portions must run on application-specific hardware to meet performance requirements. This methodology, commonly known as hardware/software partitioning or co-design, has caused an increasing number of software applications to be migrated to system-on-a-chip (SOC) platforms.

FPGAs have emerged as the SOC platform of choice, particularly in the fast-paced world of embedded computing. FPGAs enable rapid, cost-effective product development cycles in an environment where target markets are constantly shifting and standards continuously evolving. Several families of platform FPGAs are available from Xilinx. Most of these offer processing capabilities, a programmable fabric, memory, peripheral devices, and connectivity to bring data into and out of the FPGA. They provide embedded application developers with a basic hardware platform on which to build end applications, with minimal amounts of time and resources spent on hardware considerations.

DSP applications add another dimension of complexity by including MATLAB source code in addition to C/C++ and assembly language. Clearly, any tool that does not support multiple source languages will fall short.

Source-to-Hardware Tools

Several tools on the market allow you to translate high-level source descriptions in C/C++ or MATLAB into VHDL or Verilog. Most of these tools fall into the category of “behavioral synthesis” tools and are designed for hardware designers creating a hardware implementation for a complete algorithm. These tools require you to have some intrinsic knowledge of hardware design and to use only a restricted subset of the source language conducive to hardware translation. In some cases they also require you to learn new language constructs. This creates a significant learning curve, which further pushes out time to market.

More recently, some companies have introduced tools for embedded application developers. These tools promote a co-design methodology and allow you to perform hardware/software partitioning. However, they suffer from some of the same limitations of behavioral synthesis tools – you need to learn hardware-specific constructs for the tool to generate optimized hardware. This imposes a fundamental barrier for a tool targeted at embedded application developers, who in most cases are software engineers with no or limited knowledge of hardware design.

Another basic requirement for tools intended for application developers is that they support applications developed in multiple source languages. An embedded application of even moderate complexity will comprise parts written in C/C++ (calling on a suite of library functions) and other parts written in assembly language (to optimize performance). DSP applications add another dimension of complexity by including MATLAB source code in addition to C/C++ and assembly language. Clearly, any tool that does not support multiple source languages will fall short.

A new unique approach to hardware/software co-design is to start with the binary of the embedded application. It not only over-

comes all of the previously mentioned issues but also provides other significant benefits. A new commercial tool called BINACHIP-FPGA from Binachip, Inc. successfully uses this approach.

Partitioning from Binaries

The executable code or binary compiled for a specific processor platform is the least common denominator representation of an embedded application written in multiple source languages. Combined with the instruction set architecture of the processor, it provides a basis for extremely accurate hardware/software partitioning at a very fine grain level. The close-to-hardware nature of binaries also allows for better hardware/software partitioning decisions, as illustrated in the following examples.

Bottlenecks Inside Library Functions

One obvious consideration for hardware/software partitioning is computational complexity. Moving computationally intense functions into hardware can result in a significant performance improvement. Any embedded application typically relies on a suite of library functions for basic mathematical operations such as floating-point multiply/divide, sine, and cosine. Some of these functions are fairly compute-heavy and could be called millions of times during the normal execution of an application, resulting in a bottleneck.

Moving one or more of these functions into hardware could easily provide a 10x-50x speedup in execution time, depending on their complexity and how often they are called. Any co-design tool operating at the source-code level will completely miss this performance improvement opportunity.

Architecture-Independent Compiler Optimizations

Most C/C++ or other source language compilers are very efficient at architecture-independent optimizations such as con-

stant folding, constant propagation, dead code elimination, and common sub-expression elimination. For example, consider this simple snippet of pseudo code:

```
int a = 30;
int b = 9 - a / 5;
int c;

c = b * 4;
if (c > 10) {
    c = c - 10;
}
return c * (60 / a);
```

Applying the compiler optimization techniques discussed earlier, this pseudo code can be reduced to:

```
return 4;
```

A co-design tool operating at the binary level can leverage these compiler optimizations to further improve the accuracy of hardware/software partitioning. Tools that work at the source level must either perform these optimizations internally or leave some area/performance on the table, resulting in a sub-optimal co-design.

Legacy Applications

For most popular processors such as PowerPC™, ARM, and TI DSP, a host of applications are already out in the field. These applications would benefit from a port to a faster FPGA platform with a co-design implementation. In many of these situations the source code is either not available or is written in an obsolete language. If the source code is available, the original developers may no longer be associated with the project or company; therefore the legacy knowledge does not exist. In such scenarios a tool that works from the binary can prove to be an invaluable asset.

Designing with BINACHIP-FPGA

The BINACHIP-FPGA tool is designed for embedded application developers and

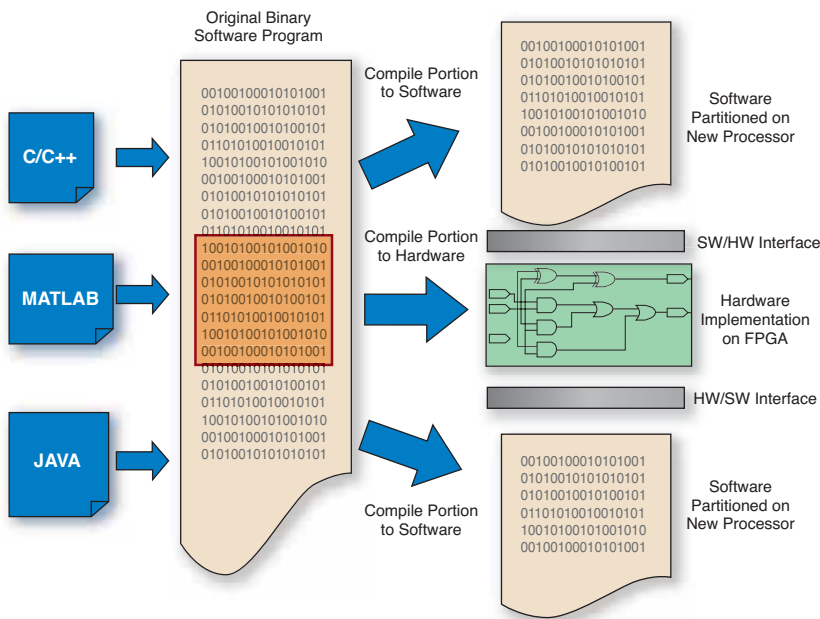


Figure 1 – Transforming a pure software implementation into a co-design

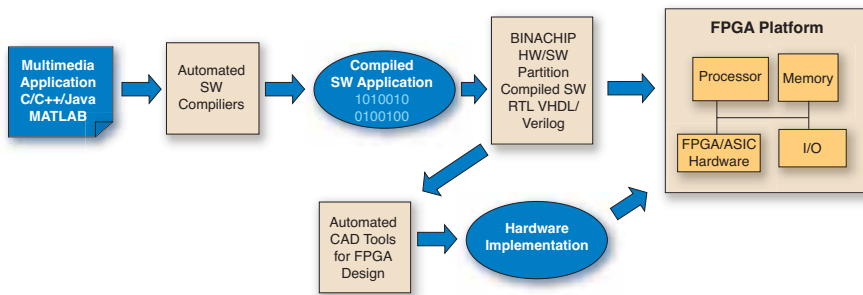


Figure 2 – Design flow using BINACHIP-FPGA

allows them to leverage all of the features of the FPGA platform, without having any intrinsic knowledge of hardware design or needing to learn a new source language. The transformation performed by the tool in going from pure software implementation to a co-design is conceptually illustrated in Figure 1.

An embedded application written in a combination of C/C++, MATLAB, and Java is compiled into a binary executable for a target platform. The bottleneck is identified based on profiling data or user input (see the orange box in Figure 1). The tool automatically generates hardware for these functions, which is mapped into the

programmable fabric of the FPGA. All hardware/software communication interfaces are also generated by the tool. Calls to these functions are now computed in hardware and control is returned to the software when the computation is complete. The tool then outputs a new executable that runs seamlessly on the target platform.

BINACHIP-FPGA is also capable of performing advanced optimizations such as loop unrolling, which allows you to make area/performance trade-offs. You can also exploit fine-grain parallelism by using advanced data scheduling techniques like as-soon-as-possible (ASAP) or as-late-as-possible (ALAP) scheduling.

Empowering Application Developers

BINACHIP-FPGA empowers embedded application developers to create advanced applications on FPGA platforms with minimal intervention or interaction with hardware designers. Figure 2 shows a typical design flow for the tool.

After the system architect has made a decision on which platform to use, the application developers start writing software and go through the usual task of compiling, debugging, and profiling the code. BINACHIP-FPGA fits into the flow once the application developers have debugged the basic functionality. After feeding the compiled application and profiler data into the tool, you have the option to manually guide the tool and select which functions should be mapped into the hardware. The tool generates the hardware and reconstructed binary with appropriate calls to the hardware.

You can then simulate the resulting implementation in a hardware/software co-design tool using the bit-true test benches generated by BINACHIP-FPGA. The RTL portion is mapped into the FPGA fabric using Xilinx Synthesis Technology (XST) and the ISE™ design environment. The application is now ready to run on the target platform.

Conclusion

Hardware/software co-design tools promise to empower embedded application developers and enable the large-scale deployment of platform FPGAs. To deliver on that promise, these tools must assume that users will have minimal or no hardware design knowledge. You should not be required to learn new language constructs to use the tools effectively.

BINACHIP-FPGA is the first tool in the market that uses a unique approach to enable application developers to fully leverage the FPGA platform and meet their price, performance, and time-to-market constraints.

For more information on hardware/software co-design or BINACHIP-FPGA, visit www.binachip.com.

The Benefits of FPGA Coprocessing

The Xilinx ESL Initiative brings the horsepower of an FPGA coprocessor closer to the reach of traditional DSP system designers.

A detailed, colorful image of an FPGA die, showing various functional blocks in shades of green, blue, purple, and yellow, arranged in a complex grid pattern. The die is shown from an isometric perspective, highlighting its intricate circuitry.

by Tom Hill
DSP Marketing Manager
Xilinx, Inc.
tom.hill@xilinx.com

High-performance DSP platforms, traditionally based on general-purpose DSP processors running algorithms developed in C, have been migrating towards the use of an FPGA pre-processor or coprocessor. Doing so can provide significant performance, power, and cost advantages (Figure 1).

Even with these considerable advantages, design teams accustomed to working on processor-based systems may avoid using FPGAs because they lack the hardware skills necessary to use one as a coprocessor. Unfamiliarity with traditional hardware design methodologies such as VHDL and Verilog limits or prevents the use of an FPGA, oftentimes resulting in more expensive and power-hungry designs. A new group of emerging design tools called ESL (electronic system level) promises to address this methodology issue, allowing processor-based developers to accelerate their designs with programmable logic while maintaining a common design methodology for hardware and software.

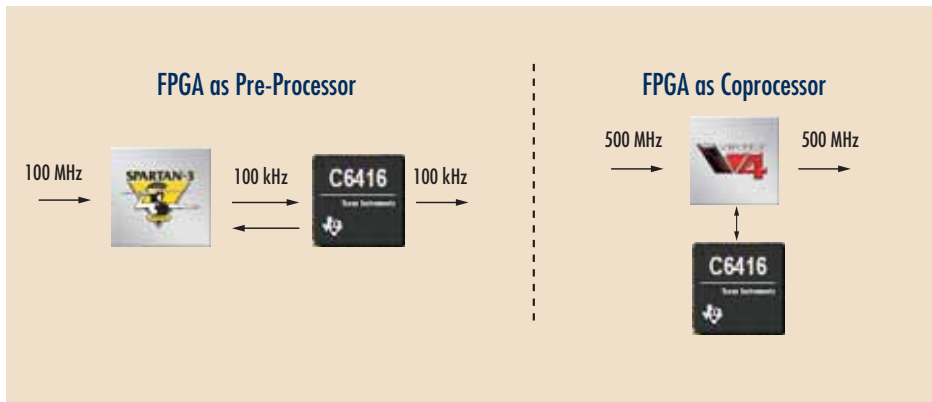


Figure 1 – DSP hardware platform

Boosting Performance with FPGA Coprocessing

You can realize significant improvements in the performance of a DSP system by taking advantage of the flexibility of the FPGA fabric for operations benefiting from parallelism. Common examples include (but are not limited to) FIR filtering, FFTs, digital down conversion, and forward error correction (FEC) blocks.

Xilinx® Virtex™-4 and Virtex-5 architectures provide as many as 512 parallel multipliers capable of running in excess of 500 MHz to provide a peak DSP performance of 256 GMACs. By offloading operations that require high-speed parallel processing onto the FPGA and leaving operations that require high-speed serial processing on the DSP, the performance and cost of the DSP system are optimized while lowering system power requirements.

Lowering Cost with FPGA Embedded Processing

A DSP hardware system that includes an FPGA coprocessor offers numerous implementation options for the operations contained within the C algorithm, such as partitioning the algorithm between the DSP processor, the FPGA-configurable logic blocks (CLBs), and the FPGA embedded processor. The Virtex-4 device offers two types of embedded processors: the MicroBlaze™ soft-core processor, often used for system control, and the higher performance PowerPC™ hard-core embedded processor. Parallel operations partitioned into the FPGA fabric can be

used directly in a DSP datapath or configured as a hardware accelerator to one of these embedded processors.

The challenge facing designers is how to partition DSP system operations into the available hardware resources in the most

C to Gates

When targeting an FPGA, the term “C to gates” refers to a C-synthesis design flow that creates one of two implementation options – direct implementation onto the FPGA fabric as a DSP module or the creation of a hardware accelerator for use with the MicroBlaze or PowerPC 405 embedded processor (Figure 2).

When an operation lies directly in the DSP datapath, the highest performance is achieved by implementing an operation as a DSP module. This involves synthesizing the C code directly into RTL and then instantiating the block into the DSP datapath. You can perform this instantiation using traditional HDL design methodologies or through system-level design tools such as Xilinx System Generator for DSP. Through direct instantiation, you can achieve the highest performance with minimal overhead.

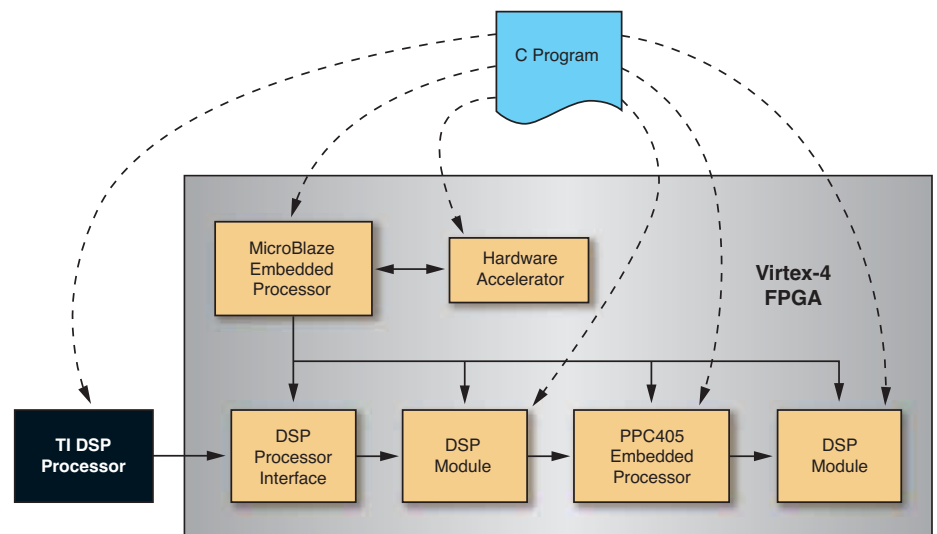


Figure 2 – C implementation options for a DSP hardware system

efficient and cost-effective manner. How best to use FPGA embedded processors is not always obvious, but this hardware resource can have the greatest impact on lowering overall system cost. FPGA embedded processors provide an opportunity to consolidate all non-critical operations into software running on the embedded processors, minimizing the total amount of hardware resources required for the system.

The leading C-synthesis tools are capable of delivering performance approaching that of hand-coded RTL – but achieving this requires detailed knowledge of C-synthesis tool operation and coding styles. Code modifications and the addition of inline synthesis instructions for inserting parallelism and pipeline stages are typically required to achieve the desired performance. Even with these modifications, how-

ever, the productivity gains can be significant. The C-system model remains the golden source driving the design flow.

An alternative and often simpler approach is to create a hardware accelerator for one of the Xilinx embedded processors. The processor remains the primary target for the C routines, with the exception that performance-critical operations are pushed to the FPGA logic in the form of a hardware accelerator. This provides a more software-centric design methodology. However, some performance is sacrificed with this

approach. C routines are synthesized to RTL, similar to the DSP module approach, except that the top-level entity is wrapped with interface logic to allow it to connect to one of the Xilinx embedded processor buses. This creates a hardware accelerator that can be imported into the Xilinx EDK environment and called through a software-friendly C function call.

The performance requirements for mapping C routines into hardware accelerators are typically less aggressive. Here the objective is to accelerate the performance beyond that of pure software while maintaining a software-friendly design flow. Although the coding techniques and in-line synthesis instructions are still available, you can typically achieve your desired performance gains without their use.

Design Methodology – The Barrier to Adoption
The effort and breadth of skill required to correctly partition and implement a complex DSP system is formidable. In 2005, the market research firm Forward Concepts conducted a survey to determine the most important FPGA selection criteria for DSP. The published results, shown in

Figure 3, identify development tools as the most important.

The survey illustrates that the benefits of a DSP hardware system utilizing an FPGA coprocessor are well understood, but that the current state of development tools represents a significant barrier to adoption for traditional DSP designers.

The Xilinx ESL Initiative

ESL design tools are pushing digital design abstraction beyond RTL. A subset of these tool vendors are specifically focused on mapping system models developed in C/C++ into DSP hardware systems that include FPGAs and DSP processors. Their vision is to make the hardware platform transparent to the software developer (Figure 4).

Rather than attempting to solve one piece of this methodology puzzle internally, this year Xilinx launched a partnership program with key providers of ESL tools called the ESL Initiative. The focus of this partnership is to empower designers with software programming skills to be able to easily implement their ideas in programmable hardware without having to learn traditional hardware design skills. This program is designed to accelerate the development and adoption of world-class design methodologies through innovation within the ESL community.

Conclusion

When combined, the collective offerings from Xilinx ESL partners offer a wide spectrum of complementary solutions that are optimized for a range of applications, platforms, and end users. Xilinx too has focused its efforts on complementary technology. For example, AccelDSP Synthesis provides a hardware path for algorithms developed in floating-point MATLAB, while System Generator for DSP allows modules developed using ESL designs to be easily combined with Xilinx IP and embedded processors. The quickest path to realizing a programmer-friendly FPGA design flow is through a motivated and innovative set of partners.

For more information about the ESL Initiative, visit www.xilinx.com/esl.

DSP CHIP SELECTION CRITERIA

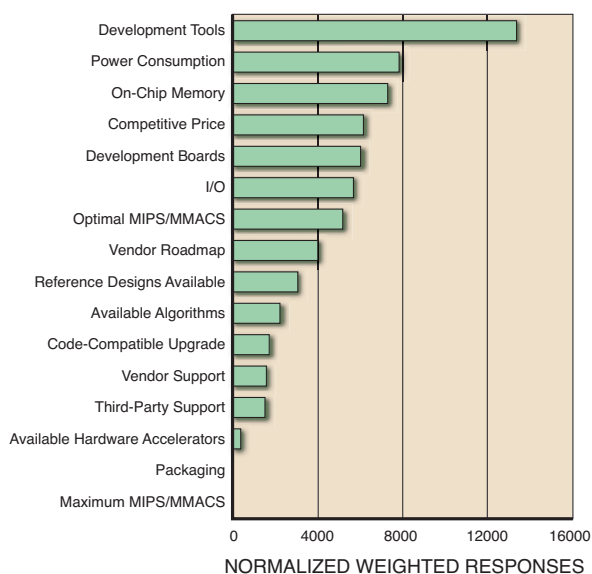


Figure 3 – 2005 Forward Concepts market survey, “DSP Strategies: The Embedded Trend Continues”

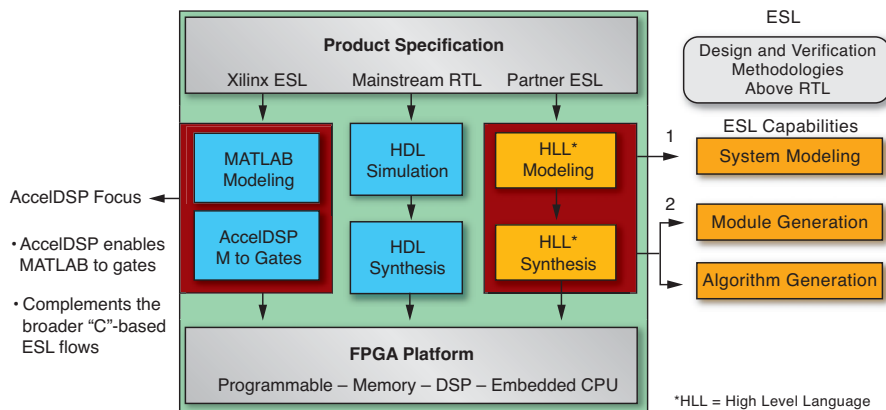


Figure 4 – Xilinx ESL Initiative design flows

Using SystemC and SystemCrafter to Implement Data Encryption

SystemCrafter provides a simple route for high-level FPGA design.

by Jonathan Saul, Ph.D.
CEO
SystemCrafter
jon.saul@systemcrafter.com

C and C++ have been a popular starting point for developing hardware and systems for many years; the languages are widely known, quick to write, and give an executable specification, which allows for very fast simulation. C or C++ versions of standard algorithms are widely available, so you can easily reuse legacy and publicly available code. For system-level design, C and C++ allow you to describe hardware and software descriptions in a single framework.

Two drawbacks exist, however. First, C and C++ do not support the description of some important hardware concepts, such as timing and concurrency. This has led to the development of proprietary C-like languages, which aren't popular because they tie users to a single software supplier. Second, you must manually translate C and C++ to a hardware description language such as VHDL or Verilog for hardware implementation. This time-consuming step requires hardware experts and often introduces errors that are difficult to find.

The first problem has been solved by the development of SystemC, which is now a widely accepted industry standard that adds hardware concepts to C++.

The second problem has been solved by the development of tools like SystemCrafter SC, which allows SystemC descriptions to be automatically translated to VHDL.

In this article, I'll explain how to use SystemCrafter SC and SystemC by describing an implementation of the popular DES encryption algorithm.

SystemC

SystemC provides an industry-standard means of modeling and verifying hardware and systems using standard software compilers. You can download all of the material required to simulate SystemC using a standard C++ compiler, such as Microsoft Visual C++ or GNU GCC, from the SystemC website free of charge (www.systemc.org).

SystemC comprises a set of class libraries for C++ that describe hardware constructs and concepts. This means that you can develop cycle-accurate models of hardware, software, and interfaces for simulation and debugging within your existing C++ development environment.

SystemC allows you to perform the initial design, debugging, and refinement using the same test benches, which eliminates translation errors and allows for fast, easy verification.

And because SystemC uses standard C++, the productivity benefits offered to software engineers for years are now

available to hardware and system designers. SystemC is more compact than VHDL or Verilog; as a result, it is faster to write and more maintainable and readable. It can be compiled quickly into an executable specification.

SystemCrafter SC

SystemC was originally developed as a system modeling and verification language, yet it still requires manual translation to a hardware description language to produce hardware.

SystemCrafter SC automates this process by quickly synthesizing SystemC to RTL VHDL. It will also generate a SystemC description of the synthesized circuit, allowing you to verify the synthesized code with your existing test harness.

You can use SystemCrafter SC for:

- Synthesizing SystemC to hardware
- System-level design and co-design
- Custom FPGA co-processing and hardware acceleration

SystemCrafter SC gives you control of the critical steps of scheduling (clock cycle allocation) and allocation (hardware reuse). Thus, the results are always predictable, controllable, and will match your expectations.

SystemCrafter SC allows you to develop, refine, debug, and synthesize hardware

and systems within your existing C++ compiler's development environment. You can run fast, executable SystemC specifications to verify your design. You can also configure the compiler so that SystemCrafter SC will automatically run when you want to generate hardware.

The DES Algorithm

The Data Encryption Standard (DES) algorithm encodes data using a 64-bit key. This same 64-bit key is required to decode the data at the receiving end. It is a well-proven, highly secure means of transmitting sensitive data. DES is particularly suitable for hardware implementation, as it requires only simple operations such as bit permutation (which is particularly expensive in software), exclusive-OR, and table look-up operations.

A DES implementation comprises two stages. During the first stage, 16 intermediate values are pre-computed based on the initial key. These 16 values are fixed for a particular key value and can be reused for many blocks of data. Calculation of the key values involves repeated shifts and reordering of bits.

The second computation stage involves 16 iterations of a circuit using one of the pre-computed key values. Encryption is based on 64-bit blocks of data with 64 bits of input data encoded for each group of 16 iterations, resulting in 64 bits of output data. Each iteration involves permutations, exclusive-OR operations, and the look-up of eight 4-bit values in 8 look-up tables.

Decryption works exactly the same way as the second computation stage, but with the 16 key values from the first stage used in reverse order.

For a full description of the DES algorithm, go to <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>.

Design Flow

The design flow using SystemC and SystemCrafter is shown in Figure 1. An important benefit of this design flow is that you can carry out the development of the initial SystemC description, partitioning, and system- and gate-level simulation all in the same framework. The designer

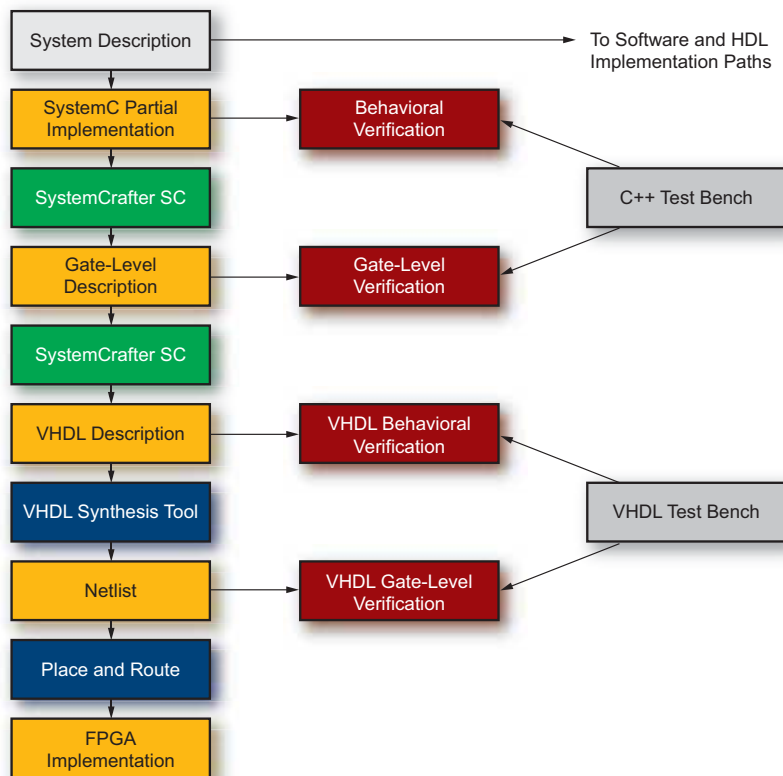


Figure 1 – Design flow

implementing the DES algorithm used his normal design environment, Microsoft's Visual C++. The target platform was the ZestSC1, an FPGA board containing a Xilinx® Spartan™-3 FPGA and some SRAM communicating with a host PC through a USB interface.

Hardware/Software Partitioning

The first step is to write an initial system-level description. It may be appropriate to decide on hardware/software partitioning at this point, or you can defer this decision until you have a working system-level description.

In the DES example, the hardware/software partitioning decision was easy and intuitive. The core of the algorithm involves high-bandwidth computation using operators unsuitable for software implementation, and thus will be implemented on the FPGA.

As I stated previously, the designer wrote a hardware description of the DES core in SystemC using Microsoft's Visual C++ design environment. Pseudo code is shown in Figure 2. Calls to SystemCrafter

are made as “custom build steps,” which allows simulation and synthesis from within the Visual C++ design environment. You can also use the SystemCrafter GUI to manage projects.

Implementing look-up tables as CoreGen modules shows how complex third-party IP blocks are usable in a SystemCrafter design. SystemCrafter treats SystemC modules with missing method definitions as black boxes, allowing you to use IP blocks written in other languages. For simulation, the designer wrote a model of the look-up tables in SystemC for use during simulation.

Simulation and Synthesis

The SystemCrafter flow allows you to simulate the design at a number of levels.

In the DES example, the designer wrote a test harness, which fed standard DES test patterns through the hardware engine and displayed the results on the console. He specified two configurations in Visual C++: system level and gate level.

The system-level configuration was first. It compiles the original SystemC design to

```

loop forever
  if KeySet = 1
    DataInBusy = 1
    K(0) = Permute(KeyIn)
    loop for i = 1 to 16
      K(i) = K(i-1) << shift amount
      K(i) = Permute(K(i-1))
    end loop
    DataInBusy = 0
  else if DataInWE = 1
    DataInBusy = 1
    D(0) = Permute(DataIn)
    loop for i = 1 to 16
      E = Permute(D(i-1))
      A = E xor K(i)
      S = LUT(A)
      P = Permute(S)
      D(i) = Concat(Range(D(i-1), 31, 0), P xor Range(D(i-1), 63, 32))
    end loop
    wait for DataOutBusy = 0
    DataOut = D(16)
    DataOutWE = 1 for one cycle
    DataOutBusy = 0
  end if
end loop

```

Figure 2 – Pseudo code for the DES implementation

produce a simulation model at the behavioral level. This model is an executable program that can produce a fast behavioral-level simulation of the DES engine.

Once the behavioral model produced the desired results, it was time for the gate-level configuration. It automatically calls SystemCrafter synthesis as part of the build process, which produces two descriptions of the synthesized circuit: a SystemC description and a VHDL description.

As part of the build process, the gate-level SystemC description is compiled into a gate-level simulation model, which can produce a fast gate-level simulation of the DES engine. This verifies that the synthesis process is correct.

The designer used Mentor Graphics's ModelSim to simulate the VHDL description.

Implementation

The VHDL produced by SystemCrafter is the core of the implementation.

Support modules supplied with the board helped with the interface between the PC and ZestSC1 FPGA board. A VHDL module simplified the connection to the USB bus. The designer wrote a small piece of VHDL to connect the 16-bit USB bus to the 64-bit DES input and output ports.

The complete DES project was then compiled using standard Xilinx tools (XST for the VHDL and CORE Generator™ software for the ROMs, followed by place and route) to generate an FPGA configuration file.

The device driver and C library contained in the ZestSC1 support package were integral in developing a simple GUI that configures the board with the FPGA configuration file. It then loads an image, sends it to ZestSC1 over the USB bus for encryption and then decryption, and displays the resulting images.

Figure 3 shows a sample output from the GUI.



Figure 3 – DES image encryption GUI

Discussion

The DES application was written by a VHDL expert who was new to SystemC. The complete design, including both hardware and software, went from concept to final version in only three days, including a number of explorations of different design implementations.

The SystemCrafter flow was flexible enough so that the designer could use a mixture of SystemC for the algorithmic part of the design, CORE Generator blocks for the look-up tables, and VHDL for low-level interfacing.

The flow allowed him to use his existing Visual C++ design environment for code development, simulation, and synthesis. The SystemC description was more concise than an equivalent VHDL design would have been, and this level of description allowed the development to focus on algorithmic issues.

Conclusion

SystemCrafter SC offers a simple route from system-level design down to Xilinx FPGAs.

It is suitable for programmers, scientists, systems engineers, and hardware engineers. It enables you to view developing hardware as a higher-level activity than writing HDL and allows you to focus on the algorithm rather than on the details of the implementation. This can improve time to market, reduce design risk, and allow you to design complex systems without learning HDLs or electronics.

Both SystemCrafter SC and the DES implementation, including working source files and a more detailed description, are available as downloads from the SystemCrafter website, www.systemcrafter.com.

Scalable Cluster-Based FPGA HPC System Solutions

Nallatech introduces an HPC family of FPGA system solutions optimized for clustering, with performance capability rising to multi-teraflops in a single 19-inch rack.



by Allan Cantle
President and Founder
Nallatech
a.cantle@nallatech.com

The high-performance computing (HPC) markets have taken a serious interest in the potential of FPGA computing to help solve computationally intensive problems while simultaneously saving space and reducing power consumption. However, to date few capable products and tools are tailored to suit HPC industry requirements.

In the last 15 years, the industry has also moved away from customized, massively parallel computing platforms (like those offered by Cray and SGI) to clusters of industry-standard servers. The majority of HPC solutions today are based on this clustered approach.

To serve the HPC market, Nallatech has introduced a family of scalable cluster-optimized FPGA HPC products, allowing you to either upgrade your existing HPC cluster environments or build new clusters with commercial-off-the-shelf FPGA computing technology.

Nallatech has developed a tool flow that also allows HPC users to continue to use a standard C-based development environment to target FPGA technology. Once familiar with this basic tool flow, you can then learn more about FPGA computing, leveraging more advanced features to fully exploit the technology.

A Scalable FPGA HPC Computing Platform

The traditional approach to HPC computing comprises a large array of microprocessors connected together in a symmetrical fashion using the highest bandwidth and lowest latency communications infrastructure possible. This approach provides a highly scalable structure that is relatively easy to manage.

Although easily scalable and manageable, these regular computing structures are rarely the best architecture to solve different algorithmic problems. Algorithms have to be tailored to fit the given computing platform.

With FPGAs, by contrast, you can build a custom computer around any specific algorithmic problem. This approach yields significant benefits in performance, power, and physical size and has been widely adopted in the embedded computing community. However, embedded computers are typically designed with one application in mind – and therefore a custom computer is crafted for each application.

Nallatech's FPGA HPC architectures are designed to merge the best of both worlds from the embedded and HPC computing industries. Our FPGA HPC architecture is symmetrical with regular communications between FPGA processors, but also has an inherent infrastructure that allows you to handcraft custom computing architectures around given algorithmic problems.

Recognizing the popularity of clustering, Nallatech selected and targeted the architecture for IBM BladeCenter computing platforms. With this approach, you can mix and match traditional microprocessor blades and FPGA computing blades in a proportion that suits the amount of code targeted at the FPGA.

Figure 1 shows the four basic types of configurations possible with Nallatech's

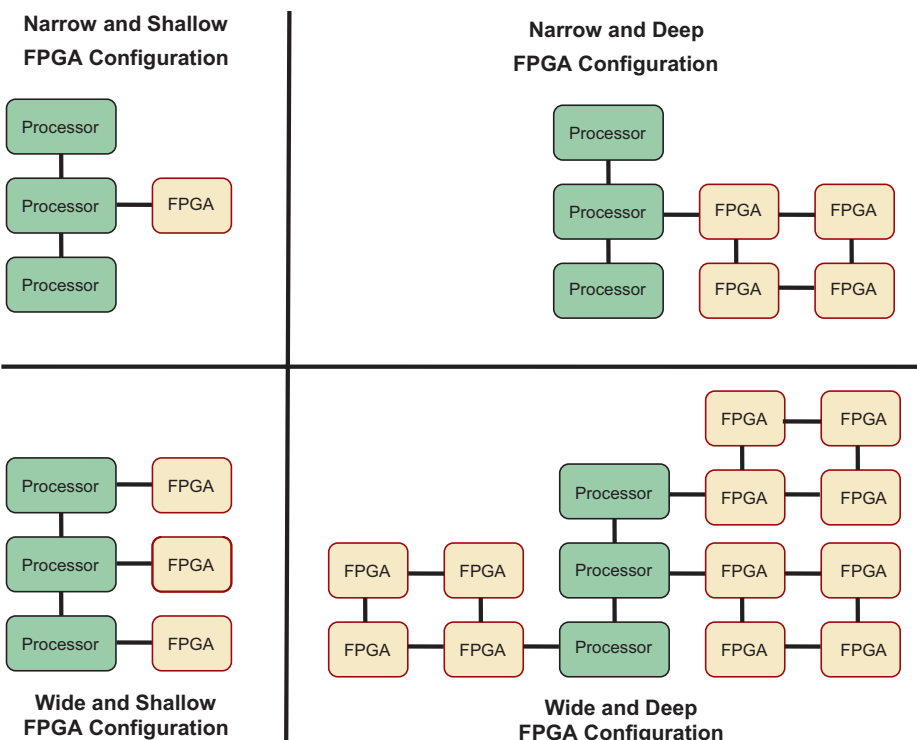


Figure 1 – Four types of processor-to-FPGA cluster configurations

FPGA HPC computing platforms. The wide scaling capability allows you to scale the FPGA computing resource in a way that is identical to standard cluster scaling. The deep and wide scaling capability of this architecture additionally permits scaling to more complex custom computing algorithms by treating multiple FPGAs as a single, large FPGA.

A 7U IBM BladeCenter chassis with a protruding Nallatech FPGA computing blade is shown in Figure 2. You can mix this FPGA blade with host processor blades in configurations similar to those detailed in Figure 1. Each blade is configurable with between one and seven FPGA processors providing the deep scaling. Several blades installed into one or more BladeCenter chassis provide the wide scaling necessary for more complex HPC applications.

Floating-point implementations of code in the fields of seismic processing and computational fluid dynamics have demonstrated performance improvements more than 10 times that of standard processors, while achieving power savings well in excess of 20 times.



Figure 2 – IBM BladeCenter chassis with protruding Nallatech FPGA blade

Creating Custom Virtual Processors

Nallatech's approach to FPGA HPC computing provides an extremely flexible architecture that enables you to create exceptionally high-performance computing solutions with a highly efficient use of power and space. However, without the appropriate development tools, using this type of platform would be virtually impossible. As mentioned previously, any physical instantiation of Nallatech's FPGA

computing platform can effectively be treated as either:

- A number of physical FPGA processors that exist in the computer connected together
- One massive FPGA processor combining multiple FPGAs and creating one custom compute engine with these processor chips
- Thousands of tiny processors with many processors existing in each FPGA

Whenever a new design commences, it is counterproductive to think of the problem in terms of the physical computing architecture at hand. Instead, consider how you can construct a custom computer to solve the specific compute problem. Effectively, you can develop a virtual processor that is explicitly designed to solve your specific algorithmic problem.

There are many ways to create a virtual processor for FPGAs, including:

- Using standard language compilers to FPGAs such as C and Fortran, pulling together various computing library functions available for the FPGA
- Using high-level algorithmic tools such as Simulink from The MathWorks's together with Xilinx® System Generator
- Designing a dedicated processor from low-level hardware languages such as VHDL

Most HPC software engineers prefer the language-based approach for obvious reasons, so language-based compilers are now maturing and becoming a viable approach to creating virtual processors.

As a high-performance FPGA computing systems provider, Nallatech has taken an approach to try to support as many compiler tools as possible. Today, you can develop virtual processors using three approaches:

- Nallatech's DIME-C enabling compiler
- Impulse Technologies's Impulse C compiler
- Xilinx System Generator for The MathWorks's Simulink environment

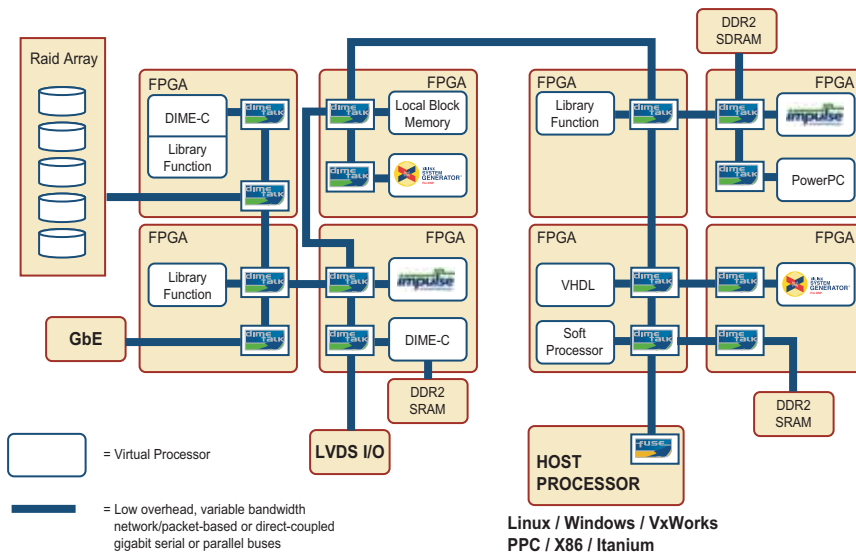


Figure 3 – Example DIMETalk network implementing many different virtual processors

Nallatech's DIME-C compiler is purely based on a subset of ANSI-C, with no requirement for pragmas or other specific actions. This means that you can design and simulate code with the tools with which you are familiar. Once the C code is functioning correctly, you have the freedom to compile directly through to a bitstream and run the code on a chosen FPGA processor. Nallatech tools provide all of the APIs necessary to access the FPGA through simple function calls from the host processor.

Maximizing Data Throughput and Integrating Virtual Processors

In many cases, creating a virtual FPGA processor using a language-based software environment and implementing the resulting executable on the allocated FPGA coprocessor can substantially improve the performance of the chosen application.

However, this approach does not necessarily yield the most optimum implementation. You can often benefit from splitting your algorithmic problem into several independent processes. In these circumstances it is often worth it to create a virtual processor for each process, as this will allow you to decouple one processor from the other and give you more architectural implementation flexibility.

DIMETalk allows you to effectively place many virtual processors across the FPGA computing resources in an HPC

platform while minimizing the communications overhead between processing elements. DIMETalk also allows you to effectively maximize the four levels of memory hierarchy that Nallatech's FPGA HPC platforms provide.

Figure 3 shows an example of a complex compute engine implemented across multiple FPGAs in the HPC platform. There are several virtual processors constructed from different compiler tools; these are connected together through the DIMETalk packet-switched communications network.

Conclusion

FPGA computing is deliverable on familiar cluster-based computing platforms; plus, with an ANSI-standard C compiler, you can achieve substantial performance gains. Looking forward, FPGA technology will likely advance HPC performance at a far greater rate than traditional processor-based technologies. For traditional software engineers, compiler tools will also become far more advanced and easy to use.

Nallatech is providing a special bundle of its entry-level HPC card with complete enabling tools software and a DIME-C to FPGA run time tool flow. Please visit www.nallatech.com/hpc for more details or to place an order.

For additional information, contact Nallatech at hpc@nallatech.com or contact@nallatech.com.

Accelerate Image Applications with Poseidon ESL Tools

Triton system-level design tools simplify the creation of high-speed solutions for imaging applications.



by Stephen Simon
Sr. Director of Marketing
Poseidon Design Systems
ssimon@poseidon-systems.com

Bill Salefski
VP of Technical Marketing
Poseidon Design Systems
bills@poseidon-systems.com

The number of embedded processor architectures implemented in today's image-processing systems is growing. Designers value the ability of processors to perform complex decision and combinational processes more efficiently than hardware logic. As a result, these designers are moving to a processor-based design that combines the advantages of processors with the inherently fast processing rates of hardware accelerators.

Image-processing applications are rich with signal-processing requirements. FFTs, FIR filtering, discrete cosine transforms (DCT), and edge detection are all common functions found in image-processing algorithms. These functions are computationally

intensive and can severely impact the performance of processor-based solutions. You can design these functions in hardware to elevate performance, but this requires a large amount of manpower and expertise, which may not be available in today's shrinking product design cycles.

Designers seem to be caught between the poor performance of processor-based solutions and the long design times of discrete hardware implementations. What they need is a way to bridge the gap and deliver a system solution that meets performance requirements without having to design discrete solutions with RTL.

Triton System-Level Tools

Poseidon Design Systems offers a way out of this dilemma. The Triton Tool Suite greatly simplifies the process of creating high-performance systems without the time-consuming task of designing hardware with RTL. The suite comprises two tools, Tuner and Builder. Tuner is a SystemC simulation, analysis, and verification environ-

ment; Builder is a hardware accelerator generation tool. Triton tools support PowerPC™ and Xilinx® MicroBlaze™ processor-based platforms.

Triton Tuner

Performing proper system analysis and design is critical when assembling any high-performance imaging system. For example, you must evaluate many architectural issues that can critically impact the performance of the entire system. Because these systems require complicated software and hardware architectures, you must also evaluate the performance of proposed architectures early in the design process rather than waiting for the working system to meet your desired performance goals.

A number of factors determine system performance beyond just the available processor MIPS. Some of these factors are proper cache management, shared-bus congestion, and system topology. With Poseidon's Tuner product (Figure 1), you can co-simulate your hardware and software

systems in the SystemC environment using transaction-level models (TLMs). You can then evaluate system operation and performance and make critical design decisions to develop a robust, efficient solution.

Tuner uses transaction-level modeling to provide the proper level of abstraction and fast simulation speeds. This environment also provides data visualization capability and event coherency, giving you a powerful tool in analyzing and verifying the performance of your target architecture. Tuner outputs data not just on processor performance, but also on cache performance, traffic congestion on shared buses, and efficiency of the memory hierarchy. Tuner links hardware events such as cache misses to the line of code and data accessed at the time; you can gain insight into the software profile and how effectively the system architecture supports the software.

Triton Builder

Hardware/software partitioning also impacts architectural decisions. Determining what portions of the algorithms need to be in hardware to achieve your desired results is not a simple task. In a traditional design flow, a long process follows the partitioning decision to build the hardware and test the resultant system. Changes at this point in the partition are very costly and will greatly impact the delivery schedule.

With Poseidon's Builder tool (Figure 2), you can quickly create a hardware/software partition and then evaluate the performance of the proposed architecture. Builder allows you to easily move critical portions of the selected algorithm to a custom hardware accelerator. Builder inputs algorithms written in ANSI C and creates a complete system solution. No special programming methodology or custom language is required. The tool generates the RTL of the accelerator, memory compiler scripts, test bench, modified C algo-

rithm, software drivers, and TLM of the resultant accelerator hardware. Builder generates all of the parts required to implement the accelerated solution.

To complete the design loop, you can import the accelerator TLM that Builder

created into Tuner and quickly evaluate the performance of the selected partition and resultant hardware. This provides you with almost instantaneous feedback about the effect of your partitioning decisions. The Tuner simulation environment displays system details that evaluation board platforms cannot. This tight loop enables you to quickly test different approaches and configurations, making key trade-offs in performance, power, and cost.

Builder supports different architectures to support different application data requirements. For large data requirements, Builder instantiates a bus-based solution (Figure 3) (such as the PowerPC PLB based on direct memory access (DMA) engines to rapidly move data into and out of the hardware. For smaller data sets or lower latency applications, Builder provides a tightly coupled solution that interfaces to the processor through the auxiliary processor unit (PowerPC) or Fast Simplex Link (MicroBlaze processor) interfaces. For very large data requirements, Builder supports the Xilinx multi-port memory controller (MPMC). This provides for large external fast memories to hold the data and overcome bandwidth limitations on existing processor buses.

With all of these options, Builder automates the process of programming the interfaces, communicating with the accelerator, and scheduling the data. With these two tools you can make critical design trade-offs and quickly realize a high-performance imaging system that meets your design requirements.

Radix 4 FFT Example

Let's look at a typical image-processing application using a fast Fourier transform. We chose an FFT for its popularity and because most designers are familiar with it. With Triton tools, you can not only quickly implement common functions such as FFTs; you can also implement the proprietary algorithms companies develop to differentiate their designs.



Figure 1 – The Tuner main screen displays software profiling data.

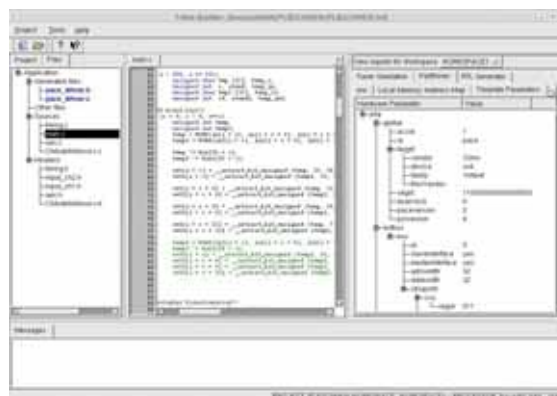


Figure 2 – The Builder main screen displays accelerator parameters.

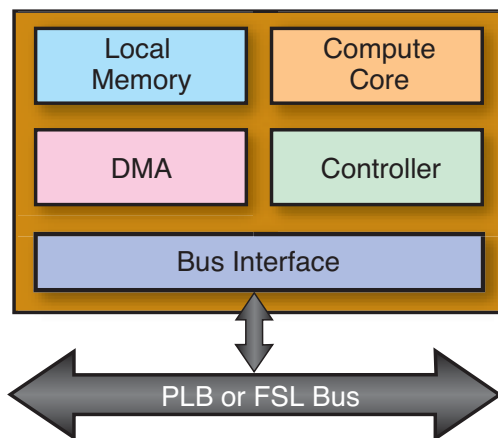


Figure 3 – Bus-based accelerator

Along with the ability to partition loops into hardware, Builder gives you the flexibility to easily move functions into hardware as well.

The specific FFT algorithm in our example is a Cooley-Tukey Radix 4 decimation in frequency written in standard ANSI C. The algorithm runs on a 300 MHz PowerPC on a Virtex™-II platform. The algorithm executes in 1.3 ms. This gives a maximum frame rate of 769 frames per second when 100% of the host processor is dedicated to the FFT task. Processing the selected algorithm in place makes efficient use of the memory resources. The software comprises five main processing for loops, one for each pass of the FFT algorithm.

The first task is to profile the FFT algorithm to determine the best candidates for moving to FPGA fabric. Tuner reads the architectural description for the Xilinx Platform Studio (XPS) and simulates the algorithm. The five loops performing the Radix 4 butterflies consume between 15% and 23% of the cycles used in the entire algorithm. These are all good candidates for acceleration and C-to-RTL conversion into dedicated FPGA logic.

Using Builder, you can determine how much of the algorithm to move to hardware to accelerate the application. If the design needs a 50% improvement in processing speed, it would be very inefficient to implement an entire FFT in hardware. In this example, by selecting all five loops for partitioning, more than 99% of the processor cycles were moved into the hardware accelerator. It would be just as easy if you required less performance to select any number of loops to match the desired acceleration of the system, thus saving space in the fabric for other uses.

Along with the ability to partition loops into hardware, Builder gives you the flexibility to easily move functions into hardware as well. With this flexibility, you have the freedom to match the partition to the requirements and structure of the specific application.

Builder also provides a C-lint tool that can help you determine which existing code is synthesizable. If any non-synthesizable structures exist within the algorithm, the

tool flags that section of code and reports the condition that makes it unrealizable into hardware. The tool also suggests enhancements and modifications that might be useful in implementing more efficient algorithms in hardware.

The partitioning process is as simple as selecting the loops or function that you would like moved into the accelerator – the tool performs the rest. If you wish, you can control the optimizations and settings to optimize the solution, making trade-offs to best match the desired system.

Builder also takes care of integrating the partitioned code back into software by generating a software driver. The tool determines the scheduling of the variables for the resultant hardware and ensures that the data will be available when the hardware requires it. Builder determines memory sizes and memory banking requirements to efficiently process the data in the central compute core. Builder then creates the memory compilation script for the LogiCORE™ system to build the optimized memory structure.

The bus-based accelerator is appropriate in this example because of the moderate amount of data required. This solution has its own DMA, which enables the accelerator to access the data in system memory and run autonomously to the processor. Builder automatically schedules and programs the accelerator to transfer data into it using the DMA, and similarly transfer the results back out into system memory.

Because of the tight integration with the Tuner environment, you do not have to guess the effect of the partitioning decisions and architecture options created using Builder. With each new option you can quickly generate the TLM and simulate the proposed solution using Tuner. This gives you instant feedback as to how fast the resultant solution is as well as how it integrates into the existing system architecture. When the software and hardware meet the desired performance, you can automatically generate the pcores for the hardware accel-


erator to import back into XPS. All of the required files and scripts are generated, allowing you to continue on with the existing design flow.

By using this automated technology, you can easily create and verify speedups of as much as 16x. This pushes the maximum frame rate of the FFT application to 12,000 frames per second for each channel of FFT implemented, with the processor load decreasing to less than 5%.

You can see with our FFT example the ease of use of Triton ESL tools and how with little effort you can generate high-performance systems. This example took only two man weeks of effort and without any manual modifications to the resultant system. If you require additional processing capability, it is a straightforward process to modify the software to support a streaming architecture. Creating concurrency between passes or creating multiple channels with separate accelerators, you can quickly achieve an overall acceleration of 48x to 80x.

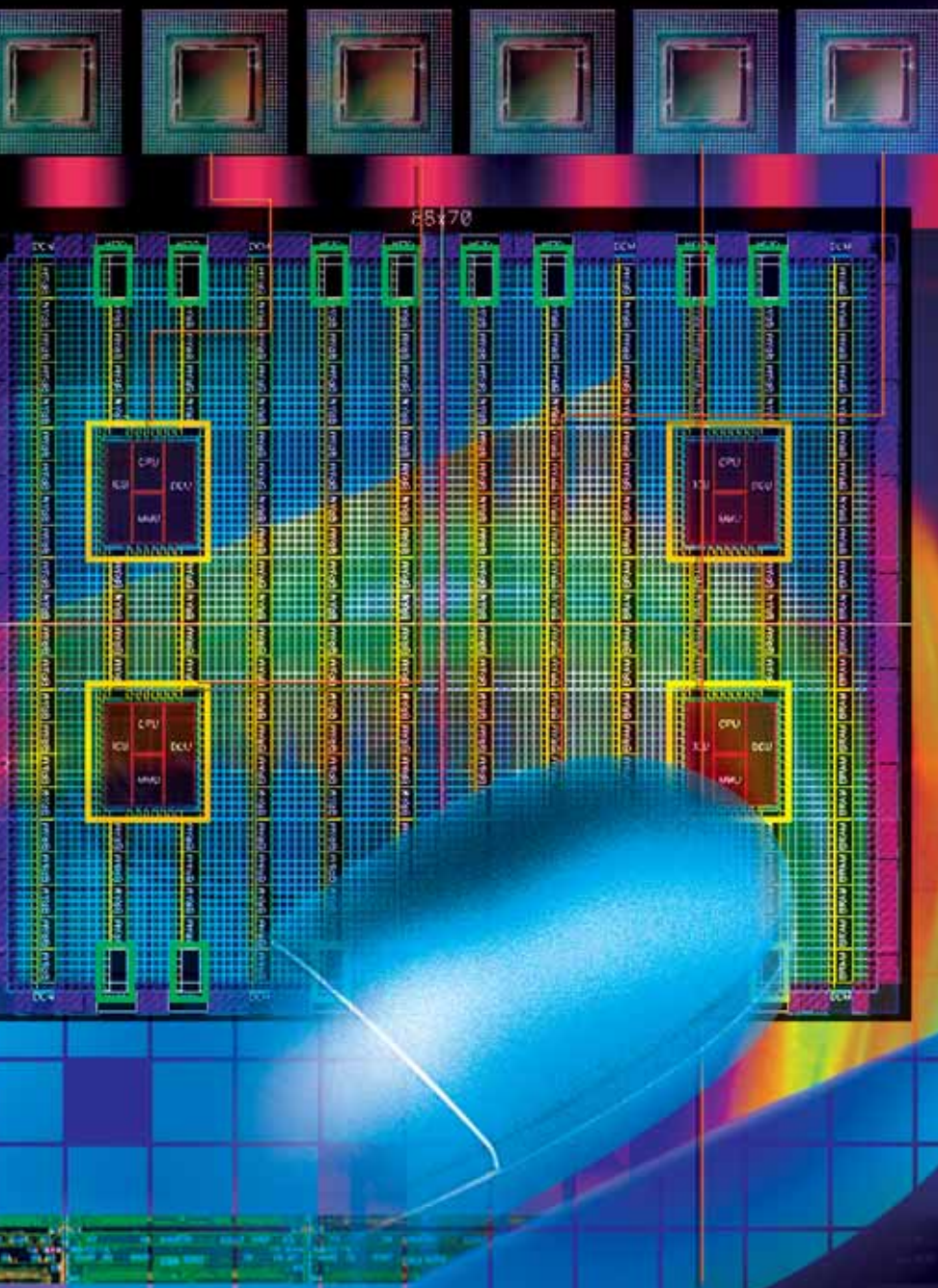
Conclusion

The Triton Tool Suite is a powerful tool for the development of image-processing architectures. The Triton Tool Suite enables you to efficiently perform and verify architectural development, hardware/software partitioning, and the automated hardware generation of key digital signal processing algorithms. Tuner and Builder not only provide large savings in your design efforts; they allow you to create more efficient high-performance and scalable solutions. The tools are highly automated, yet give you the flexibility to optimize your solution and meet your system design requirements.

The system also automates the interface between the Triton Tool Suite and the Xilinx XPS system. This solution is optimized for computationally intensive algorithms found in audio, video, VoIP, imaging, wireless, and security applications. For more information, please visit our website at www.poseidon-systems.com and request a free 30-day evaluation. 

Accelerating System Performance Using ESL Design Tools and FPGAs

ESL design tools provide the keys for opening more applications to algorithm acceleration.



by James Hrica
Systems Architect
Celoxica Inc.
james.hrica@celoxica.com

Jeff Jussel
GM Americas
Celoxica Inc.
jeff.jussel@celoxica.com

Chris Sullivan
Strategic Marketing Director
Celoxica Inc.
chris.sullivan@celoxica.com

The microprocessor has had more impact on electronics and our society than any other invention since the integrated circuit. The CPU's programmability using simple sets of instructions makes the capabilities of silicon available to a broad range of applications. Thanks to the scalability of silicon technologies, microprocessors have continuously grown in performance for the last 30 years.

But what if your processor does not have enough performance to power your application? Adding more CPUs may not fit power or cost budgets – and more than likely won't provide the needed acceleration anyway. Adding custom hardware accelerators as co-processors adds performance at lower power. But unlike processors, custom hardware is not easily programmable. Hardware design requires special expertise, months of design time, and costly NRE development charges.

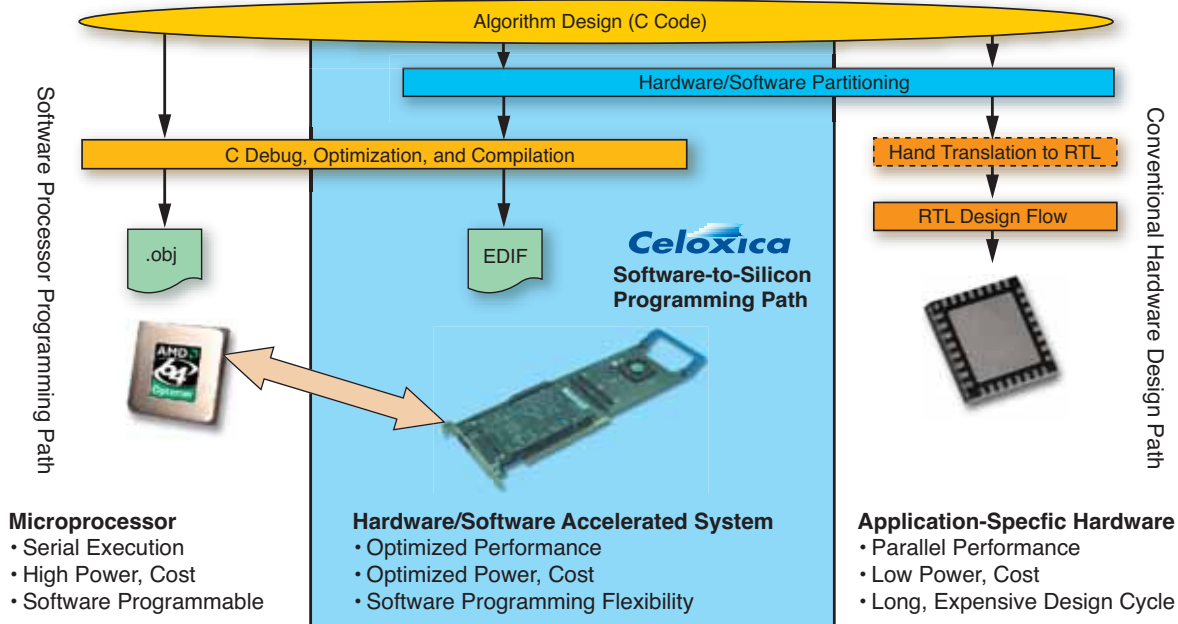


Figure 1 – Software design flows directly program FPGAs.

Enter the combination of high-performance Xilinx® FPGAs and electronic system-level (ESL) design tools, expressly tuned for FPGA design. By taking advantage of the parallelism of hardware, certain algorithms can gain faster performance – at lower clock speeds using less power – than what is possible in serial processor operations. Xilinx FPGAs provide a reconfigurable option for hardware acceleration, while the addition of ESL design flows from Celoxica gives developers access to these devices using a software design flow. Software-based design flows are now able to directly program Xilinx FPGA devices as custom co-processors (see Figure 1).

from parallelism. Specifically applied to systems using multiple CPUs, Amdahl's law states that the acceleration possible is limited to the proportion of the accelerating algorithm's total processing time. If the algorithm takes up p% of the total processing time and is spread across N processors, then the potential speedup can be written as $1/[(1-p) + p/N]$. For example, assuming that 20% of an algorithm is spread across four processors, the total processing time is at best reduced to 85% of the original. Given the additional cost and power required by three extra processors, a mere 1.2x speedup may not provide a big enough return.

Using custom hardware accelerators, the N in Amdahl's equation becomes the speedup multiplier for accelerated algorithms. In hardware, this multiplier can be orders of magnitude higher; the power utilization and costs of FPGA accelerators are generally much lower. Still, Amdahl's law would still seem to limit the potential theoretical total acceleration times. Fortunately, in practice Amdahl's law has proven too pessimistic; some suggest that the proportional throughput of parallel hardware increases with additional processing power, making the serial portion smaller in comparison. For certain systems, total accelerations of one, two, or even three orders of magnitude are possible.

Will FPGA Acceleration Work for All Algorithms?

When a general-purpose CPU or DSP alone can't deliver the performance, several options are available (Figure 2). However, not all algorithms lend themselves well to all types of acceleration. The first step in the process is to determine whether the application will benefit from the parallelism of hardware.

Amdahl's law, developed by computer architect Gene Amdahl in 1967, asserts a pessimistic view of the acceleration possible

Processor/DSP	Xilinx FPGA	Full Custom Hardware
Sequential Execution	Parallel Execution	Parallel Execution
Very Flexible	Flexible	Inflexible
Power-Inefficient	Power-Efficient	Very Power-Efficient
Inexpensive	Relatively Inexpensive (Compared to Custom ASIC)	Very Expensive (Very High Volumes Only)
* C-Based Design for Software and Hardware		
Configurable μ P/FPGA/Programmable SOC/DSP		

Figure 2 – System acceleration possibilities

What Portions of the Algorithm Should Go in the FPGA?

To accelerate a software application using FPGAs, you must partition the design between those tasks that will remain on the CPU and those that will run in hardware. The first step in this process is to profile the performance of the algorithm running entirely on the CPU.

You can gain some insight into where the application spends most of its run time using standard software profiling tools. The profiler report points to the functions, or even lines of code, that are performance bottlenecks. Free profiling tools such as gprof for the GNU development environment or Compuware's DevPartner Profiler for Visual Studio are readily available.

You should analyze the application code profile around areas of congestion to determine opportunities for acceleration. You should also look for sub-processes that can benefit from the parallelism of hardware. To accelerate the system, you move portions of your design into hardware connected to the CPU, such as an FPGA situated on a PCI, HyperTransport, or PCI Express interface board. PowerPC™ or MicroBlaze™ embedded processors provide another possible approach, putting the reconfigurable fabric and processor within the same device.

It is important during the analysis process not to overlook the issues of data transfer and memory bandwidth. Without careful consideration, data transfer overhead can eat into gains made in processing time. Often you may need to expand the scope of the custom hardware implementation to minimize the amount of data to be transferred to or from the FPGA. Processes that expand data going into the accelerated block (such as padding or windowing) or processes that contract data coming out of that block (such as down-sampling) should all be included in the FPGA design to reduce communication overhead.

Where Is Parallelism in the Algorithm?

Hardware accelerates systems by turning large serial software functions and performing them in parallel over many smaller processes. This results in a highly optimized, very specific set of hardware func-

tions. ESL tools are an important part of this process, allowing you to insert parallelism from software descriptions, using FPGAs like custom co-processors.

You can exploit parallelism at many levels to move functionality from software into custom hardware. Fine-grained parallelism is achieved by executing many independent, simple statements simultaneously (variable assignments, incrementing counters, basic arithmetic, and others). The pipelining of sequentially dependent statements is another form of fine-grain parallelism, accomplished by arranging individual stages in parallel such that each stage feeds the input of the next stage downstream.

For example, by pipelining the inside of a code loop, including the loop control logic, it may be possible to reduce iterations to only one clock cycle. After priming the pipeline with data, the loop could be made to execute in the number of clock cycles equal to the number of loop iterations (plus the relatively modest latency). This type of optimization often provides the most benefits when accelerating an algorithm in hardware.

Coarse-grained parallelism can also provide acceleration. Larger independent sub-processes like functions or large looped blocks, which run sequentially on the processor, can run simultaneously in custom hardware. Similarly, unrolling loops either fully or partially can add parallelism. In unrolling, the processing logic inside loops is replicated multiple times so that each instance works in parallel on different data to reduce the number of loop iterations. Pipelining large sequentially dependent processes can further remove cycles at the cost of some latency.

Another important form of coarse-grained parallelism occurs when the software and hardware operate in tandem. For example, a system accelerated by transferring frames of data to custom hardware for co-processing might be structured so that while the hardware processes one frame, the software is accomplishing any requisite post-processing of the previous frame, or pre-processing the next frame. This type of coordination and efficiency is made possible by the common development environ-

ments for both the hardware and software provided by ESL design.

Parallelism can also optimize the memory bandwidth in a custom hardware accelerator. For example, the addition operator for two vectors may be written as a loop, which reads an element of each vector from memory, adds the operands, and writes the resulting sum vector to memory. Iterations of this operation require three accesses to memory, with each access taking at least one cycle to accomplish.

Fortunately, modern Xilinx FPGAs incorporate embedded memory features that can help. By storing each of the three vectors in a separate embedded memory structure, you can fully pipeline the action of the summing loop so that iterations take only one cycle. This simple optimization is one example of how you can maximize system acceleration in hardware.

How Do Accelerators Handle Floating Point?

The use of floating-point mathematics is often the most important issue to resolve when creating custom hardware to accelerate a software application. Many software applications make liberal use of the high-performance floating-point capabilities of modern CPUs, whether the core algorithms require it or not. Although floating-point arithmetic operations can be implemented in PLD hardware, they tend to require a lot of resources. Generally, when facing floating-point acceleration, it is best to either leave those operations in the CPU portion of the design or change those operations to fixed point. Fortunately, you can implement many algorithms effectively using fixed-point mathematics, and there are pre-built floating-point modules for those algorithms that must implement floating point in hardware.

A detailed description of the process for converting floating-point to fixed-point operations can be highly algorithm-dependent, but in summary, the process begins by analyzing the dynamic range of the data going into the algorithm and determining the minimum bit width possible to express that range in an integer form. Given the width of the input data, you can trace through the operations involved to determine the bit growth of the data.

For example, to sum the squares of two 8-bit numbers, the minimum width required to express the result without loss of information is 17 bits (the square of each input requires 16 bits and the sum contributes 1 more bit). By knowing the desired precision of the output, simply work backwards through the operation of the algorithm to deduce the internal bit widths.

You can implement well-designed fixed-point algorithms in FPGAs quite efficiently because you can tailor the width of internal data paths. Once you know the width of the inputs, internal data paths, and outputs, the conversion of data to fixed point is straightforward, leading to efficient implementation on both the hardware and software sides.

Occasionally, an application requires more complex mathematical functions – $\sin()$, $\cos()$, \sqrt{x} , and others – on the hardware side of the partition. For example, these functions may operate on a discrete set of operands or may be invoked inside of loops, with an argument dependent on the loop index. In these cases, the function can usually be implemented in a modestly sized lookup table that can often be placed in embedded memories. Functions that take arbitrary inputs can also be used in hardware lookup tables with interpolation. If you need more precision, you can try iterative or convergent techniques at the cost of more cycles. In these cases, the software compilation tools should make good use of existing FPGA resources.

What Skills Are Required to Use FPGA Acceleration?

Fortunately, hardware design from ESL tools has come a long way in the last few years. It is now possible to use C-based hardware languages to design the hardware portions of the system. You can easily generate FPGA hardware from the original software algorithms.

Design tools from Celoxica can compile or synthesize C descriptions directly into FPGA devices by generating an EDIF netlist or RTL description. These tools commonly provide software APIs to abstract away the details of the hardware/CPU connection from the application development. This allows you to truly treat the FPGA as a co-

processor in the system and opens doors to the use of FPGAs in many new areas such as high-performance computing, financial analysis, and life sciences.

The languages used to compile software descriptions to the FPGA are specifically designed to both simplify the system design process with a CPU and add the necessary elements to generate quality hardware (see Figure 3). Languages such as Handel-C and SystemC provide a common code base for developing both the hardware and software portions of the system. Handel-C is based on ANSI-C, while SystemC is a class library of C++.

Any process that compiles custom hardware from software descriptions has to deal with the following issues in either the language or the tool: concurrency, data types, timing, communication, and resource usage. Software is written sequentially, but efficient hardware must translate that code into parallel constructs (using potentially multiple hardware clocks) and implement that code using all of the proper hardware resources.

Both Handel-C and SystemC add simple constructs in the language to allow expert users control over this process, while maintaining a high level of abstraction for representing algorithmic designs. Software developers who understand the concepts of parallelism inherent to hardware will find these languages very familiar and can begin designing accelerated systems using the corresponding tools in the matter of weeks.

How Much Acceleration Can I Achieve?

Table 1 gives a few examples of the acceleration achieved in Celoxica customer projects. In another example, a CT (computed tomography) reconstruction-by-filtered-back-projection application takes in sets of 512 samples (a tomographic projection) and filters that data first through an FFT. It then applies the filter function to the frequency sample and applies an inverse FFT. These filtered samples are used to compute their contribution to each of the pixels in the 512 x 512 reconstructed image (this is the back-projection process). The process is repeated with the pixel values accumulated from each of the 360 projections. The fully reconstructed image is then displayed on a computer screen and stored to disk. Running on a Pentium 4 3.0 GHz computer, the reconstruction takes about 3.6 seconds. For the purposes of this project, the desired accelerated processing time target for reconstruction is 100 ms.

Profiling the application showed that 93% of the CPU run time is spent in the function that does the back projection, making that function the main target for acceleration. Analyzing the back-projection code showed the bottleneck clearly. For every pixel location in the final image, two filtered samples are multiplied by coefficients, summed, and accumulated in the pixel value. This function is invoked 360 times, and so the inner loop executes around 73 million times, each time requir-

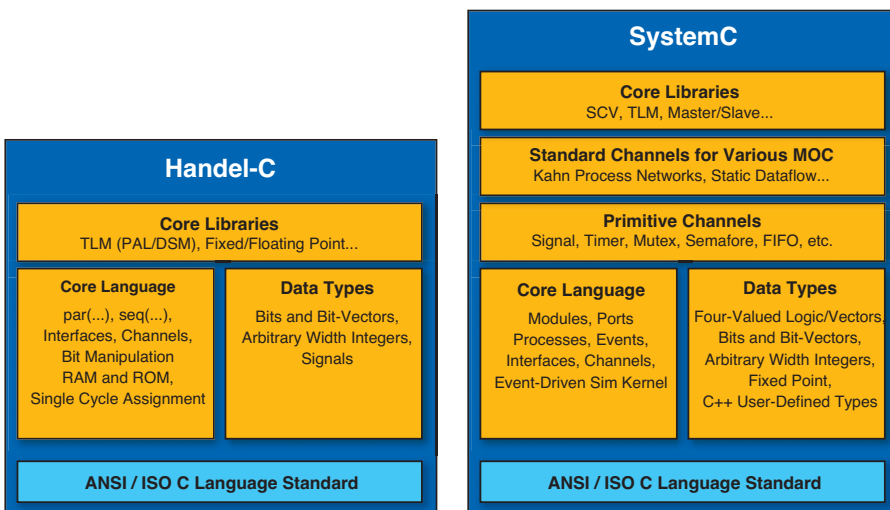


Figure 3 – Adding hardware capabilities

ing 3 reads and 1 write to memory, 3 multiplies, and several additions.

Fortunately, the back-projection algorithm lends itself well to parallel processing. Although the code was written using floating point, the integer nature of the input and output data made a fixed-point implementation perfectly reasonable. Working together, engineers from Celoxica and their customer accelerated the application, running on a standard PC, using a PCI add-in card with a Xilinx Virtex™-II FPGA and six banks of 512,000 x 32 SRAM. A sensible first-order partitioning of the application put the back-projection execution in the PLD, including the accumulation of the reconstructed image pixels, leaving the filtering and all other tasks running on the host processor.

The engineers designed a single instance of the fixed-point back-projection algorithm and validated it using the DK Design Suite. An existing API simplified the data transfer directly from the software application to the Xilinx FPGA over the PCI bus. They exploited fine-grained parallelism and rearranged some of the arithmetic to maximize the performance of the algorithm in hardware. The core unit was capable of pro-

cessing a single projection in about 3 ms running at a 66 MHz clock rate.

The fact that each projection could be computed independently meant that multiple back-projection cores could be running concurrently. The engineers determined that 18 cores would be enough parallelism to reach the performance goal comfortably. The software would pass 18 sets of 512 samples to the FPGA, which would store them in 18 individual dual-port RAMs. Both ports of these RAMs were interfaced to each back-projection core, allowing them to read the two samples required in a single cycle.


The cores were synchronized and the 18 contributions were fed into a pipelined adder tree. The final summed value was accumulated with the previous pixel value stored in one of the external SRAM banks. The accumulated value is written into the other SRAM bank and the two are read in ping-pong fashion on subsequent passes. With this architecture sample sets are transferred from software to hardware and the reconstructed image is passed back once when complete. The software was modified so that while one set of data was being processed in the FPGA, the next set was

being filtered by the host processor, further minimizing design time.

In the end, leveraging the features of FPGAs and modern ESL tools and taking advantage of the flexibilities provided by a software-based design flow, the example design was able to realize a 45x acceleration for a real-world application.

Conclusion

Accelerating complex software algorithms in programmable logic and integrating them into a system that contains processors or DSPs has never been easier or more affordable. Advanced FPGA architectures, high-speed interconnect such as PCI Express and HyperTransport, and the maturity of C-based ESL design tools reduce the design burden on developers and open up new possibilities for the application of FPGA technology. Balancing the needs of power, performance, design time, and cost is driving the FPGA into new market applications.

For more information about ESL design solutions for Xilinx FPGAs, low-cost starter kits, and FPGA-based acceleration benchmarks, visit www.celoxica.com/xilinx. 

Application	Hardware Co-Processor	Software Only
Hough and Inverse Hough Processing	2 sec of Processing Time @ 20 MHz 370x Faster	12 Minutes Processing Time Pentium 4 - 3 GHz
AES 1 MB Data Processing/Cryptography Rate Encryption Decryption	424 ms / 19.7 MBps 424 ms / 19.7 MBps 13x Faster	5,558 ms / 1.51 MBps 5,562 ms / 1.51 MBps
Smith-Waterman search34 from FASTA	100 sec FPGA processing 64x Faster	6461 sec Processing Time Opteron
Multi-Dimensional Hypercube Search	1.06 sec FPGA @ 140 MHz Virtex-II Device 113x Faster	119.5 sec Opteron - 2.2 GHz
Callable Monte-Carlo Analysis 64,000 Paths	10 sec of Processing @ 200 MHz FPGA System 10x Faster	100 sec Processing Time Opteron - 2.4 GHz
BJM Financial Analysis 5 Million Paths	242 sec of Processing @ 61 MHz FPGA System 26x Faster	6300 sec Processing Time Pentium 4 - 1.5 GHz
Mersenne Twister Random Number Generation	319M 32-bit Integers/sec 3x Faster (Bus Bandwidth BW Limited - Processing ~ 10-20x Ratio)	101M 32-bit Integers/sec Opteron - 2.2 GHz

Table 1 – Augmenting processor performance

Transforming Software to Silicon

C-based solutions are the future of EDA.

by Steve Kruse
Corporate Application Engineer
CebaTech, Inc.
kruse@cebatech.com

Sherry Hess
Vice President, Business Development
CebaTech, Inc.
hess@cebatech.com

Rumors of the possibility of translating the results of a high-level language such as C directly into RTL have abounded for the last few years. CebaTech, a C-based technology company, has turned rumors into reality with the introduction of an untimed C-to-RTL compiler, the central element in an advanced system-level design flow.

Operating in stealth mode for several years, the company developed its C-to-RTL compiler for use with IP development in TCP/IP networking. The nature of TCP/IP networking code, combined with CebaTech's desire to have a complete system-level design tool, dictated that the compiler handle both very large and very complex code bases. The CebaTech flow is a software-centric approach to FPGA and ASIC design and implementation, allowing hardware designers to describe system architectures and software engineers to implement and verify these architectures in a pure software environment with traditional software tools.

The challenge facing a team of system architects is to make the best design trade-off decisions so that the marketing requirements document (MRD) has been converted to the detailed specification for a balanced product. Only a balanced product is likely to find market acceptance, where careful consideration has been given to both innovative features and power-efficient usage of silicon. Until recently, a lack of suitable tools has hampered this all-important early decision-making.

The CebaTech flow allows architectural design at a high level of abstraction, namely untimed C. From this perspective, you can make design trade-offs in speed, area, and power of the end product up-front in C, where design issues are most easily addressed. The resulting “optimized” C-based design has a much higher likelihood of being competitive in the marketplace, thanks to the energy expended in honing it to an edge above and beyond the MRD.

A product with the correct cost/performance balance is what EDA strives to achieve. The eventual market price of a product is dictated by factors in the features and efficiency dimensions.

A Software-Oriented ESL Approach

Aside from striking the right balance between features and silicon efficiency, engineering teams must also grapple with verification of the chip design. Verification is by far the biggest component of product development costs because verification productivity has not grown at the same rate as chip capacity and performance. This issue has been acute for many years: the feasibility of integrating new features has not been matched by the designer’s/architect’s ability to guarantee correct implementation.

As represented by Figure 1, CebaTech aims to harness the convergence of three technological trends. First is the growing momentum of the open-source movement, wherein system-level specifications result from the collective intelligence of the global development community. Secondly, while these “golden” standards are freely available, successful implementation in silicon depends on domain-specific design

expertise. Finally, the cost of product development must be kept in check through increasing use of the latest system-level tools. Although the toolflow pioneered by

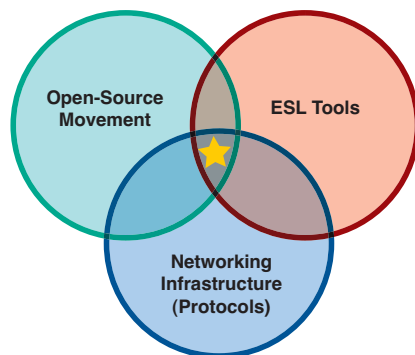


Figure 1 – The evolution and eventual convergence into CebaTech’s technology and methodology of C-based products that take “software to silicon” efficiently.

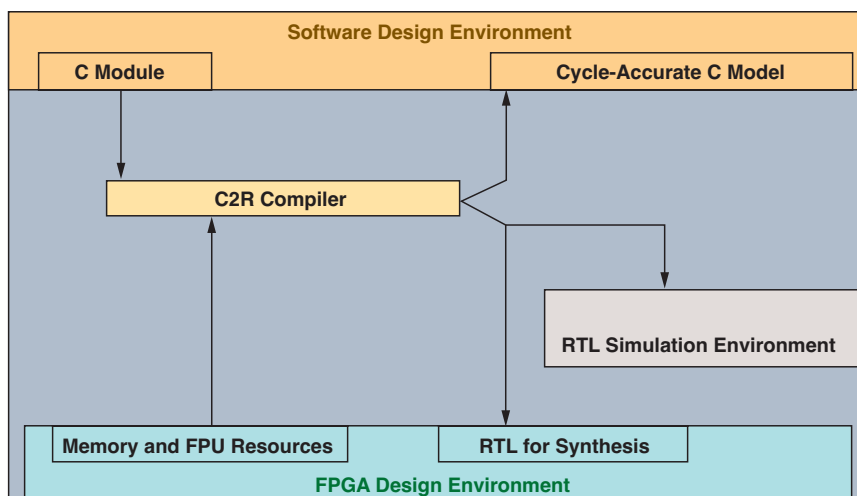


Figure 2 – The C2R compiler from CebaTech allows untimed ANSI C code bases to be rapidly implemented using an FPGA approach.

CebaTech has already proven its merit in IP networking, it is applicable to other emerging market sectors

CebaTech’s approach is a logical step in the evolution of the EDA industry, especially given the current bottleneck in verification. For the most part, complex system-on-a-chip (SoC) products are still developed in the traditional bottom-up way. Developers learned to use hardware description languages and write test benches, and eventually came to rely on FPGA-based prototyping systems and development environments for embedded cores. In recent years, traditional simulation

methodology has yielded ground to virtual system prototyping (VSP) platforms, assertion-based verification, and equivalence-checking methods. Nonetheless, the cost of design continues to grow.

CebaTech bypasses a bottom-up approach by using a direct top-down compilation from the original C source code base. The pervasiveness of C, especially for systems software, was a major consideration. C’s efficiency and intuitiveness for capturing hardware design benchmarks has achieved an overall improvement of at least 10x in manpower requirements as well as at least a 5x order-of-magnitude improvement in real-time verification.

The real benefit of compiling from the original C source code becomes apparent during the verification phase. CebaTech’s

belief is that the entire SoC ought to be coded in C and run as software in a native C environment, where running tool-generated cycle-accurate C will precisely represent the behavior of the generated RTL running in an HDL simulator. The architecture of the compiled RTL is congruent to the C source and you can achieve extensive functional verification in the C environment.

As shown in Figure 2, the CebaTech compiler generates a cycle-accurate C model in addition to synthesizable RTL. The functional correctness of the compiler-generated RTL can be shown using formal tools that prove equivalency with the cycle-

accurate C, which in turn has been validated in the pure C software environment. The compiler is able to map arrays and other C data structures onto on-chip or external memory resources. It has the structure of a typical software compiler, complete with linker. Thus, it can quickly compile designs ranging from small test cases to truly enormous software systems, making it feasible for a small team of engineers to implement full hardware offload for a system as complex as the OpenBSD TCP/IP stack. There are enormous time-to-market and die-size advantages to this methodology.

A Real-World FPGA Example

The CebaTech flow is unbounded by design size. It can address designs from small (several hundred lines of C code) to immense (45,000 lines or more, as is the case with TCP/IP). To give you an overview of an implementation with the CebaTech compiler, let's look at an FPGA conversion of a C-based compression algorithm. Although quite modest in the amount of code, this example demonstrates the power and uniqueness of the CebaTech flow for architecture exploration/design trade-offs with respect to area and performance. It highlights the narrowing of the verification gap and provides an excellent vehicle with which to demonstrate CebaTech's software-to-silicon methodology.

To illustrate the extent to which the CebaTech flow utilizes pure C source language constructs to describe system architecture, we developed and compiled six incremental versions of the compression code. All versions function identically when compiled and run natively on a computer system; however, each successive version improves the performance and synthesis results of the generated RTL, using a small number of source code changes for each step. (Further refinement beyond the architectures explored in this article is also possible.)

Architecture Revision Details	Xilinx XC4V – Resource Utilization (%)						Fmax (MHz)	
	Slices	Slice Flops	LUTs	SelectRAM	# ROM	Clocks	Area	Speed
1 - Baseline Architecture					1	146		
2 - Eliminate Redundant Functions	89	24	68	18	1	56	12	58
3 - Define Parallel Processes	44	14	33	18	1	33	40	69
4 - Optimize Any Shared Functions	20	8	17	18	1	33	35	57
5 - ROM and FIFO Improvements	20	9	18	18	3	12	36	52
6 - Utilize Block SelectRAM	20	9	18	10	3	12	36	52

Table 1 – The results of exploring six versions of compression source code

We can summarize the six versions of the compression source code as:

- Original restructuring of the source code. The bulk of the program is put into a “process” that can be directly compiled into an RTL module. This initial restructuring does not try for any optimization, and is indicated as Revision 1 in Table 1.
- A first cut at optimization where certain critical functions are placed into separate modules. This prevents them from in-line expansion and yields a baseline architecture, shown as Revision 2 in Table 1. Regrouping the functions into two parallel processes is shown as Revision 3. The area-optimized implementation yields a tripling of clock frequency and the number of clock cycles decreases from 56 to 33. Revision 4 involves functions that are common to different callers.
- Flagging these functions as shared results in a quick area reduction. Various other common optimizations – notably loop unrolling – occur during Revision 5. A process-splitting operation allowed three simultaneous accesses to what was previously a single ROM. This step brought the clock cycles from 33 down to 12.
- The final iteration in this example, shown as Revision 6, is an example of a vendor-specific optimization. The Xilinx® Virtex™-II family makes both

distributed and block on-chip RAM available. Using the latter implementation allows storage utilization to drop by almost 50%.

For all of these C-based design instances, we synthesized the resulting Verilog RTL netlist using Xilinx ISE™ software v8.1 against a Virtex-4 XC4VLX25-12 FPGA.

For this C-based design flow, we expended less than one week to understand the compression code base, create the test benches used for RTL simulation, perform the six design explorations, and synthesize and verify the results. You can see that the CebaTech approach enables much speedier product development and delivery with the best price/performance balance.

Conclusion

CebaTech has been using a compiler-driven ESL flow to achieve rapid prototyping of leading-edge networking products for several years. The compiler was forged in the crucible of high-performance SoC development, but is now available to other forward-looking development teams who need to quickly develop and deliver novel SoCs that strike the right balance between price and performance.

To learn more about CebaTech's ESL flow, visit www.cebatech.com and register to download our compiler white paper. Alternatively, you can call (732) 440-1280 to learn how to apply our flow to your current SoC design and verification challenges. 🌟

A Novel Processor Architecture for FPGA Supercomputing

The Mitrion Virtual Processor is a fine-grain massively parallel reconfigurable processor for FPGAs.

by Stefan Möhl
CTO
Mitronics Inc.
stefan.mohl@mitronics.com

Göran Sandberg
Vice President of Product Marketing
Mitronics Inc.
goran.sandberg@mitronics.com

Ever-increasing computing performance is the main development driver in the world of supercomputing. Researchers have long looked at using FPGAs for co-processors as a promising way to accelerate critical applications. FPGA supercomputing is now generating more interest than ever, with major system vendors offering general-purpose computers equipped with the largest Xilinx® Virtex™-4 FPGAs, thus making the technology accessible to a wider audience.

There are compelling reasons to give serious consideration to FPGA supercomputing, especially given the significant performance benefits you can achieve from using FPGAs to accelerate applications. Typical applications can be accelerated 10-100x compared to CPUs, making FPGAs very attractive from a price/performance perspective.

What is even more important is that FPGAs use only a fraction of the power per computation compared to CPUs. Increased computing power with traditional clusters requires more and more electrical power. Today, the cooling problems in large computing centers have become major obstacles to increased computing performance.

Plus, the development of FPGA devices is still keeping up with Moore's law. In practice, this means that the performance of FPGA-based systems could continue to double every 18 months, while CPUs are struggling to deliver increased performance.

FPGAs as Supercomputing Co-Processors

FPGAs are highly flexible general-purpose electronic components. Thus, you must consider their particular properties when using them to accelerate applications in a supercomputer.

Compared to a CPU, FPGAs are Slow

It may seem like a contradiction to claim that FPGAs are slow when they can offer such great performance benefits over CPUs. But the clock speed of an FPGA is about 1/10th of a CPU. This means that the FPGA has to perform many more operations in parallel per clock cycle if it is to outperform a CPU.

FPGAs are Not Programmable

(From a Software Developer's Perspective) Software developers taking their first steps in FPGA supercomputing are always surprised to find out that the P for programmable in the acronym FPGA means "you can load a circuit design." Without such a circuit design, an FPGA does nothing – it is just a set of electronic building blocks waiting to be connected to each other.

Creating a circuit design to solve a computational problem is not "programming" from a software developer's point of view, but rather a task for an experienced electrical engineer.

FPGAs Require Hardware Design

With their general-purpose electronic components, FPGAs are usable in any electronic hardware design. That is why they work so

well as computing co-processors in computers, provided that you have loaded the proper circuit design. Traditional development methods for FPGAs are focused on the hardware design aspects, which are very different from software development (see Table 1).

FPGAs in Supercomputers

Enable a New Paradigm

Putting FPGAs in computers to accelerate supercomputing applications changes the playing field and rules dramatically. Traditional hardware design methods are no longer applicable for several reasons:

- Supercomputer programmers are typically software developers and researchers who would prefer to spend their time on research projects rather than acquiring the expertise to design electronic circuits.
- The sheer complexity of the designs required to compute real-world supercomputing problems is prohibitive and usually not viable.
- Hardware design methods are targeted at projects where a single circuit design is used in a large number of chips. With FPGA supercomputers, each chip will be configured with many different circuit designs. Thus, the number of different circuit designs will be greatly multiplied.
- Hardware design tools and methods focus on optimizing the size and cost of the final implementation device. This is significantly less important in a supercomputer because the investment in the FPGA has already occurred.
- Programs run in supercomputers are often research tasks that continuously change as the researcher gains deeper

understanding. This requires the program to evolve over its life cycle. In contrast, circuit designs are usually created once and then used without changes for the life span of the target application.

The good news is that in a supercomputer, the FPGA is there to perform computing. This allows you to create software development methods for FPGAs that circumvent the complexity and methodology of general hardware design.

The Mitrion Virtual Processor

The key to running software in FPGAs is to put a processor in the FPGA. This allows you to program the processor instead of designing an electronic circuit to place in the FPGA. To obtain high performance in an FPGA, the circuitry of the processor design is adapted to make efficient use of the resources on the FPGA for the program that it will run. The result is a configuration file for the FPGA, which will turn it into a co-processor running your software algorithm. This approach allows you as a software developer to focus on writing the application instead of getting involved in circuit design. The circuit design process has already been taken care of with the Mitrion Virtual Processor from Mitronics.

A Novel Processor Architecture

How do you get good performance from a processor on an FPGA when typical FPGA clock speeds are about 10 times slower than the fastest CPUs? Plus, the FPGA requires and consumes a certain amount of overhead for its reconfigurability. This means that it has a 10-100x less efficient use of available chip area compared to a CPU. Still, to perform computations 10-100x faster than a CPU, we need to put a massively parallel processor onto the FPGA. This processor should also take advantage of the FPGA's reconfigurability to be fully adapted to the program that it will run. Unfortunately, this is something that the von Neumann processor architecture used in traditional CPUs cannot provide.

To overcome the limitations of today's CPU architecture, the Mitrion Virtual Processor uses a novel processor architecture that resembles a cluster-on-a-chip.

Hardware Design	Software Development
Driven by Design Cycle	Driven by the Code-Base Life Cycle
Main Concern: Device Cost (Size, Speed)	Main Concern: Development Cost and Maintenance
Precise Control of Electrical Signals	Abstract Description of Algorithm

Table 1 – Hardware/software development comparison

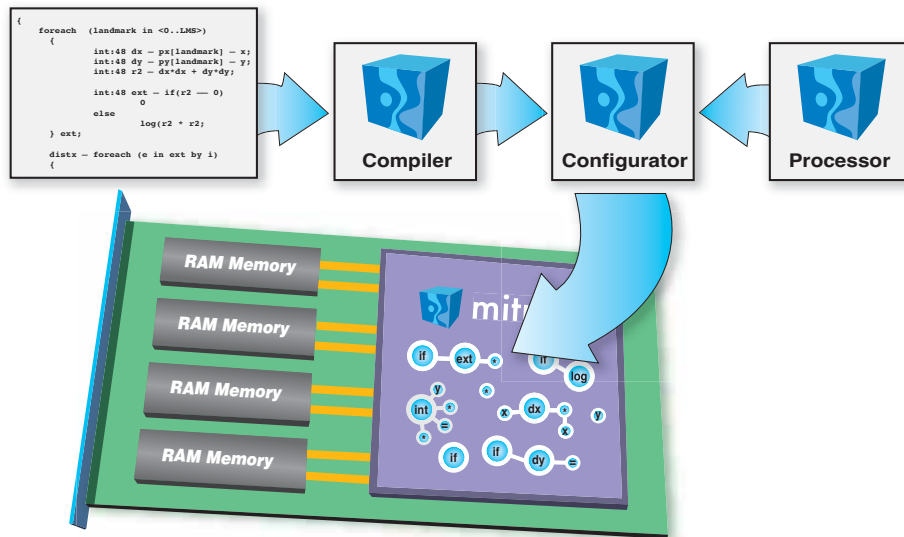


Figure 1 – The Mitrion Platform

Normal computer clusters comprise a number of compute nodes (often standard PCs) connected in a fixed network. The basic limitation of such a cluster is the network latency, ranging from many thousands up to millions of clock cycles, depending on the quality of the network. This means that each node has to run a large block of code to ensure that it has sufficient work to do while waiting for a response to a message. Lower latency in the network will enable each node to do less work between communications.

In the architecture of the Mitrion Virtual Processor, the entire cluster is on the same FPGA device. The network is fully adapted in accordance with the program requirements, creating an ad-hoc, disjunct network with simple point-to-point connections where possible, switched only where required. This allows the network to have a guaranteed latency of a single clock cycle.

The single clock cycle latency network is the key feature of the Mitrion Virtual Processor architecture. With a single-cycle latency network, it becomes possible for the nodes to communicate on every clock cycle. Thus, the nodes can run a block of code comprising only a single instruction. In a node that runs only one instruction, the instruction scheduling infrastructure of the node is no longer necessary, leaving

only the arithmetic unit for that specific instruction. The node can also be fully adapted to the program.

The effect of this full adaptation of the cluster in accordance with the program is that the problem of instruction scheduling has been transformed into a problem of data-packet switching. For any dynamic part of the program, the network must dynamically switch the data to the correct node. But packet switching, being a network problem, is inherently parallelizable. This is in contrast to the inherently sequential problem of instruction scheduling in a von Neumann architecture.

This leaves us with a processor architecture that is parallel at the level of single instructions and fully adapted to the program.

Mitrion-C

To access all of the parallelism available from (and required by) the Mitrion Virtual Processor, you will need a fully parallel programming language. It is simply not sufficient to rely on vector parallel extensions or parallel instructions.

We designed the Mitrion-C programming language to make it easy for programmers to write parallel software that makes the best use of the Mitrion Virtual Processor. Mitrion-C has an easy-to-learn C-family syntax, but the focus is on describing data dependencies rather than order of execution.

The syntax of Mitrion-C is designed to help you achieve high performance in a parallel machine, just like ANSI C is designed to achieve high performance in a sequential machine. Thus the Mitrion-C compiler extracts all of the parallelism of the algorithm being developed. We should note, though, that Mitrion-C is purely a software programming language; no elements of hardware design exist.

The Mitrion Platform

Together with Mitrion-C and the Mitrion Virtual Processor, the Mitrion Software Development Kit (SDK) completes the Mitrion Platform (see Figure 1). The Mitrion SDK comprises:

- A Mitrion-C compiler for the Mitrion Virtual Processor
- A graphical simulator and debugger that allows you to test and evaluate Mitrion-C applications without having to run them in actual FPGA hardware
- A processor configuration unit that adapts a Mitrion Virtual Processor to the compiled Mitrion-C code

The Mitrion Platform is tightly coupled to Xilinx ISE™ Foundation™ software for synthesis and place and route, targeting Xilinx Virtex-II and Virtex-4 devices. It features a diagnostic utility that analyzes the output from these applications and can report any problems it encounters in a format that does not require you to have FPGA design skills.

Conclusion

The novel, massively parallel processor architecture of the Mitrion Virtual Processor to run software in an FPGA is a unique solution. It is a solution readily available on the market today that addresses the major obstacles to the widespread adoption of FPGAs as a means to accelerate supercomputing applications. With the Mitrion Virtual Processor, you will not need hardware design skills to make software run in FPGAs.

For more information about the Mitrion Virtual Processor and the Mitrion Platform, visit www.mitrionics.com or e-mail info@mitrionics.com

Turbocharging Your CPU with an FPGA-Programmable Coprocessor

Coprocessor synthesis adds the optimal programmable resources.

by Barry O'Rourke
Application Engineer
CriticalBlue
barry.orourke@criticalblue.com

Richard Taylor
CTO
CriticalBlue
richard.taylor@criticalblue.com

What do you do when the CPU in a board-level system cannot cope with the software upgrades that the next-generation product inevitably requires? You can increase the clock speed, but when – not if – that approach runs out of steam, you must design in a faster CPU, or worse, an additional CPU. Worse yet, you could add another CPU board if the end product's form factor and cost target allow it.

In this article, we'll discuss some of the pros and cons of these approaches and

describe a programmable coprocessor solution that leverages an on-board Xilinx® FPGA to turbocharge the existing CPU and deliver the pros with none of the cons. You do not need processor design expertise, nor will you have to redevelop the software. Moreover, if the board already deploys an FPGA with sufficient spare capacity, the incremental silicon cost of the solution is zero.

The Brute Force Approach

Deploying faster or additional CPUs is an approach that is scalable as long as the software content growth remains relatively linear. However, software content growth is now exponential. A faster CPU might solve the problem temporarily, but it will soon be overwhelmed, necessitating a multiprocessor solution.

Many products cannot bear the redesign and additional silicon costs of a multiproces-

sor solution. Design teams do not often have the time to implement such a solution, and most teams certainly do not have the resources to re-architect the whole multiprocessor ensemble every time the software content exceeds the hardware's capacity to process it. It is more than just a hardware development problem – it is a major software partition and development problem.

When a methodology breaks, you must re-examine the fundamentals. Why can't a general-purpose (GP) CPU deliver more processing power? The answer is that most such CPUs offer a compromise between control functions and the parallel processing capability required by computationally intensive software. That capability is limited by a lack of both instruction-level parallelism and parallel processing resources.

When you deploy an additional general-purpose CPU to assist in the execution of, for example, a 20,000-line MPEG4 algo-

rithm, you add only limited parallel processing capability, along with unnecessary control logic, all wrapped up in a package that consumes valuable board space. It is an expensive, brute force approach that requires the multiprocessor partitioning of software that has generally been developed for only one processor; software redevelopment where the additional CPU's instruction set differs from that of the original CPU; the development of complex caching schemes; and the development of multiprocessor communication protocols – a problem exacerbated by the use of heterogeneous real-time operating systems.

However, even this may not deliver the requisite results. A major Japanese company recently published data that illustrates this point. The company partitioned software developed on one CPU for execution on four CPUs and achieved a maximum acceleration of only 2.83x. A less-than-optimal partition may have failed to utilize the additional parallel processing resources as efficiently as possible, but communications overhead surely played a role as well. In other words, maximizing CPU performance does not necessarily maximize system performance.

The Cascade Coprocessor Synthesis Solution

You can circumvent these problems by synthesizing an FPGA-implemented programmable coprocessor that, acting as an extension of the CPU, supplies the missing parallel processing capability. It increases both instruction-level parallelism and parallel processing resources. A mainstream engineer can design it in a matter of days without having any processor design expertise; CriticalBlue's Cascade synthesizer automatically architects and implements the coprocessor.

Moreover, Cascade optimizes cache architecture and communications overhead to prevent performance from being "lost in traffic." In other words, it boosts not only CPU performance but overall system performance.

The FPGA coprocessor accelerates software offloaded from the main CPU as-is, so no software partitioning and redevelopment are required, although you can also

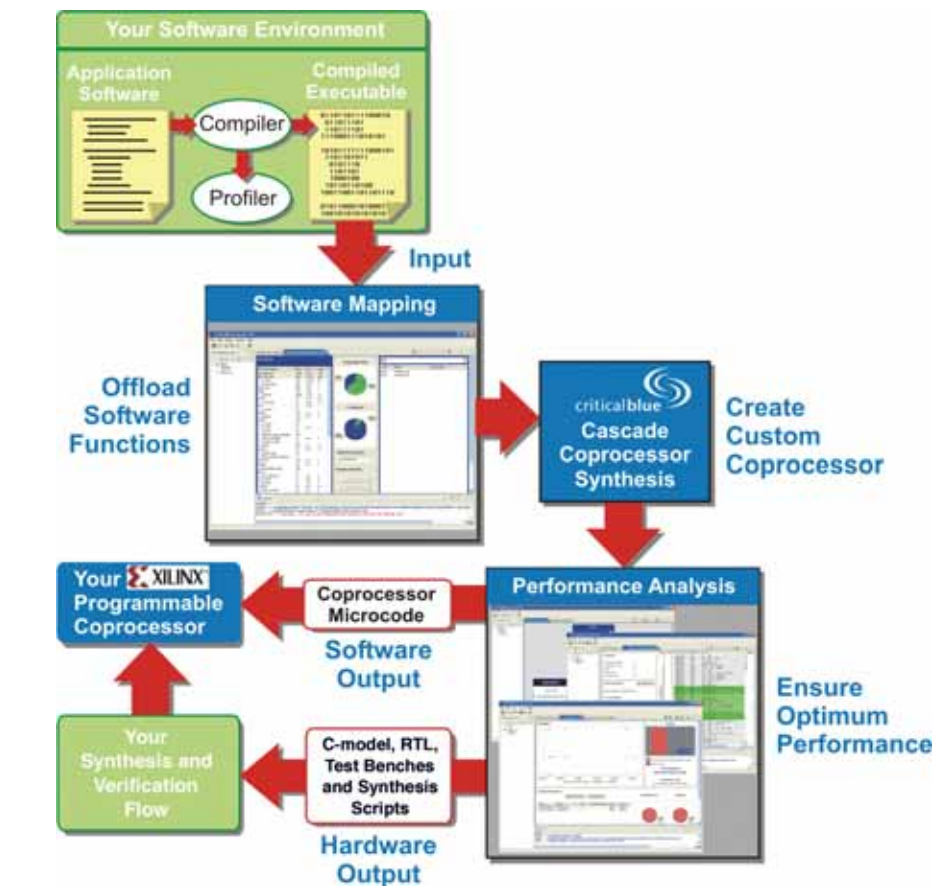


Figure 1 – Cascade coprocessor synthesis flow

co-optimize software and hardware. Cascade thus supports both full legacy software reuse and new software deployment. The former is especially important if you have inherited the legacy software and do not necessarily know how it works.

Unlike fixed-function hardware, a Cascade FPGA coprocessor is not restricted to executing only one algorithm, or part thereof. Cascade can synthesize an FPGA coprocessor that executes two or more disparate algorithms, often with only marginally greater gate counts than a coprocessor optimized to execute only one.

A Cascade FPGA coprocessor is reprogrammable. Although it is optimized to boost the execution of particular algorithms, it will execute other algorithms, too.

Adding a Cascade FPGA coprocessor circumvents the problems of adding a CPU. As you add additional software to the system, it can run on an existing FPGA coprocessor or you can add a new one opti-

mized for that software – all with software developed for single processor operation. Adding multiple coprocessors can be more effective, less costly, and less time consuming than deploying multiple processors.

How Does It Work?

First, you should determine which software routines must be offloaded from the CPU. Cascade analyzes the profiling results of the application software running on the CPU to identify cycle-hungry candidates (Figure 1).

Cascade then automatically analyzes the instruction code – including both control and data dependencies – and maps the selected tasks onto multiple coprocessor architecture candidates that comply with user-defined clock rate and gate count specifications. For each candidate, the tool provides estimates of performance, gate count, and coprocessor/CPU communication overhead. You then select the optimum architecture.

Cascade generates an instruction- and bit-accurate C model of the selected coprocessor architecture for use in performance analysis. Cascade uses the C model together with the CPU's instruction set simulator (ISS) and the stimuli derived from the original software to profile performance, as well as analyze memory access activity and instruction execution traces. The analysis also identifies the instruction and cache "misses" that cause so many performance surprises at system integration. Because Cascade can generate coprocessor architectures so quickly, you can perform multiple "what if" analyses quite rapidly. The model is also used to validate the coprocessor within standard C or SystemC simulation environments.

In the next step, Cascade generates the coprocessor hardware in synthesizable RTL code, which you verify using the same stimuli and expected responses as those used by the CPU. The tool generates the coprocessor/CPU communications circuitry. It also simultaneously generates coprocessor microcode, modifying the original executable code so that function calls are automatically directed to a communications library. The library manages coprocessor handoff and communicates parameters and results between the CPU and the coprocessor. Cascade generates microcode independently of the coprocessor hardware, allowing new microcode to be targeted at an existing coprocessor design.

The Cascade tool provides interactive features if you wish to achieve even higher performance. For instance, it enables you to manually optimize code, co-optimize code and hardware, and deploy custom functional hardware units to achieve greater processing performance. You can explore these manual options in a "what-if" fashion to determine the optimal configuration.

What Are The Results?

The results for single and multiple algorithm execution are shown in Table 1. Executing a single real-time image processing algorithm as-is was 5x faster than on the CPU. Software code optimization and a custom functional unit increased that to 15x.

Application	Single Algorithm		Multiple Algorithms		
Algorithm	Real-Time Image Processing		SHA-1	MD5	SHA-1 + MD5
Lines of Original Code	~200		150	700	850
Code Modified?	No	Yes	No	Yes	Same as MD5
Custom Functional Unit?	No	Yes	No	Yes	Same as MD5
Boost vs. CPU	5x	15x	5x	10x	6.4x
Effort (in Days)	1	3	1	5	1 More Day

Table 1 – Single and multiple algorithm execution

Application	Reprogrammability	
Algorithm	MP3 Decoder	MPEG1 Layer 2 Encoder
Lines of Original Code	~9,700	~16,700
Code Modified?	No	No
Custom Functional Unit?	No	No
Boost vs. CPU on its Own Dedicated Coprocessor	2.13x	1.62x
Boost vs. CPU on the Other's Coprocessor	1.18x	1.19x
Effort (in days)	2	1

Table 2 – Coprocessor reprogrammability with as-is code re-use

In the case of multiple algorithms, a coprocessor synthesized for SHA-1, a secure hash algorithm for cryptography applications, achieved a boost of 5x. A separate coprocessor with a custom functional unit synthesized for MD5, another hash algorithm, achieved 10x. A coprocessor optimized for both achieved 6.4x – using only 8% more gates than either of the two single coprocessors.

The power of reprogrammability is demonstrated in Table 2. One of our customers synthesized a coprocessor to execute an MP3 decoder algorithm, achieving a 2.13x boost despite using unoptimized code. The designer then generated a coprocessor to execute an MPEG1 Layer 2 encoder algorithm, also with no code optimization, achieving a 1.62x boost. When each algorithm was executed on the other's coprocessor, they achieved boosts of 1.18x and 1.19x, respectively. Code optimization would have improved this performance further, and custom functional units to execute serial operations even more.


How Can You Implement It?

You can implement Cascade FPGA coprocessors in multiple Xilinx families: Virtex™-II, Virtex-II Pro/Pro X, Spartan™-3/3E/3L, and Virtex-4 FPGAs.

Cascade's synthesizable RTL output works with Synopsys, Synplicity, and Xilinx synthesis tools and is fully integrated into the Xilinx ISE™ tool flow, while the verification approach works with the Xilinx ISE tool flow and Mentor Graphics's ModelSim XE/PE.

Cascade can target any Xilinx-populated board without translation or modification – no family-specific design kit is required.

Conclusion

If you want to boost the processing power of your design without deploying new or improved microprocessors, and if you want to use software without repartitioning and redevelopment, contact CriticalBlue at info@criticalblue.com. We can tell you how to do it – without having to become a processor designer. 

Tune Multicore Hardware for Software

Teja FP and Xilinx FPGAs give you more control over hardware configuration.

by Bryon Moyer
VP, Product Marketing
Teja Technologies, Inc.
bmoyer@teja.com

One of the key preferences of a typical embedded designer is stability in the hardware platforms on which they program; poorly defined hardware results in recoding hassles more often than not. But a completely firm, fixed hardware platform also comes with a set of constraints that programmers must live with. These constraints – whether simply design decisions or outright bugs – can force inelegant coding workarounds or rework that can be cumbersome and time-consuming to implement.

By combining an FPGA platform with a well-defined multicore methodology, you can implement high-performance packet processing applications in a manner that

gives software engineers some control over the structure of the computing platform, potentially saving weeks or months of coding time and reducing the risk of delays.

Much of the hardware design process goes into defining a board. Elements such as the type of memory, bus protocol, and I/O are defined up front. If a fixed processor is used, then that is also defined early on. But for gigabit performance on packet processing algorithms, for example, a single processor typically won't cut it, and multiple processors are necessary.

The best way to build a processing fabric depends on the software being run. By using an FPGA to host the processing, you can defer making specific decisions on the exact implementation until you know more about the needs of the code. The new Teja FP platform provides a methodology and multicore infrastruc-

ture on Xilinx® Virtex™-4 FPGAs, allowing you to decide on the exact configuration of the multicore fabric after the code has been written.

When Software Engineers Design Hardware

Hardware and software design are two fundamentally different beasts. No matter how much hardware design languages are made to look like software, it is still hardware design. It is the definition of structure, and processes ultimately get instantiated as structure. However, it is clear that software engineers are designing more and more system functionality using C programming skills; tools now make it possible for this functionality to target software or hardware.

The software approach is much more process-oriented. It is the “how to do it” more than the “what to build” because traditionally there has been nothing to build –

the hardware was already built. A truly software-based design approach includes key functionality that is not built into the structure but is executed by the structure in a deployed system. The benefit of software-based functionality is flexibility: you can make changes quickly and easily up until and even after the system has shipped. FPGAs can also be changed in the field, but software design turns can be handled much more quickly than hardware builds.

Because hardware and software design are distinct, designers of one or the other will think differently. A hardware engineer will not become a software engineer just by changing the syntax of the language. Conversely, a software engineer will not become a hardware engineer simply by disguising the hardware design to resemble software. So allowing a software engineer to participate in the design of a processing fabric cannot be done blithely. In addition, turning a hardware-oriented design over to a software engineer is not likely to be met with cheers by hardware designers, software designers, or project managers. Hardware decisions made by software engineers must be possible using a methodology that resonates with a software engineer in a language familiar to the software engineer.

The key topology of the processing fabric for a multicore packet processing engine is the parallel pipeline, shown in Figure 1. Such an engine comprises an array of processors, along with possible hardware accelerators. Solving the problem means asking the following questions:

- How many processors do I need?
- How should they be arranged?
- How much code and local data store does each processor require?
- What parts of the code need hardware acceleration?

Let's take these questions case by case and assemble a methodology that works for software engineers.

Processor Count and Configuration

The number of processors required is more or less a simple arithmetic calculation based on the cycle budget and the number of cycles required to execute code. The cycle budget is a key parameter for applications when you have a limited amount of time to do the work. With packet processing, for

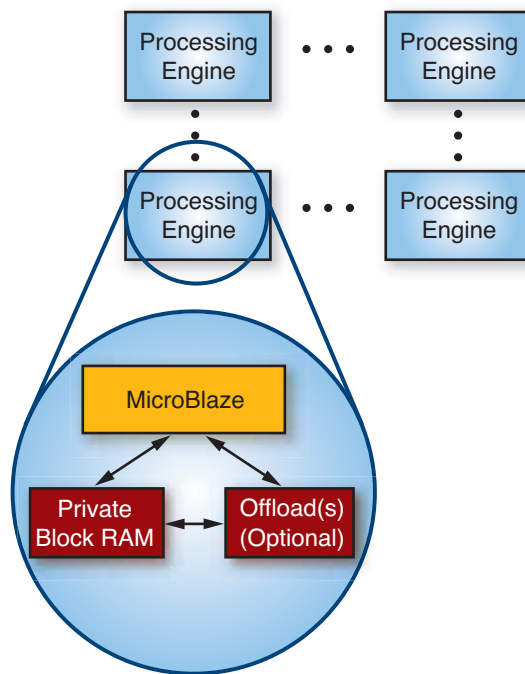


Figure 1 – A parallel pipeline where each engine comprises a MicroBlaze processor, private memory, and optional offloads.

example, the packets keep coming, so you only have so many cycles to finish your work before the next packet arrives. If your code takes longer than that, you need to add more processors. For example, if the cycle budget is 100 cycles and the code requires 520 cycles, then you need 6 processors (520 divided by 100 and rounded up). You can fine-tune this, but it works for budgetary purposes. The cycle count is determined by profiling, which you can perform using Teja's tools.

The next question is how to arrange the processors. The simplest way to handle this is to decide if you want to partition your code to create a pipeline. A pipeline uses less resources but adds latency. If you want to partition your code, then pick an obvious place – something natural that works

intuitively. There's no need to figure out where the exact cycle midway point is.

Let's say that with six processors you will have a two-stage pipeline. Now you need to figure out how many processors go in each stage; you can do this by profiling the cycle counts of each of the partitions and dividing again by the cycle budget. So if the first partition took 380 cycles, it will require 4 processors; that leaves 140 cycles for the second stage, which will require 2 processors. (The cycle counts of the two partitions won't actually add neatly to the cycle count of the unpartitioned program, but it is close enough for our purposes.) So this two-stage pipeline will have four processors in the first stage and two in the second stage. By using Xilinx MicroBlaze™ soft cores, you can instantiate any such pipeline given sufficient logic resources.

By contrast, a fixed pipeline structure would have a pre-determined number of processors in each stage. This would mandate a specific partition, which can take a fair bit of time to implement. Instead, the pipeline is made to order and can be irregular, with a different number of processors in each stage. So it doesn't really matter where the partition occurs. The hardware will be designed around the partitioning decision instead of vice versa.

The question of how a software engineer can implement this is key. Teja has assembled a set of APIs and a processing tool that allows the definition of the hardware platform in ANSI C; the tool executes that program to create the processing platform definition in a manner that the Xilinx embedded tools can process. This set of APIs is very rich and can be manipulated at a very low hardware level, but most software engineers will not want to do this. Thus, Teja has put together a "typical" pipeline definition in a parameterized fashion. All that is required to implement the pipeline in the example is to modify the two simple #define statements in a configuration header file.

The following statements define a two-stage pipeline, with four engines in the first stage and two in the second stage:

```
#define PIPELINE_LENGTH 2
#define PIPELINE_CONFIG {4,2};
```

From this and the predefined configuration program, the TejaCC program can build the pipeline. Of course, if for whatever reason the pipeline configuration changes along the way, similar edits can easily take care of those changes.

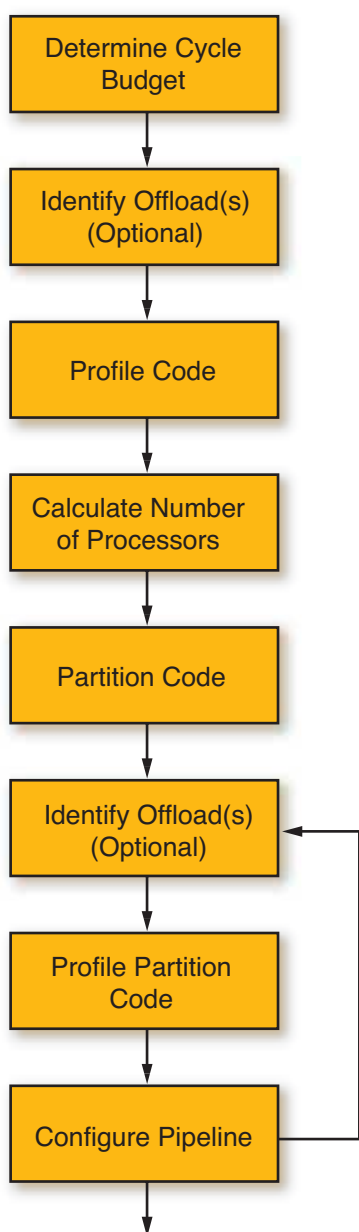


Figure 2 – Process for configuring a parallel pipeline

Memory

The third question is about the amount of memory required. In a typical system, you are allocated a fixed amount of code and data store. If you miss that target, a significant amount of work can go into squeezing things in. With FPGAs, however, as long as the amount of memory needed is within the scope of what's available in the chip, it is possible to allocate a tailored memory size to each processor. More typically, all would have the same size (since there is a 2K minimum allocation block, limiting the amount of fine tuning possible). Code compilation provides the size required, and the configuration header file can be further edited with the following statements, which in this example allocates 8 KB of memory for both code and data store:

```
#define CPE_CODE_MEM_SIZE_KB 8
#define CPE_DATA_MEM_SIZE_KB 8
```

Offloads for Acceleration

The fourth question deals with the creation of hardware accelerators. There may be parts of the code that take too many cycles. Those cycles translate into more processors, and a hardware accelerator can eliminate some of those processors. As long as the hardware accelerator uses fewer gates than the processors it is replacing, this will reduce the size of the overall implementation.

Teja has a utility for creating such accelerators, or offloads, directly from code. By annotating the program, the utility will create:

- Logic that implements the code
- A system interface that allows the accelerator to plug into the processor infrastructure
- An invocation prototype that replaces the original code in the program
- A test bench for validating the offload before integrating it into the system

Once an offload is created, the cycle count is reduced, so you will have to re-evaluate the processor arrangement. But because it is so easy to redefine the pipeline structure, this is a very straightforward task.

A Straightforward Methodology

Putting all of this together yields the process shown in Figure 2. You can define offloads up-front (for obvious tasks) or after the pipeline has been configured (if the existing code uses too many processors).

The controls in the hands of the software engineer are parameters that are natural or straightforward for a software engineer and expressed in a natural software language – ANSI C. All of the details of instantiating hardware are handled by the TejaCC program, which generates a project for the Xilinx Embedded Development Kit (EDK). EDK takes care of the rest of the work of compiling/synthesis/placing/routing and generating bit-streams and code images.

In this manner, hardware engineers can design the boards, but by using FPGAs they can leave critical portions of the hardware configuration to the software designer for final implementation. This methodology also lends itself well to handling last-minute board changes (for example, if the memory type or arrangement changes for performance reasons). Because the Teja tool creates the FPGA hardware definition, including memory controllers and other peripherals, board changes can be easily accommodated. The net result is that the hardware target can be adapted to the software so that the software designer doesn't have to spend lots of time coding around a fixed hardware configuration.

The Teja FP environment and infrastructure make all of this possible by taking advantage of the flexible Virtex-4 FPGA and the MicroBlaze core. Armed with this, you can shave weeks and even months off of the development cycle.

Teja also provides high-level applications; these applications can act as starting points for launching a project and reducing the amount of work required. By combining a time-saving flexible methodology with pre-defined applications, creators of networking equipment can complete their designs much more quickly.

For more information on Teja FP, contact bmoyer@teja.com.

Automatically Identifying and Creating Accelerators Directly from C Code

The Mimosys Clarity tool exploits the power of the Virtex-4 FX PowerPC APU interface.

by Jason Brown, Ph.D.
CEO
Mimosys
jason.brown@mimosys.com

Marc Epalza, Ph.D.
Senior Hardware Engineer
Mimosys
marc.epalza@mimosys.com

Let's say that you have been tasked to ensure that your company has an H.264 solution that supports high-definition video decoding at 30 frames per second. You are not a video expert. What do you do? You could get on the Internet and perform a Web search for H.264; before you know it, you'll have the source code and be on your way.

You managed to compile the code and get it running on the target, but it decodes at a whopping two frames per second. Now what? After sifting through pages and pages of profiling data, you find some hotspots, but you are not sure which parts to focus on to maximize the acceleration and you do not have enough time to try to optimize them all.

Many of us have found ourselves in this situation at one time or another. Maybe you have even delivered a solution, but not without a lot of sweat and tears.

Mimosys Clarity

The Mimosys Clarity software tool automatically identifies candidate hardware accelerators directly from application C source code, guided by the execution profile of the application. It also takes a set of constraints on the number of I/O parameters available to the accelerator and a model of the execution costs for operations on the PowerPC™ and in Xilinx® FPGA fabric.

Clarity's approach to profiling is unique in that the execution profiling information is visually presented in the tool as annotations to basic blocks of control flow graphs (CFGs), as shown in Figure 1. The nodes of a control flow graph represent the basic blocks of the C source code, where a basic block is any sequence of C expressions without conditions. The edges of a CFG represent an execution path between basic blocks, where a specific path is followed (or taken) if a particular condition is true or false.

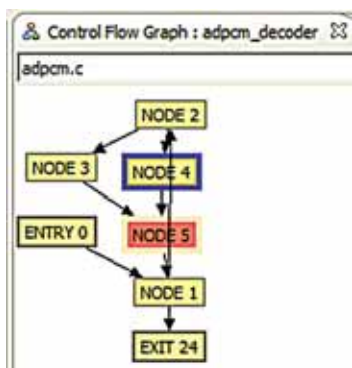


Figure 1 – Control flow graph (CFG); each node represents a basic block (DFG).

Because each basic block is a sequence of expressions, it can be represented by a data flow graph (DFG), the nodes of which are operations (+, -, *, /) and the edges values (Figure 2). In the vast majority of cases, this information is not automatically created or visualized. Furthermore, this visualization, if done at all, is typically static and thus unable to retain one of the most important aspects: the correspondence between the graphs and the source code from which they came.

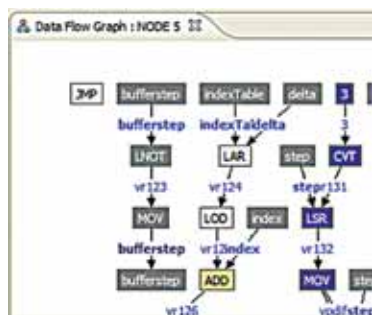


Figure 2 – Data flow graph (DFG); each node represents a single operation.

This unique approach of visually representing profiling information in CFGs and DFGs, along with the aforementioned correspondence, helps you quickly hone in on and understand the most computationally intensive parts of the application, as all of the views are always synchronized.

Once you have gathered the application profiling information, you can invoke the automatic accelerator identification search algorithm. This algorithm identifies a set of optimal hardware accelerators that provide the best execution speedup of the application given the constraints.

An important constraint on accelerators is the number of available data inputs and the number of data outputs, which in the current Virtex™-4 FX device is fixed by the PowerPC to two inputs and one output. Figure 3 shows the results from two invocations of the accelerator search algo-

provides little or no benefit, it indicates that you need to rework the application source code to expose improved acceleration opportunities. Furthermore, the automatic identification algorithm takes both local (DFG) and global (execution profile) information into account, thus providing unique insight into where to focus your efforts in optimizing an application.

In the iterative process of optimizing an application, the identification algorithm will discover a set of accelerators that will satisfy your requirements. However, the task remains to realize them on the Virtex-4 FX platform. You must perform two important steps to achieve this: first, create an HDL definition of the accelerators that includes the necessary logic to interface the PowerPC APU port with the accelerator itself. You must assign the new accelerators unique identifiers

Name	Inputs	Outputs	Cycles Saved	Est. SW Latency	Est. HW Latency
ISE	2	1	38.10 %		
Instr_0	2	1	1.52 %	2	1 (0.16)
Instr_1	2	1	27.43 %	10	1 (0.90)
Instr_2	2	1	6.10 %	3	1 (0.75)
Instr_3	1	1	3.05 %	2	1 (0.25)
Name	Inputs	Outputs	Cycles Saved	Est. SW Latency	Est. HW Latency
ISE	4	1	47.24 %		
Instr_0	2	1	1.52 %	2	1 (0.16)
Instr_1	3	1	39.62 %	15	2 (1.40)
Instr_2	2	1	6.10 %	3	1 (0.75)

Figure 3 – Instructions found with a constraint of 2-1 (top) and 4-1 (bottom), with corresponding performance increases.

gorithm. The upper table uses a constraint of two inputs and one output, with the lower using a constraint of four inputs and one output for the application ADPCM (adaptive differential pulse code modulation).

Clearly, the number of I/Os has a significant impact on the acceleration achieved. In order to realize the higher acceleration while overcoming the hard constraints imposed by the PowerPC, you can use pipelining techniques at the cost of increasing design complexity. Clarity automatically pipelines accelerators, giving access to the higher performance with no extra work for you.

The identified hardware accelerators are optimal; no DFG of the application considered by Clarity contains a sub-graph with better estimated performance. As a consequence, if the set of accelerators discovered

(instructions) so that the PowerPC can be instructed to activate the accelerator by the software application.

The second step is to modify the original application source code to use the new instructions, thus achieving the execution speedup offered by the new hardware accelerators. In a normal design flow both of these steps are manual, although in some ESL design flows you can describe the hardware accelerators using C/C++/SystemC or a derivative programming language. But the software integration step is always manual.

Realizing the Accelerators

Clarity automates the entire process of creating accelerators for Virtex-4 FX FPGAs, including software integration. This is achieved when you commit the newly dis-

covered accelerators to the Xilinx platform. The commit stage is fully automatic and comprises three parts:

- Modifying the original application source code to use the newly generated accelerators. The source code of the application is modified so that the parts to be implemented in the FPGA are removed and replaced by calls corresponding to the new hardware accelerators. Figure 4 shows a snippet of code from the ADPCM application, in which the removed source code is commented out and replaced with a call to the macro `Instr_1`. This macro contains the special PowerPC instruction used to activate the hardware accelerator corresponding to the `Instr_1` that Clarity discovered and you selected.
- Generating RTL for each accelerator along with the necessary logic to interface the Xilinx PowerPC APU and the accelerator data path. Each accelerator is implemented in VHDL as an execution data path along with the required PowerPC interfacing logic. The data path is translated directly from the C source code and is thus correct by construction. This translation is possible because the data path implements a pure data flow with limited control, avoiding the issues of high-level synthesis. As described above, if the I/O constraints specified for the search exceed those of

the PowerPC architecture, the data path will automatically be split into stages and pipelined to fit these constraints.

- Creating a Xilinx Platform Studio (XPS) project containing the RTL implementations and modified application source code. To provide confidence in the correctness and synthesized result of the HDL accelerators, you can have a test bench created automatically for each accelerator along with the necessary simulation and synthesis scripts. You can invoke an HDL simulator such as ModelSim from within Clarity to compare the functionality of the original C code with the HDL code of the accelerator replacing it. The generated synthesis script enables you to perform synthesis on each accelerator in the target technology and obtain information on the critical path and area. Furthermore, this verification test bench provides a framework to ensure the correctness of identified accelerators that may be subsequently modified by a design engineer.


Hardware Design for Software Engineers

The challenge of identifying hardware accelerators for an application is formidable, especially if expert domain knowledge is required but unavailable. This challenge is made more acute by the difficulties in realizing these accelerators as coprocessors, requiring new interfaces and software inte-

gration to manipulate them. By providing a standardized interface, the Virtex-4 FX PowerPC enables new automation technologies that address both identification and realization of accelerators. Furthermore, Clarity can automatically circumvent the apparent I/O limitations of the PowerPC APU.

In particular, Clarity offers a unique solution to this challenge through an innovative combination of automation and visualization techniques (see Figure 5). Working at the level of the C programming language, you can visualize all aspects of the control flow, data flow, and execution behavior of an application. A unique hardware accelerator identification technology automatically discovers and creates application-specific hardware accelerators targeted to the Xilinx Virtex-4 FX device. Fully automatic HDL generation, application software integration, and test bench generation mean that you are freed from any concerns about how to realize application acceleration in hardware, thus empowering you to focus on your product differentiation.

So finding the H.264 source code on the Web was not such a bad idea. You created some useful accelerators and implemented them on the Virtex-4 FX device before lunch time, leaving the afternoon free to explore some different solutions just for fun.

For more information about Clarity, please e-mail enquiries@mimosys.com or visit www.mimosys.com/xilinx. 

```

adpcm.c: X
/* Step 2 - Find new index value (for later) */
index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;

/* Step 3 - Separate sign and magnitude */
sign = delta & 8;
/* MIMOSYS delta = delta & 7; */
Instr_1(delta, step, &vpdiff);

/* Step 4 - Compute difference and new predicted value */
/*
** Computes 'vpdiff = (delta+0.5)*step/4', but see comment
** in adpcm_coder.
*/
/* MIMOSYS vpdiff = step >> 3; */
/* MIMOSYS if ( delta & 4 ) vpdiff += step; */
/* MIMOSYS if ( delta & 2 ) vpdiff += step>>1; */
/* MIMOSYS if ( delta & 1 ) vpdiff += step>>2; */

if ( sign )
    valpred -= vpdiff;
else
    valpred += vpdiff;

```

Figure 4 – Modified application; the code moved into the hardware accelerator has been commented out and replaced with a call to the accelerator. The code is from ADPCM.

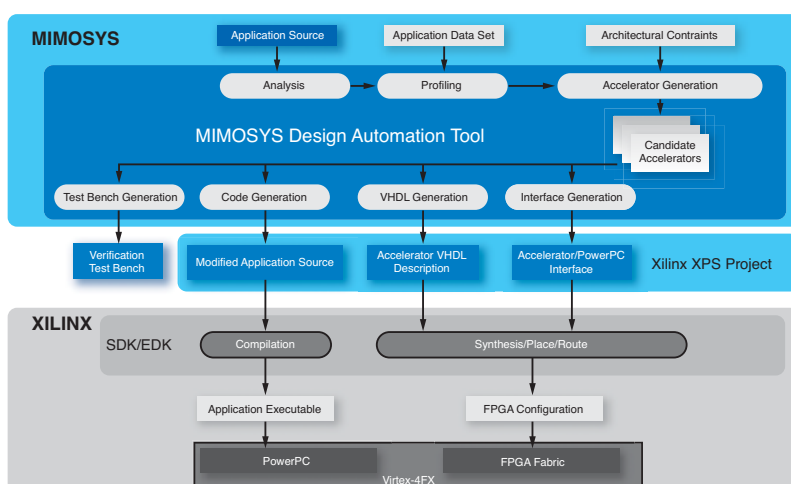


Figure 5 – Mimosys Clarity flow; from the application source code and some constraints, you can generate a complete Xilinx Platform Studio project with accelerators.

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

ExploreAhead Extends the PlanAhead Performance Advantage

By leveraging computing resources, you can achieve improved design results.

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

Robert Shortt
Senior Staff Software Engineer
Xilinx, Inc.
robert.shortt@xilinx.com

David Knol
Senior Staff Software Engineer
Xilinx, Inc.
david.knol@xilinx.com

Brian Jackson
Product Marketing Manager
Xilinx, Inc.
brian.jackson@xilinx.com

FPGA design sizes and complexity are now in the stratosphere. An increasing number of designers are struggling to meet their design goals in a reasonable amount of time.

Xilinx introduced PlanAhead™ software as a means to combat lengthening design closure times. The PlanAhead hierarchical design and analysis tool helps you quickly comprehend, modify, and improve your designs. Earlier PlanAhead versions (7.1 and earlier) were used to improve performance of the design through floorplanning. The software does encapsulate Xilinx® ISE™ back-end tools to complete the EDIF/NGC-to-bitstream flow. However, earlier versions left the complex job of design closure through place and route options to the user.

With ExploreAhead, you can order multiple implementation runs based on the strategies you defined or on the predefined strategies shipped as factory defaults. These runs can be parallelized to take advantage of multi-core CPU machines.

PlanAhead 8.1 design tools introduce ExploreAhead, which simplifies the tedious task of wading through myriad place and route options to obtain the best achievable QoR for a given floorplan. ExploreAhead also enables an effective use of multi-core CPU machines to speedup the design convergence process.

ExploreAhead is an implementation exploration tool. It manages multiple

predefined strategies. Xilinx has tested these predefined strategies and found them to be some of the most effective techniques to get better performance on designs. These factory-default strategies are prioritized by their effectiveness. Predefined strategies eliminate the need for you to learn new options each time a new version of ISE software is released to achieve the best QoR.

Expert users are encouraged to craft strategies suitable for their designs. User-defined strategies are stored under \$HOME/.hdi/strategies for Unix users and C:\documents and settings\%HOME%\application data\hdi\strategies for Microsoft Windows users. These are simply XML files for teams of users to share. Design groups wanting to create group-wide custom strategies accessible to anyone using PlanAhead software can copy user-defined strategies to the <InstallDir>/strategies directory.

Id	Name	Flow	Status	Progress	Start	Elapsed	Timing Score	Description
1	Run #1	ISE B	Complete	100%	12/15/05 11:15 AM	00:04:48	425 8.1 Default Settings	
2	Run #2	ISE B	Complete	100%	12/15/05 11:20 AM	00:06:41	448 PAR -ci med	
3	Run #3	ISE B	Complete	100%	12/15/05 11:26 AM	00:11:11	12 PAR -ci high	
4	Run #4	ISE B	Running MAP...	20%	12/15/05 11:28 AM	00:01:47	Map -timing -ci high	
5	Run #5	ISE B	Queued...	0%			Map -logic-opt on -retiming on -register_duplication on	

Figure 1 – Multiple implementation runs with varying strategies

implementation runs of your design through the NGDDBuild, map, and place and route steps. ExploreAhead allows you to create, save, and share place and route options as “recipes” or “strategies.” With ExploreAhead, you can order multiple implementation runs based on the strategies you defined or on the predefined strategies shipped as factory defaults. These runs can be parallelized to take advantage of multi-core CPU machines.

ExploreAhead manages reports and statistics on the runs, allowing you to pick the best implementation for your design. Figure 1 shows an illustration of ExploreAhead.

Strategy

A “strategy” is defined as a set of place and route options. This is a recipe that you can use to implement a single place and route run. Strategies are defined by ISE release and encapsulate command-line options for each of the implementation tools: NGDDBuild, map, and place and route.

Using strategies is a very powerful concept that makes ad hoc management of your place and route options a seamless task. ExploreAhead ships with a set of

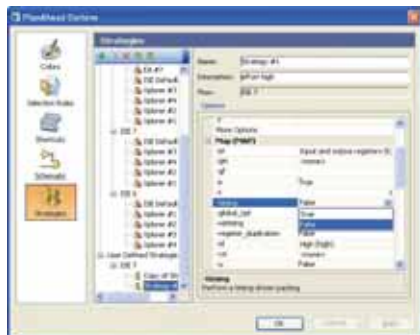


Figure 2 – Strategies editor

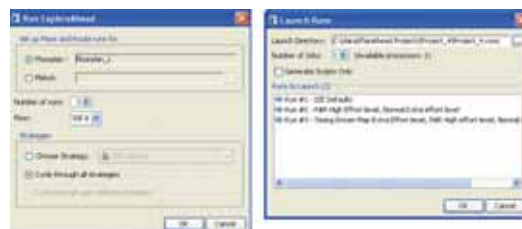


Figure 3 – Run ExploreAhead and Launch Runs dialog boxes

ExploreAhead also introduces an easy-to-use strategies editor for you to create your own favorite strategy. Figure 2 shows the strategies editor.

Run

ExploreAhead introduces the concept of a “run” object. Once launched, the run will work through the back-end tools to implement the design. Each run is associated with a set of place and route options or a defined strategy. ExploreAhead gives you the capability to launch multiple implementation runs simultaneously.

Launching an ExploreAhead run is a two-step process. The first step involves queuing up the run with different strategies. The second step will actually launch the place and route tools on each of the runs. The two dialog boxes in Figure 3 show the two steps.

Once you have interacted with the “Run ExploreAhead” dialog box and generated the required set of runs, a summary table of runs appears in the PlanAhead console window. Figure 1 displays one such table of runs. Each of the runs is selectable. Selecting a run will display the properties of this run in the PlanAhead properties window. Selecting one or many runs and hitting the launch button will bring up the launch dialog box. Here ExploreAhead will allow you to start multiple place and route runs simultaneously on a multi-core CPU machine. ExploreAhead will push all of the requested place and route runs into a queue

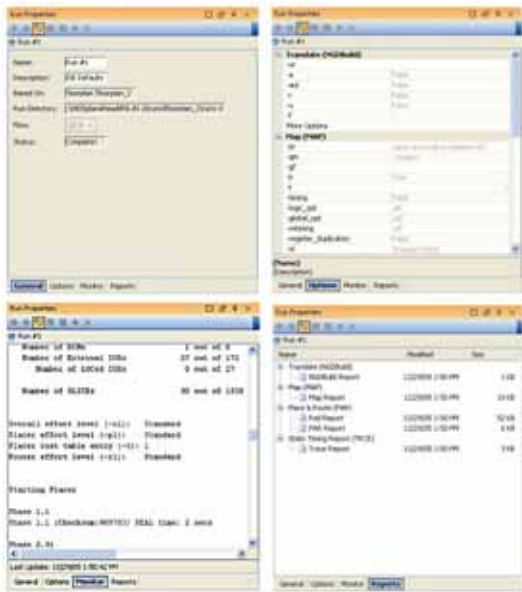


Figure 4 – Selected run properties: general, place and route options, live monitor, reports

and will then launch runs from this queue when CPU resources become available.

Monitor

ExploreAhead has an easy-to-use interface to monitor ongoing runs. The “summary runs” table in the console window helps you quickly browse through the relevant characteristics of a run: the CPU time implications of a strategy, percent completion, timing results, and description of the strategy employed. In addition to the summary results table, there is also a live “run monitor” that displays all console messages gen-



Figure 5 – Import Run dialog box

erated by place and route tools. You can simply select a run and tab over to the monitor tab of the property dialog box to engage the console messages. Figure 4 shows the properties dialog box for a single run.

Reports

The reports tab in the “Run Properties” dialog box shown in Figure 4 lists all of the potential reports that could be produced during an implementation run. These reports are grayed out at the start of the run and become available for browsing as the run proceeds through the various back-end steps. Simply double-click on each of the available reports to add it to the desktop browser.

Results

Once the ExploreAhead runs are complete and you have all of the reports for all of their runs at your disposal, you can then decide to import the results into

PlanAhead software for further investigation. After selecting a run, you can then right-click to import the run, which will also allow you to import placement and timing results into PlanAhead design tools. Figure 5 shows the import run dialog box.

Project

PlanAhead 8.1 software introduces the PlanAhead project repository. The PlanAhead project will save your ExploreAhead run information. ExploreAhead acknowledges that some of the place and route runs can take a significant amount of run time. If you launch a large number of runs, this can also add to your total completion time. As such, PlanAhead software allows you to exit the program, allowing the place and route runs to continue on your machine. You can then launch PlanAhead design tools at a later time; it will re-open the project, re-engage the monitors, and open the summary run tables the report files. This powerful feature allows you to free up a PlanAhead license during place and route runs.

ExploreAhead Design Flow

You can employ floorplanning – a key enabling methodology – within the PlanAhead framework in conjunction with ExploreAhead to get the best-in-class

QoR. You can use ExploreAhead on a floorplanned design or on a single Pblock. You can then piece the design together using a combination of floorplanning, a block-based implementation approach, and incremental design techniques. ExploreAhead, however, makes no assumptions as to the need to floorplan a design. The basic ExploreAhead design flow, shown in Figure 6, requires no floorplanning.

Conclusion

ExploreAhead expands the PlanAhead portfolio to include QoR optimization in the implementation tools. ExploreAhead brings together several key technologies to make PlanAhead design tools an extremely productive environment for interactive design exploration and closure. The combination of multiple what-if floorplans and what-if ExploreAhead runs on each of these floorplans expands the search space of possibilities enormously.

ExploreAhead enables you to search through this large space effectively to pick the most optimal implementation solution for your design. ●●●

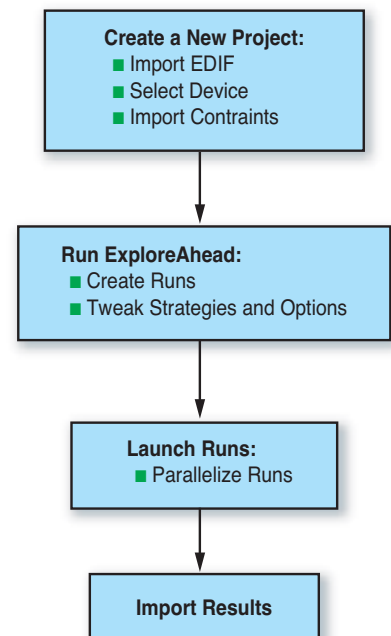


Figure 6 – Basic PlanAhead/ExploreAhead flow diagram

Honey, will you please tell Alex to stop programming the FPGA!



Viva! Absurdly Easy.

Viva! Software

The Easiest Development Tool for Programming FPGAs

Viva may just be the easiest and most intuitive development tool ever created for FPGAs. That's because only Viva delivers a true data set recursive and polymorphic object oriented programming language with a graphical, drag-and-drop interface. In fact, Viva is so easy you can get the hang of it in a just few minutes. Whether you're working with FPGAs in high-performance or embedded environments, Viva is the right solution for you. And Viva is perfectly suited for use with hardware from SGI, Nallatech and Digilent, including Xilinx Starter Kits. Find out why organizations like NASA and the US Air Force use Viva! software to program FPGAs. Visit www.starbridgesystems.com/viva or call (801) 984-4444 today.



Designing Flexible, High-Performance Embedded Systems

Soft processing and customizable IP enable the best of both worlds.

by Jay Gould
Product Marketing Manager
Xilinx, Inc.
jay.gould@xilinx.com

Which do you want in your next embedded project: flexible system elements so that you can easily customize your specific design, or extra performance headroom in case you need more horsepower in the development cycle? Why put yourself under undue development pressure and settle for one or the other? Soft processing and customizable IP offer the best of both worlds, integrating the concepts of custom design and co-processing performance acceleration.

Discrete processors offer a fixed selection of peripherals and some kind of performance ceiling capped by clocking frequency. Embedded FPGAs offer platforms upon which you can create a system

with a myriad of multiple customizable processor cores, flexible peripherals, and even co-processing offload engines. You now have the power to design an uncompromised custom processing system to satisfy the most aggressive project requirements and punch a hole in that performance ceiling, while maximizing system performance by implementing accelerated software instructions in the FPGA hardware. With FPGA fabric acceleration, the sky's the limit.

Flexibility

In addition to the high-performance PowerPC™ hard-processing core available in Xilinx® Virtex™ Platform FPGAs and the small footprint PicoBlaze™ microcontroller core programmed with assembly language, Xilinx offers you the choice of designing with a customizable

general-purpose 32-bit RISC processor. The MicroBlaze™ soft processor is highly flexible because it can be built out of the logic gates of any of the Virtex or Spartan™ families, and you can customize the processing IP peripherals to meet your exact requirements.

With customizable cores and IP, you create just the system elements you need without wasting silicon resources. When you build a processing system in a programmable device like an FPGA, you will not waste unused resources in a discrete device, nor will you run out of limited peripherals if you require more than what are offered (say your design requires three UARTs and your discrete device offers only one or two). Additionally, you are not trapped by your initial architecture assumptions; instead, you can continue to dramatically modify and tune your system

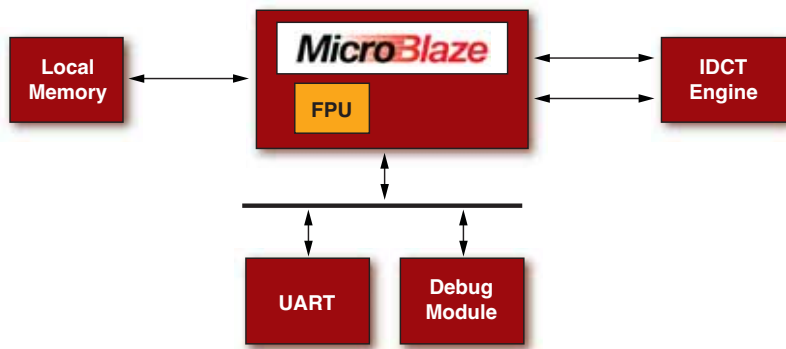


Figure 1 – Simple MicroBlaze block diagram

architecture and adapt to changes in newly required features or changing standards.

One FIR filter design example from the workshop materials for the 2006 Embedded System Conference had a MicroBlaze system configured with an optional internal IEEE 754-compliant floating-point unit (FPU), which facilitates a significant performance increase over software-only execution on the processor core. By including optional MicroBlaze components, you can quickly improve the performance of your application.

A side benefit of these optional internal components is that they are fully supported by the MicroBlaze C compiler, so source-code changes are unnecessary. In the FIR filter design example, including the FPU and recompiling the design meant immediate performance improvements, as calls to external C library floating-point functions are automatically replaced with instructions to use the new FPU.

Utilizing specialized hardware-processing components improves processor performance by reducing the number of cycles required to complete certain tasks by orders of magnitude over software recoding methods. The simple block diagram in Figure 1 represents a MicroBlaze processing system with an internal FPU IP core, local memory, and choice of other IP peripherals such as a UART or a JTAG debugging port. Because the system is customizable, we could very well have implemented multiple UARTs or other IP peripheral cores from the Xilinx processor IP catalog, including a DMA controller, IIC, CAN, or DDR memory interface.

The IP catalog provides a wide variety of other processing IP (bridges, arbiters, interrupt controllers, GPIO, timers, and memory controllers) as well as customization options for each IP core (baud rates and parity bits, for example) to optimize elements for feature, performance, and size/cost. Additionally, you can configure the processing cores with respect to clock frequency, debugging modules, local memory size, cache, and other options. By merely turning on the FPU core option, here at Xilinx we built a MicroBlaze system that optimized the aforementioned FIR implementation from 8.5 million CPU cycles to only 177,000 CPU cycles, enabling a per-

formance improvement of 48x with no changes to the C source file.

In a second example, we'll build on an additional design module, implementing an IDCT engine for an MP3 decoder application that will accelerate the application module by more than an order of magnitude.

You can easily create both processor platform examples referenced here with a development kit like the one depicted in Figure 2. The Integrated Hardware/Software Development Kit includes a Virtex-4 reference board that directly supports both PowerPC and MicroBlaze processor designs. The kit also includes all of the compiler and FPGA design tools required, as well as an IP catalog and pre-verified reference designs.

With the addition of a JTAG probe and system cables, the kit allows you to have a working system up and running right out of the box before you start editing and debugging your own design changes. Development kits for various devices and boards are available from Xilinx, our distributors, and third-party embedded partners.

Locate Bottlenecks and Implement Co-Processing

The MicroBlaze processor, one of EDN's Hot 100 Products of 2005, utilizes the IEC (International Engineering Consortium)



Figure 2 – Integrated Hardware/Software Development Kit

award-winning Xilinx Platform Studio (XPS) embedded tool suite for implementing the hardware/IP configuring and software development. XPS is included in our pre-configured Embedded Development Kits and is the integrated development environment (IDE) used for creating the system. If you have a common reference board or create your own board description file, then XPS can drive a design wizard to quickly configure your initial system.

Reduce errors and learning curves with intelligent tools so that you can focus your design time on adding value in the end application. After creating the basic configuration, you can spend your time iterating on the IP to customize your specific system and then develop your software applications.

XPS provides a powerful software development IDE based on the Eclipse framework for you power coders. This environment is ideal for developing, debugging, and profiling code to identify the performance bottlenecks that hide in otherwise invisible code execution. These inefficiencies in the code are often what makes a design miss its performance requirement goals, but they are hard to detect and often even harder to optimize.

Using techniques like “in-lining code” to reduce the overhead of excessive function calls, you can improve application performance 1%-5%. But with programmable platforms, more powerful design techniques now exist that can yield performance improvements by an order or two in magnitude.

Figure 3 shows a montage of Platform Studio views for performance analysis. XPS displays profiling information in a variety of forms so that you can identify trends or individual offending routines that spike on performance charts. Bar graphs, pie charts, and metric tables make it easy to locate and identify function and program inefficiencies so that you can take action to improve those routines that leverage the most benefit for total system performance.

Soft-Processor Cores with Their Own IP Blocks

For the MP3 decode example that I described earlier, we built a custom system (Figure 4), starting with the instantiation of multiple MicroBlaze processors. Because the

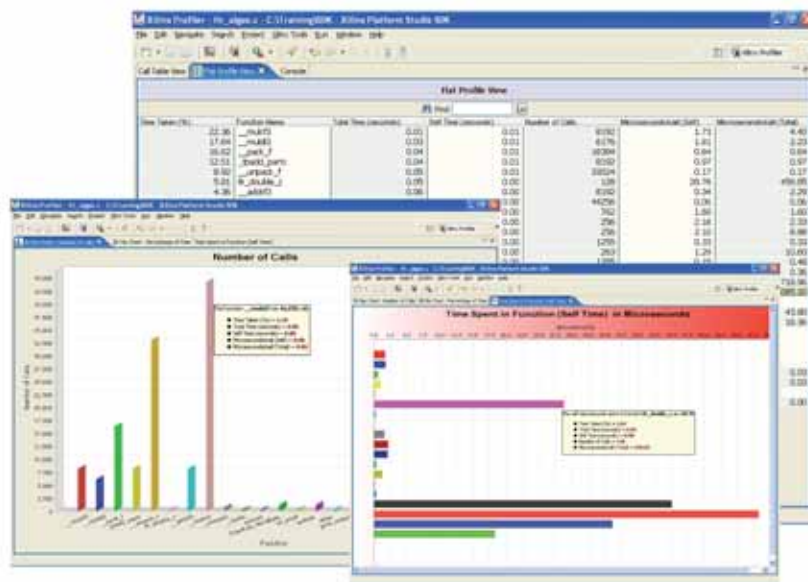


Figure 3 – Platform Studio embedded tool suite

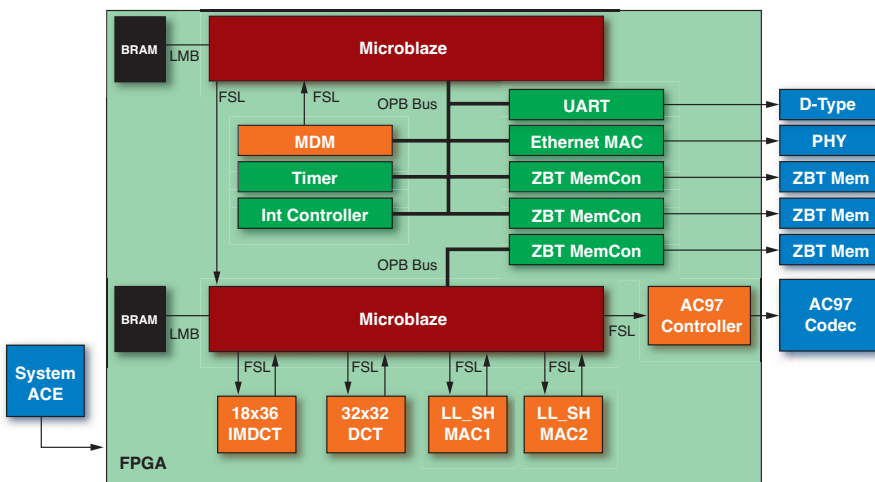


Figure 4 – MicroBlaze MP3 decoder example

MicroBlaze processor is a soft-core processor, we can easily build a system with more than one processor and balance the performance loading to yield an optimal system.

In Figure 4 you can clearly see the top MicroBlaze block with its own bus and peripheral set separate from the bottom MicroBlaze block and its own, different peripheral set. The top section of the design runs embedded Linux as an OS with full file system support, enabling access to MP3 bitstreams from a network. We offloaded the decoding and playing of

these bitstreams to a second MicroBlaze processor design, where we added tightly coupled processor offload engines for the DCT/IMDCT (forward and inverse modified discrete cosine transform) functions and two high-precision MAC units.

The IMDCT block covers data compression and decompression to reduce transmission-line execution time. DCT/IMDCT are two of the most computationally intense functions in compression applications, so moving this whole function to its own co-processing block greatly

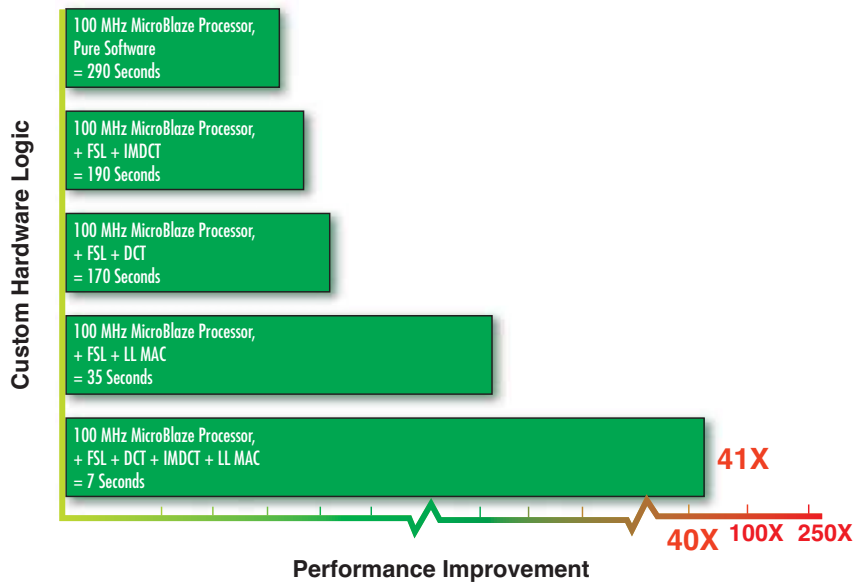


Figure 5 – Co-Processing acceleration results

improves overall system performance. Whereas we implemented an internal FPU in the earlier FIR filter example, the MP3 example has implemented MicroBlaze customizations and added external dedicated hardware within the FPGA.

Co-Processing + Customizable IP = Performance

By offloading computationally intense software functions to co-processing “hard instructions,” you can develop an optimal balance for maximizing system performance. Figure 4 also shows a number of IP peripherals for the Linux file system module, including UART, Ethernet MAC, and many other memory controller options. The coder/decoder application block, by comparison, uses different IP customizable for different system capabilities.

The second MicroBlaze soft core is slaved to the first MicroBlaze processor and acts as a task engine decoding the MP3 bitstream. The decoder algorithm with the addition of specific IP cores is connected directly in the FPGA fabric hardware resources through a Xilinx Fast Simplex Link (FSL) connection interface. This co-processing design technique takes advantage of the parallel and high-speed nature of FPGA hardware compared to the slower, sequential execution of instructions of a stand-alone processor.

Direct linkage to the high-performance FPGA fabric introduces fast multiply accumulate modules (LL_SH MAC1 and LL_SH MAC2 on Figure 4) to complement dedicated IP for the DCT and IMDCT blocks. The long-long MAC modules provide higher precision while offloading the processing unit. You will also notice high-speed FSL links utilized for the AC97 controller core to interface to an external AC 97 codec, allowing the implementation of CD-quality audio input/output for the MP3 player.

The co-processing system depicted in Figure 4 results in a cumulative 41x performance acceleration over the original software application with the additive series of component boosts. Comparing a pure “software-only” implementation (see the top horizontal bar as illustrated in Figure 5) to each subsequent stage of the hardware instruction instantiations, you can see how the performance improvements add up. Moving software into the IMDCT alone yields a 1.5x improvement, while adding the DCT as a hardware instruction moves it up to 1.7x. Another larger improvement is realized by implementing a long-long multiply accumulate to reach 8.2x.

Implementing all of the software modules in hardware through co-processing techniques, the total end result rolls up to

yield an amazing 41x improvement – with the added benefit of reducing the application code size. Because we removed multiple functions requiring a large number of instructions and replaced them with a single instruction to read or write the FSL port, we have far fewer instructions and thus achieved some code compaction. In the MP3 application section, for example, we saw a 20% reduction in the code footprint.

Best of all, design changes through intelligent tools like Platform Studio are easy, quick, and can still be implemented well into the product development cycle. Software-only methods of performance improvement are time-consuming and usually have a limited return on investment. Balancing the partition of software application, hardware implementation, and co-processing in a programmable platform, you can attain much more optimal results.

Conclusion

Based on the examples described in this article, we were able to easily customize a full embedded processing system, edit the IP for the optimal balance of feature/size/cost, and additionally squeeze out huge performance gains where none appeared possible. The Virtex-4 and Spartan-3 device families offer flexible soft-processor solution options that can be designed and refined late into the development cycle. The award-winning MicoBlaze soft-processor core combined with the award-winning Platform Studio tool suite provides a powerful combination to kick-start your embedded designs.

Co-processing techniques, such as implementing computationally intense software algorithms as high-performance FPGA hardware instructions, allow you to accelerate your performance of modules by 2x, 10x, or as much as 40x+ in our common industry example. Imagine what it can do for your next design – think about the head room and flexibility available for your design late in the development cycle, or being able to proactively plan improvements to the next generation of your product.

For more information on Xilinx embedded processing solutions, visit www.xilinx.com/processor.

Tracing Your Way to Better Embedded Software

Techniques to isolate and debug critical software issues.

by Jay Gould
Product Manager
Xilinx, Inc.
jay.gould@xilinx.com

Steve Veneman
Product Manager
Wind River Systems
steven.veneman@windriver.com

Have you ever worked on an embedded processing project where your team invested an exorbitant amount of test time only to find out at the end of the project that your product had a series of intermittent and hard-to-find bugs? You used your software debugger throughout the development process and spent man-weeks stepping through the code in your lab, yet when your software passed integration testing you were stunned to find that your QA team (or worse yet, a customer) discovered serious system flaws.

Some embedded bugs are harder to find than others and require advanced detection methods. Testing small units of code in

your office or lab does not fully exercise your embedded product the same way as when it is fully assembled and deployed in real-time conditions. Add other modules from other engineers and your simple unit tests may still pass for hours at a time. Often bugs will not reveal themselves until you run the code for much longer periods of time or with more complex interactions with other code modules.

So how do you know if you are really testing all your code? Is it possible that a colleague's software may be overwriting a variable or memory address utilized in your module?

The more complex your embedded products become, the more sophisticated your development and debugging tools must scale. Often "test time" is equated with "good testing" practices: the longer you run tests, the better you must be exercising the code. But this is often misleading for a number of reasons. Stubbing code and testing a module in a stand-alone method will miss many of the interaction mistakes of the rest of the final

system. Running dozens or hundreds of "use cases" may seem powerful, but may create a false sense of security if you don't actually track metrics like code "coverage," where you pay more attention to exercising all of the code instead of repeatedly testing a smaller subset.

Standard software debuggers are a useful and much-needed tool in the embedded development process, especially in the early stages of starting, stepping, and stopping your way through brand-new code units. Once you have significant code applications developed, testing the system as a whole becomes much more representative of an end-user product, with all of the interactions of a real-time embedded system. However, even a good software debugger may not improve your chances of finding an intermittent, seldom-occurring bug. Serious code interaction – made more complex with asynchronous interrupts of real-world embedded devices – mandates the use of proper tools to observe the entire system, creating a more robust validation environment in which to find the sneakier bugs.

Trace Your Code Execution

Once your code goes awry, locks up your system, core dumps, or does something seriously unexpected, it is time to improve the visibility of exactly what your software was doing before the flaw. Often the symptom of the failure is not clearly tied to the cause at all. Some debuggers or target tools running on the embedded device will die with a fatal system flaw, so they cannot be used to track that flaw. In these cases, you can use a separate and external tool with its own connections and hardware memory to provide an excellent view of system execution all the way up to a complete failure. “Trace” tools, often associated with JTAG probes or ICEs (in-circuit emulators), can display an abundance of useful information for looking at software instructions executed, lines of code executed, and even performance profiling.

Capturing software execution with a trace probe allows more debugging capability than just examining a section of code. An intelligent trace tool provides a wide variety of trace and event filters, as well as trigger and capture conditions. One of the most useful situations for trace techniques is to identify defects that cause target crashes on a random, unpredictable basis.

Connecting a probe (Figure 1) like Wind River ICE with the integrated Wind River Trace tool introduces a method by which you can configure trace to run until the target crashes. By triggering on “no trigger,” the trace will continue to capture in a circular buffer of memory in the probe until the system crashes. You may have unsuccessfully tried to manually find a randomly occurring defect with your software debugger, but with a smart trace tool, you can now run the full battery of system tests and walk away to let the trace run. When that intermittent event finally occurs again, crashing the target, the tool will capture and upload the data for your examination.

The execution trace will show a deep history of software instructions executed on your system leading up to the fatal flaw, providing much more insight into what the system was doing and allowing you to find and fix the bug faster. Often these prob-

lems are not easily found by post-mortem debugging through register and memory accesses. Embedded systems commonly crash when a function is overwriting a variable that should not be accessed/written, or when memory is corrupted by executable code being overwritten by another routine. It is difficult to diagnose these types of problems when no information exists on what actions led up to an event, and the crash of the system may not actually take place until much later when the corrupted

information is next accessed.

With advanced tools like Wind River Trace, you can capture the actual instruction and data information that was executed on the target before the event occurred. Using this information, you can back up through the executed code and identify what events happened before the crash, identifying the root cause of the problem (see Figure 2). In addition to a chronologically accurate instruction sequence, when used on a PowerPC™ embedded platform like Xilinx® Virtex™-II Pro or Virtex-4 devices, the Wind River Trace execution information also includes specific time-stamp metrics. Coincidentally, the Wind River ICE actually implements Xilinx Virtex-II Pro FPGA devices in the probe hardware itself.

Coverage Analysis and Performance Profiling

Because the trace tool and probe are able to provide a history of the software execution, they can tell you exactly where you have been – and what code you have missed. You miss 100% of the bugs that you don’t look for, so collecting coverage analysis



Figure 1 – The Wind River ICE JTAG probe implements Virtex FPGAs.

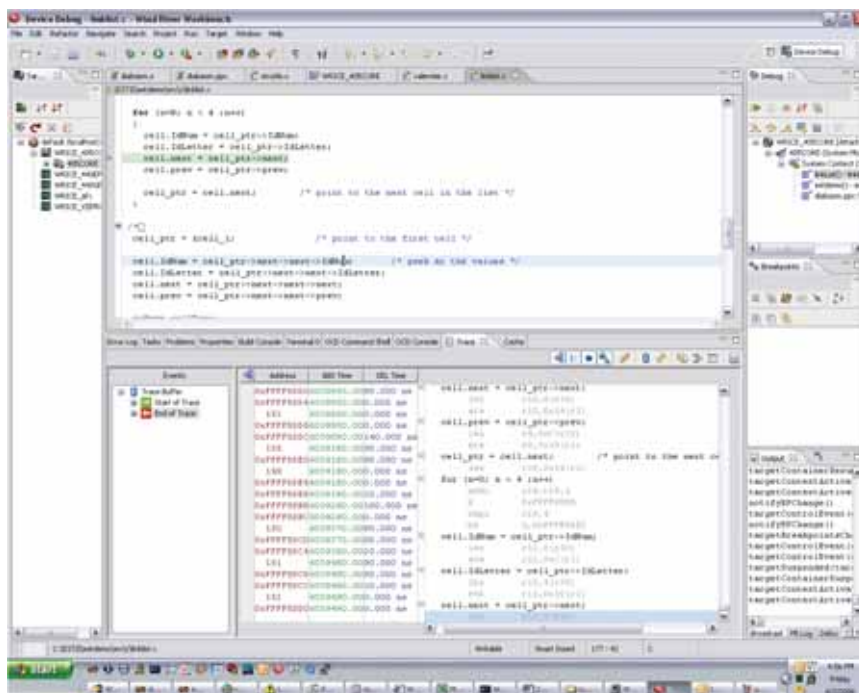


Figure 2 – Wind River Trace software execution view

metrics is important to your testing strategy. As code execution is almost invisible without an accurate trace tool, it is common for entire blocks or modules of code to go unexecuted during test routines or redundant user-case suites. Testing the same function 10 times over may make sense in many cases, but missing a different

space, higher than normal software testing standards may be mandated. Certainly the software running the ABS brakes in your car, a medical device, or an aircraft control system should be tested to a higher degree than a software game or other non-life-threatening application. In these life-critical applications code must be executed and

built into probes and trace tools allow you to look at the data in a performance view in addition to the go/no-go code coverage view. By identifying bottlenecks in software execution, you can identify routines that fail to meet critical timing specifications or optimize routines for better performance. With the time-stamp information in Virtex-II Pro and Virtex-4 devices, you can also use trace tools to determine how long a software function or a particular section of code took to run.

Whether you are trying to find routines that could use some optimization recoding (perhaps in-lining code blocks rather than accruing overhead by repeatedly calling a separate function) or identifying a timing flaw in a time-critical routine, performance-metric tools are required to collect multiple measurements. Settling for a couple passes of a routine with a manual debugger is not sufficient for validating timing specs on important code blocks. Additionally, if a real-time system is running an embedded operating system, verifying the capability of the interrupt service routines (ISRs) is essential to determine how quickly the system responds to interrupts and how much time it spends handling them.

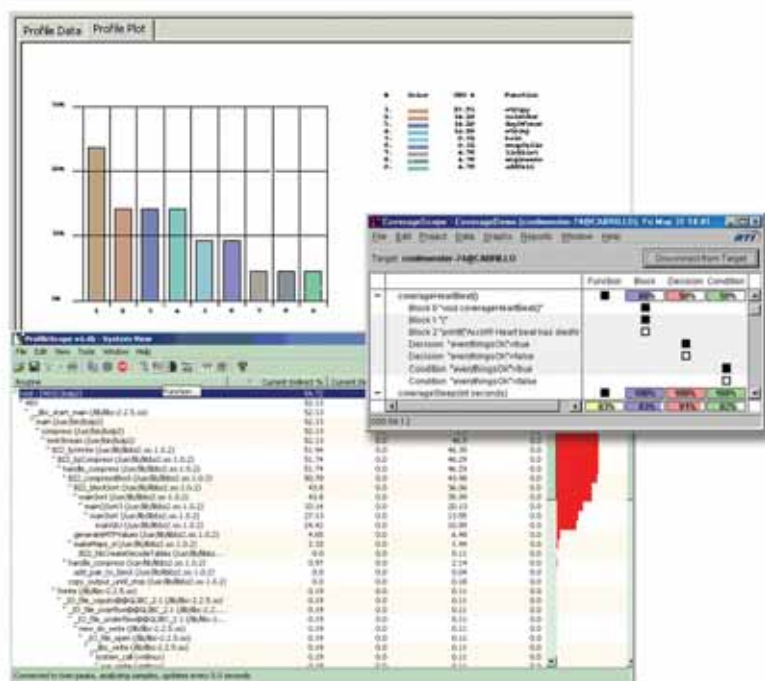


Figure 3 – Wind River profiling and coverage views

function altogether means that you will not find those bugs in your lab – your customer will find them in the field.

Coverage metrics showing which functions are not executed are useful for writing new, additional tests or identifying unused “dead” code, perhaps leftover from a previous legacy project. Because many designs are re-ports or updates of previous products, many old software routines become unnecessary in a new design, but the legacy code may be overlooked and not removed. In applications where code size is critical, removing dead code reduces both waste as well as risk; nobody wants to wander into an obsolete software routine that may not even interact properly with a newer version of hardware.

In safety-critical applications like avionics, medical, automotive, or defense/aero-

tested (covered) down into every function of software instructions, and that execution may need to be documented to a formal, higher level governing body.

In many cases code coverage can also be used to analyze errant behavior and the unexpected execution of specific branches or paths. Erratic performance, failure to complete a task, or not executing a routine in a repeatable fashion may be just as inappropriate in some applications as a fatal crash or other defect. By tracing function entries and exits, comprehensive data is collected on software operations and you can throw away the highlighter and program listing (Figure 3).

Because performance is always a concern in modern embedded applications, having an accurate way to measure execution times is paramount. The capabilities

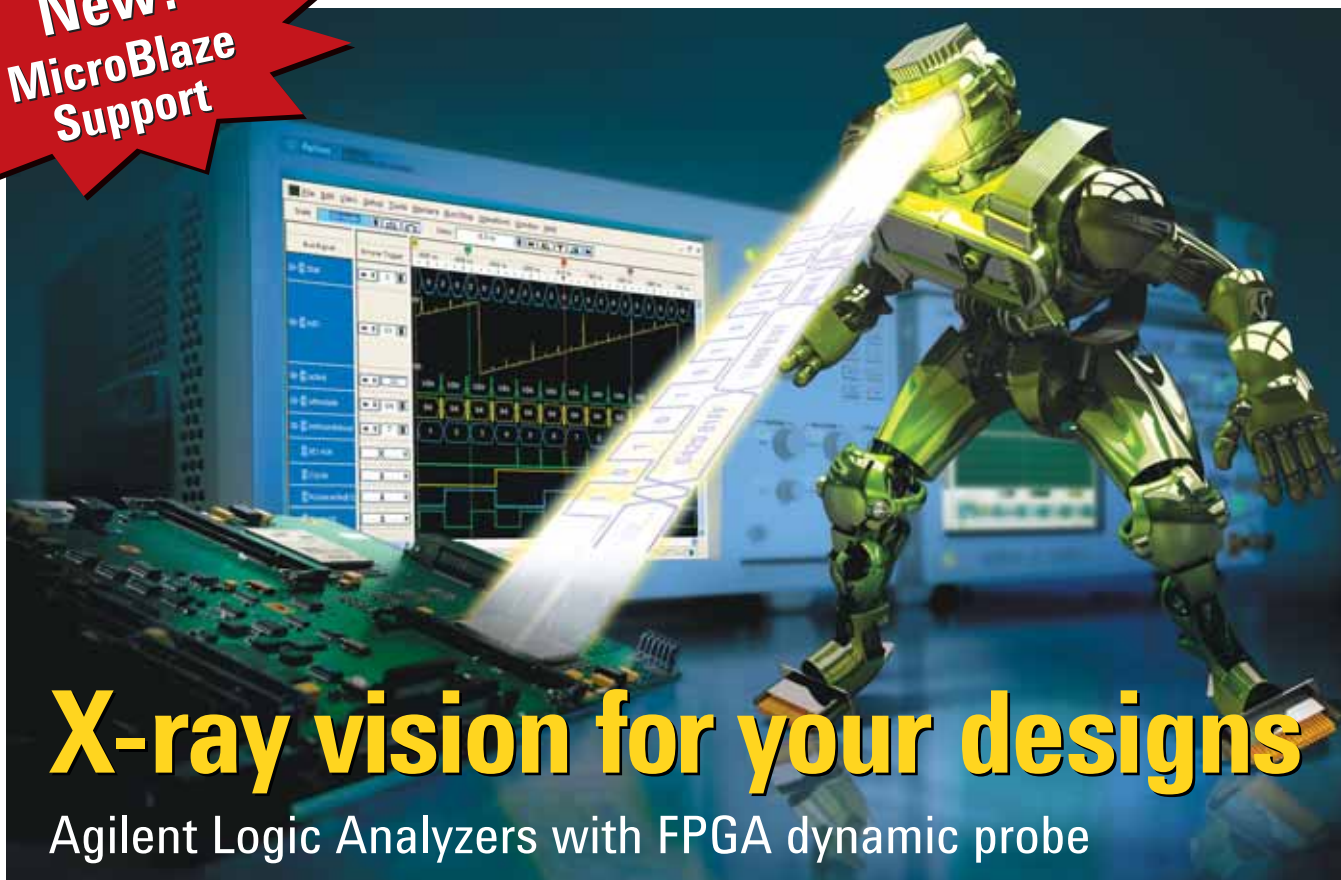
Conclusion

If critical performance measurements or code coverage are required before your embedded product can ship, then you need to use specialized tools to capture and log appropriate metrics. Accurate system trace metrics are a must if you have ever been challenged by a seemingly random, hard-to-reproduce bug that your customers seem to experience regularly but that you cannot duplicate in your office.

Utilizing an accurate hardware probe like Wind River Trace with a powerful trace/coverage/performance tool suite will provide you with the data you require. Stop guessing about what is really going on and gain new visibility into your embedded applications.

For more information about Wind River Trace and other products, visit www.windriver.com/products/development_suite/wind_river_trace/.

New!
MicroBlaze
Support



X-ray vision for your designs

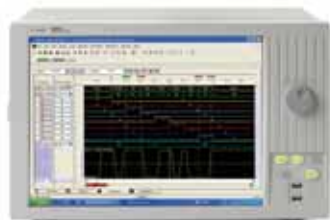
Agilent Logic Analyzers with FPGA dynamic probe

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 Windows-based logic analyzer, allows you to access different groups of signals inside your FPGA for debug—without requiring design changes. You'll increase visibility into internal FPGA activity by gaining access up to 64 internal signals with each debug pin.

Our newest 16800 Series logic analyzers offer you unprecedented price-performance in a portable family, with up to 204 channels, 32M memory depth and a pattern generator available.

Agilent's user interface makes our logic analyzers easy to get up and running. A touch screen or mouse makes it simple to use, with prices to fit your budget. Optional soft touch connectorless probing solutions provide excellent reliability, convenience and the smallest probing footprint available. Contact Agilent Direct today to learn more.



- 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
- 16800 Series portable logic analyzers starting at \$9,430



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/logic

www.agilent.com/find/new16903quickquote



Agilent Technologies

dreams made real

To Interleave, or Not to Interleave: That is the Question

Controlling crosstalk in PCI Express and other RocketIO implementations.

by Patrick Carrier
Technical Marketing Engineer
Mentor Graphics Corporation
patrick_carrier@mentor.com

With the emergence of multi-gigabit-per-second SERDES bus architectures such as PCI Express, Xilinx® RocketIO™ technology has become a vital part of FPGA designs. But accompanying these faster buses is a resurgence of an old design problem: crosstalk. The source of concern is similar to what occurs in buses like PCI and PCI-X: a large number of signals all need to go to the same place, from the FPGA to a connector or to another chip.

The bad news is that PCI Express edge rates are much faster than PCI and PCI-X. The fastest edges can be on the order of 50 ps, which equates to around a third of an inch on a printed circuit board. This equates to more crosstalk for a given amount of parallelism, which can be quite large for typical PCI Express link lengths.

The good news is that because PCI Express links comprise unidirectional differential pairs, you only need to control crosstalk at the receiver. This characteristic of PCI Express has actually led to two theories on how to implement board routing: interleaving TX and RX pairs or keeping like pairs together. There is no one correct answer to this design question; the answer depends on the characteristics of the design.

FEXT, NEXT, and Interleaving

In order to determine the best method for routing adjacent PCI Express differential pairs, you should understand the nature of both forward and reverse crosstalk. Both types of crosstalk are caused by coupling between traces; that coupling includes mutual inductance as well as mutual capacitance.

In forward crosstalk, also referred to as far-end crosstalk or FEXT, coupled energy propagates onto the “victim” signal in the same direction as the “aggressor” signal. As such, the FEXT pulse has an edge equivalent in length to the aggressor signal and continues growing in amplitude as the signals propagate down the line. The amplitude of FEXT is thus determined by the

length of parallelism between adjacent traces and the amount of coupling, as well as the balance between the inductive and capacitive components of coupling.

In reverse crosstalk, also known as near-end crosstalk or NEXT, the coupled energy propagates in the opposite direction of the aggressor signal. As such, its amplitude is also length-dependent, but saturates for any coupled length greater than the signal edge rate. In NEXT, the inductive and capacitive coupling are additive.

Figure 1 shows examples of these two types of crosstalk in the Mentor Graphics HyperLynx oscilloscope. Notice how the forward crosstalk (shown in light blue) is a large spike whose leading edge has an edge rate equivalent to that of the aggressor signal (in this case 100 ps). The reverse crosstalk (shown in yellow) is much smaller in amplitude, with a width equal to twice the total line length (in this case, 5 inches or 712 ps). The topology used to generate these waveforms in HyperLynx LineSim is depicted in Figure 2.

You can reduce the coupling between traces most effectively by increasing the spacing between them. Because PCI Express comprises unidirectional differential pairs, you can further control crosstalk by altering the aggressors' direction to allow for only reverse crosstalk or only forward crosstalk.

For trace configurations where the forward crosstalk exceeds the reverse crosstalk, the preferred method of routing differential pairs would be to interleave them. If a TX signal is placed adjacent to an RX signal, the forward crosstalk created by that signal will go towards the transmitter of the RX signal, where it is not of concern. The reverse crosstalk created by the TX signal will go towards the receiver of the RX signal.

Conversely, if the reverse crosstalk exceeds the forward crosstalk, non-interleaved routing would be preferable; all forward crosstalk would be directed towards the victims' receivers instead of the reverse crosstalk.

Microstrip and Stripline Crosstalk Analysis

To determine when each method of routing is appropriate, let's examine both inner and outer layer routing in simulation with a topology similar to that shown in Figure 2. In the first example, a set of three microstrip differ-

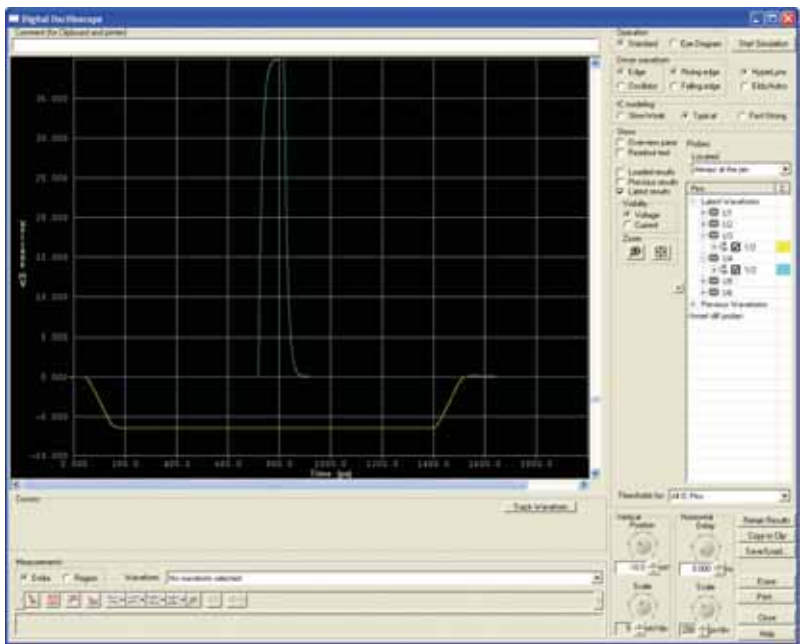


Figure 1 – Forward and reverse crosstalk

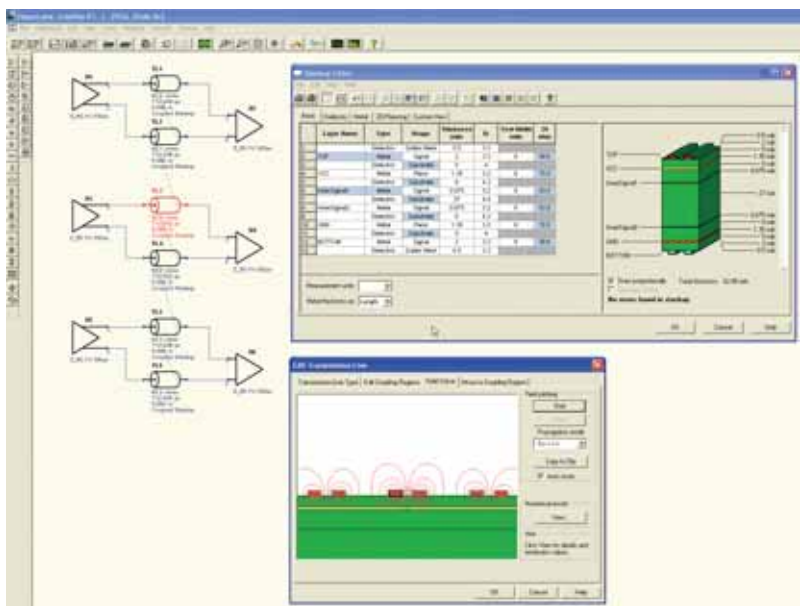


Figure 2 – Topology used for crosstalk analysis

ential pairs, with 5-mil wide copper traces 2 mils thick separated by 5.7 mils, sitting on a 5-mil dielectric with an $\epsilon_r = 4$, covered with a 0.5-mil solder mask with an $\epsilon_r = 3.3$, have their spacing and lengths varied in simulation to obtain the information shown in Table 1.

In microstrip routing, far-end crosstalk dominates, except in the very rare case of super-short routing and super-tight spacing. As such, it would be reasonable to

make a general rule that PCI Express pairs should be routed as interleaved when being routed on microstrip.

In the second example, three stripline (inner-layer) differential pairs, with 4-mil wide copper traces 2 mils thick separated by 5.0 mils, in a dual-stripline configuration with dielectric heights of 4, 20, and 4 mils, all modeled with an $\epsilon_r = 3.5$, have their spacing and lengths varied in simulation to obtain the information shown in Table 2.

Notice that the far-end crosstalk is 0 mV. This is because of the balance that exists between the counteracting capacitive and inductive couplings on the homogenous stripline. This might lead you to believe that non-interleaved routing should always be used on the stripline. However, that would be an erroneous assumption. Homogenous stripline rarely exists on a real printed circuit board. There are differences in dielectric constants between cores and prepregs, the two types of dielectrics that make up a stripline. Furthermore, there are resin-rich areas localized to the traces that alter the dielectric constant around the traces. To properly determine whether or not to interleave stripline traces, you must appropriately model the dielectric constants.

The third example uses the same dual stripline trace configuration, except that the two 4-mil dielectrics are assigned an $\epsilon_r = 3.5$, the 20-mil dielectric an $\epsilon_r = 4.5$, and the thin layer of dielectric on the signal layer is given an $\epsilon_r = 3.2$. Differential pair spacings and lengths are varied to produce the results shown in Table 3.

From this data, it is evident that on longer buses, or buses with longer amounts of parallelism, you should use interleaved routing because far-end crosstalk dominates. Near-end crosstalk dominates for shorter routes with tighter spacings. The point at which the far-end crosstalk exceeds the near-end crosstalk varies based on length, spacing, and trace configuration. That crossover point is what you should use to determine whether or not to interleave TX and RX differential pairs.

Conclusion

Clearly, the majority of board routing implementations for PCI Express and similar RocketIO implementations requires that you interleave TX and RX differential pairs in the layout. This is certainly true for microstrip, where forward crosstalk dominates. It is also true for stripline, with the exception of tighter spacing with shorter lengths. That is because forward crosstalk is non-zero in an actual stripline, and will in many cases exceed reverse crosstalk. Simulation, with proper modeling of the stripline dielectrics, reveals this phenomenon, while pinpointing the trace configuration where the FEXT will exceed the NEXT and indicate whether or not to interleave.

To learn more about designing with RocketIO technology, visit www.xilinx.com/serialdesign and www.mentor.com/highspeed.

	1H	2H	3H	5H
1 in.	NEXT = 75 mV, FEXT = 25 mV	NEXT = 27.9 mV, FEXT = 18 mV	NEXT = 12.7 mV, FEXT = 12 mV	NEXT = 3.7 mV, FEXT = 5.4 mV
2 in.	NEXT = 75 mV, FEXT = 47 mV	NEXT = 27.9 mV, FEXT = 37.8 mV	NEXT = 12.7 mV, FEXT = 24 mV	NEXT = 3.7 mV, FEXT = 10.5 mV
5 in.	NEXT = 75 mV, FEXT = 125 mV	NEXT = 27.9 mV, FEXT = 94 mV	NEXT = 12.7 mV, FEXT = 62 mV	NEXT = 3.7 mV, FEXT = 26 mV
10 in.	NEXT = 75 mV, FEXT = 255 mV	NEXT = 27.9 mV, FEXT = 192 mV	NEXT = 12.7 mV, FEXT = 126 mV	NEXT = 3.7 mV, FEXT = 53 mV
20 in.	NEXT = 75 mV, FEXT = 508 mV	NEXT = 27.9 mV, FEXT = 381 mV	NEXT = 12.7 mV, FEXT = 254 mV	NEXT = 3.7 mV, FEXT = 105 mV
30 in.	NEXT = 75 mV, FEXT = 625 mV	NEXT = 27.9 mV, FEXT = 555 mV	NEXT = 12.7 mV, FEXT = 379 mV	NEXT = 3.7 mV, FEXT = 158 mV

Table 1 – Crosstalk in terms of differential pair spacing versus trace length for a microstrip configuration

	1H	2H	3H	5H
1 in.	NEXT = 72.1 mV, FEXT = 0 mV	NEXT = 29.9 mV, FEXT = 0 mV	NEXT = 14.5 mV, FEXT = 0 mV	NEXT = 4.5 mV, FEXT = 0 mV
2 in.	NEXT = 72.1 mV, FEXT = 0 mV	NEXT = 29.9 mV, FEXT = 0 mV	NEXT = 14.5 mV, FEXT = 0 mV	NEXT = 4.5 mV, FEXT = 0 mV
5 in.	NEXT = 72.1 mV, FEXT = 0 mV	NEXT = 29.9 mV, FEXT = 0 mV	NEXT = 14.5 mV, FEXT = 0 mV	NEXT = 4.5 mV, FEXT = 0 mV
10 in.	NEXT = 72.1 mV, FEXT = 0 mV	NEXT = 29.9 mV, FEXT = 0 mV	NEXT = 14.5 mV, FEXT = 0 mV	NEXT = 4.5 mV, FEXT = 0 mV
20 in.	NEXT = 72.1 mV, FEXT = 0 mV	NEXT = 29.9 mV, FEXT = 0 mV	NEXT = 14.5 mV, FEXT = 0 mV	NEXT = 4.5 mV, FEXT = 0 mV
30 in.	NEXT = 72.1 mV, FEXT = 0 mV	NEXT = 29.9 mV, FEXT = 0 mV	NEXT = 14.5 mV, FEXT = 0 mV	NEXT = 4.5 mV, FEXT = 0 mV

Table 2 – Crosstalk in terms of differential pair spacing versus trace length for a homogenous stripline configuration

	1H	2H	3H	5H
1 in.	NEXT = 78.4 mV, FEXT = 5.2 mV	NEXT = 34 mV, FEXT = 5.2 mV	NEXT = 17.2 mV, FEXT = 3.5 mV	NEXT = 5.6 mV, FEXT = 3.6 mV
2 in.	NEXT = 78.4 mV, FEXT = 9.5 mV	NEXT = 34 mV, FEXT = 10.1 mV	NEXT = 17.2 mV, FEXT = 7 mV	NEXT = 5.6 mV, FEXT = 3.6 mV
5 in.	NEXT = 78.4 mV, FEXT = 18.2 mV	NEXT = 34 mV, FEXT = 21.5 mV	NEXT = 17.2 mV, FEXT = 17.4 mV	NEXT = 5.6 mV, FEXT = 10.8 mV
10 in.	NEXT = 78.4 mV, FEXT = 34.8 mV	NEXT = 34 mV, FEXT = 42.8 mV	NEXT = 17.2 mV, FEXT = 34.6 mV	NEXT = 5.6 mV, FEXT = 19.7 mV
20 in.	NEXT = 78.4 mV, FEXT = 67.1 mV	NEXT = 34 mV, FEXT = 82.1 mV	NEXT = 17.2 mV, FEXT = 70.5 mV	NEXT = 5.6 mV, FEXT = 39.2 mV
30 in.	NEXT = 78.4 mV, FEXT = 101.4 mV	NEXT = 34 mV, FEXT = 124.8 mV	NEXT = 17.2 mV, FEXT = 104 mV	NEXT = 5.6 mV, FEXT = 58.6 mV

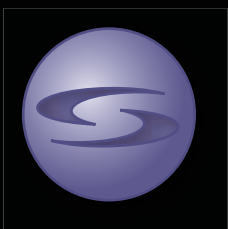
Table 3 – Crosstalk in terms of differential pair spacing versus trace length for a realistic stripline configuration

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

Great Test, Less Filling

SystemBIST™ enables FPGA Configuration that is less filling for your PCB area, less filling for your BOM budget and less filling for your prototype schedule. All the things you want less of and more of the things you do want for your PCB – like embedded JTAG tests and CPLD reconfiguration.

Typical FPGA configuration devices blindly “throw bits” at your FPGAs at power-up. SystemBIST is different – so different it has three US patents granted and more pending. SystemBIST’s associated software tools enable you to develop a complex power-up FPGA strategy and validate it. Using an interactive GUI, you determine what SystemBIST does in the event of a failure, what to program into the FPGA when that daughterboard is missing, or which FPGA bitstreams should be locked from further updates. You can easily add PCB 1149.1/JTAG tests to lower your down-stream production costs and enable in-the-field self-test. Some capabilities:

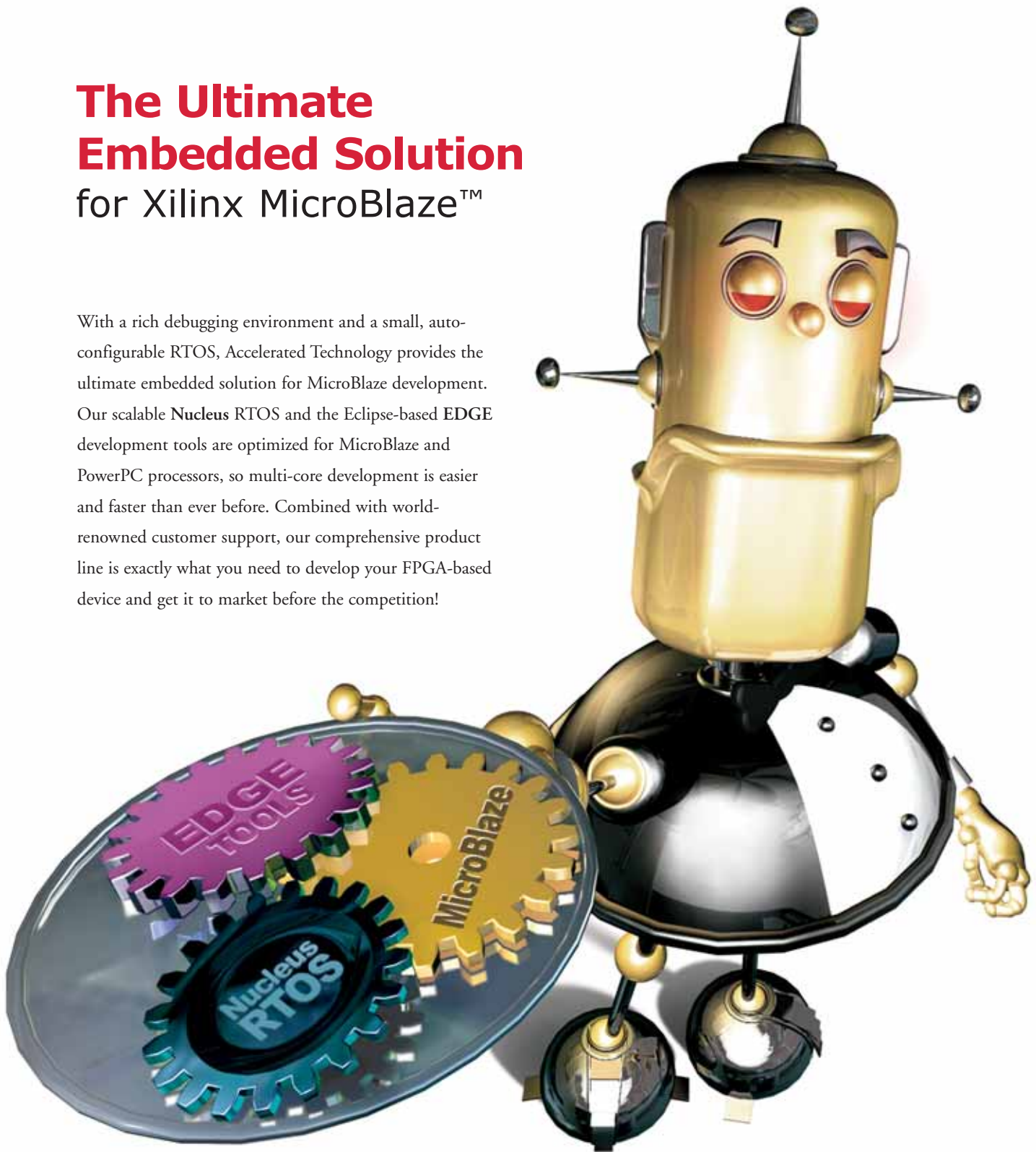
- User defined FPGA configuration/CPLD re-configuration
- Run Anytime-Anywhere embedded JTAG tests
- Add new FPGA designs to your products in the field
- “Failsafe” configuration – in the field FPGA updates without risk
- Small memory footprint offers lowest cost per bit FPGA configuration
- Smaller PCB real-estate, lower parts cost compared to other methods
- Industry proven software tools enable you to get-it-right before you embed
- FLASH memory locking and fast re-programming
- New: At-speed DDR and RocketIO™ MGT tests for V4/V2

If your design team is using PROMS, CPLD & FLASH or CPU and in-house software to configure FPGAs please visit our website at <http://www.intellitech.com/xcell.asp> to learn more.



The Ultimate Embedded Solution for Xilinx MicroBlaze™

With a rich debugging environment and a small, auto-configurable RTOS, Accelerated Technology provides the ultimate embedded solution for MicroBlaze development. Our scalable Nucleus RTOS and the Eclipse-based EDGE development tools are optimized for MicroBlaze and PowerPC processors, so multi-core development is easier and faster than ever before. Combined with world-renowned customer support, our comprehensive product line is exactly what you need to develop your FPGA-based device and get it to market before the competition!



For more information on the Nucleus complete solution, go to
www.acceleratedtechnology.com/xilinx

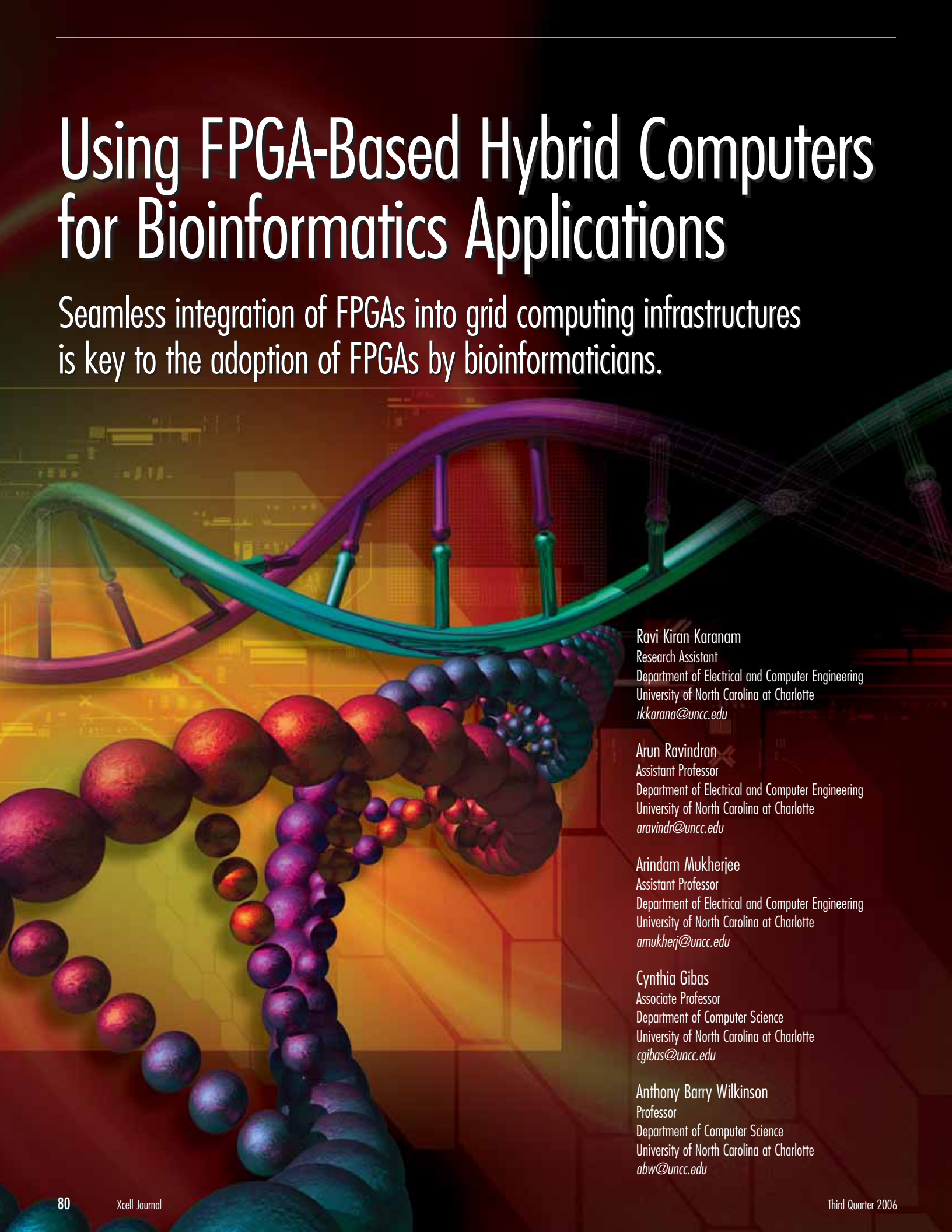
Accelerated Technology, A Mentor Graphics Division
info@AcceleratedTechnology.com • www.AcceleratedTechnology.com

©2006 Mentor Graphics Corporation. All Rights Reserved. Mentor Graphics, Accelerated Technology, Nucleus is a registered trademarks of Mentor Graphics Corporation. All other trademarks and registered trademarks are property of their respective owners.

**Accelerated
Technology®**
A Mentor Graphics Division

Using FPGA-Based Hybrid Computers for Bioinformatics Applications

Seamless integration of FPGAs into grid computing infrastructures is key to the adoption of FPGAs by bioinformaticians.



Ravi Kiran Karanam
Research Assistant
Department of Electrical and Computer Engineering
University of North Carolina at Charlotte
rkkarana@uncc.edu

Arun Ravindran
Assistant Professor
Department of Electrical and Computer Engineering
University of North Carolina at Charlotte
aravindr@uncc.edu

Arindam Mukherjee
Assistant Professor
Department of Electrical and Computer Engineering
University of North Carolina at Charlotte
amukherj@uncc.edu

Cynthia Gibas
Associate Professor
Department of Computer Science
University of North Carolina at Charlotte
cgibas@uncc.edu

Anthony Barry Wilkinson
Professor
Department of Computer Science
University of North Carolina at Charlotte
abw@uncc.edu

The past decade has witnessed an explosive growth of data from fields in the biology domain, including genome sequencing and expression projects, proteomics, protein structure determination, and cellular regulatory mechanisms, as well as biomedical specialties that focus on the digitization and integration of patient information, test, and image records. Bioinformatics refers to the storage, analysis, and simulation of biological information and the prediction of experimental outcomes.

To address the computing and data management needs in bioinformatics, the traditional approach has been to use clusters of low-cost workstations capable of delivering gigaflops of computing power. However, microprocessors have general-purpose computing architectures, and are not necessarily well suited to deliver the teraflops of high-performance capability required for compute or data-intensive applications. The recent availability of off-the-shelf high-performance FPGAs – such as Xilinx® Virtex™-II Pro devices with on-board high capacity memory banks – has changed the computing paradigm by enabling high-speed processing and high-bandwidth memory access.

Nallatech offers multi-FPGA computing cards containing between one and seven Virtex FPGAs per card. Several such cards are plugged into the PCI slots of a desktop workstation and networked using Nallatech's DIMETalk networking software, greatly increasing the available computing capability. This is our concept of a hybrid computing platform. The hybrid platform comprises several workstations in a cluster that you can integrate with Nallatech multi-FPGA cards to construct a high-performance computing system.

Grid Computing

Grid computing is a form of distributed computing that employs geographically distributed and interconnected computing sites for high-performance computing and resource sharing. It promotes the establishment of so-called virtual organizations – teams of people from different organizations working together on a common goal, sharing computing resources and possibly experiment equipment.

In recent years, grid computing has become increasingly popular for tackling difficult bioinformatics problems. The rise of “bio-grids” (such as the NIH Cancer Biomedical Informatics Grid and Swiss Biogrid) is driven by the increasingly enormous datasets and computational complexity of the algorithms involved. Computational grids allow researchers to develop a bioinformatics workflow locally and then use the grid to identify and execute tasks seamlessly across diverse computing platforms.

To integrate the computing power of FPGA-based hybrid computing platforms with the existing computing infrastructure used by the bioinformatics community, we have grid-enabled the hybrid platform using the open-source Globus Toolkit. Thus, the hybrid platform and the associated software are available as a grid resource so that bioinformatics applications can be run over the grid. The hybrid platform partitions the tasks to be run on processors and FPGAs and uses application program interfaces (APIs) to transfer data to and from the FPGAs for accelerated computation. Bioinformaticians can now take advantage of the computing power of our FPGA hybrid computing platform in a transparent fashion.

Hybrid Computing Platform

The different FPGA related components in the hybrid platform are:

- High-capacity motherboard – BenNUEY
The Nallatech BenNUEY motherboard features a Xilinx Virtex-II Pro FPGA and module sites for as many as six additional FPGAs. The PCI/control and low-level drivers abstract the PCI interfacing, resulting in a simplified design process for designs/applications.
- Virtex-II expansion module – BenBlue-II
The Nallatech BenBlue-II DIME-II module provides a substantial logic resource ideal for implementing applications that have a large number of processing elements. Through support for as many as two on-board Xilinx Virtex-II Pro FPGAs (XC2VP100

devices), the BenBlue-II can provide more than 200,000 logic cells on a single module.

- Multi-FPGA management – DIMETalk
To manage the large silicon resource pool provided by the hardware, the Nallatech DIMETalk tool accelerates the design flow for creating a reconfigurable data network by providing a communications channel between FPGAs and the host user environment.
- FUSE Tcl/Tk control and C++ APIs
Nallatech's FUSE is a reconfigurable operating system that allows flexible and scalable control of the FPGA network directly from applications using the C++ development API, which is complemented by a Tcl/Tk toolset for scripting base control.

DNA Microarray Design – A Case Study

Our goal was to accelerate the Smith-Waterman implementation in the EMBOSS suite of publicly available bioinformatics code. The Smith-Waterman algorithm is widely used to screen gene databases for sequence similarity, with many different applications in bioinformatics research. Smith-Waterman is specifically used in situations where faster heuristic methods fail to detect some potentially meaningful sequence hits.

Dr. Cynthia Gibas of the Bioinformatics Center at UNC Charlotte currently uses water.c, the Smith-Waterman implementation in the open-source EMBOSS software, as a part of a DNA microarray design workflow. The biology goal is to select the optimal probe sequences to be printed on a DNA microarray, which will then be used in the lab to detect individual gene transcripts in a target mixture with high specificity.

Hardware/Software Partitioning

The EMBOSS implementation of the Smith-Waterman (water.c) is an extensive C program comprising more than 400 functions. A partitioning strategy is required to identify the functions that need to be implemented on the FPGA and those that remain in software (and run on the processor). The partition is done by profil-

Each sample counts as 0.01 seconds.						
Percentage Time	Cumulative Time (sec)	Self Time (sec)	Function Calls	ms/call (self)	ms/call (self)	Name of Function
83.32	178.94	178.94	32768	5.46	5.65	embAlignPathCalcSW
5.12	189.94	11.00	32768	0.34	0.35	embAlignWalkSWMatrix
4.62	199.87	9.93	32768	0.30	0.30	embAlignScoreSWMatrix
2.92	206.13	6.26	2719612928	0.00	0.00	ajSeqCvtK
1.86	210.13	4.00	53936723	0.00	0.00	match

Table 1 – Profiling results of the EMBOSS Smith-Waterman implementation

ing the execution characteristics of the code. First, the legacy C code in the water.c file is profiled using the Linux gprof tool. Profiling tells us where a given code spends time during execution, as well as different functional dependencies.

Table 1 shows the results of profiling in terms of the execution times of the top five functions executed by the code, listed in decreasing order of execution time. Note that one function, embAlignPathCalcSW, accounts for 83% of the total amount of program execution time. The embAlignPathCalcSW function uses the Smith-Waterman-based local alignment algorithm to create a “path matrix” containing local alignment scores of comparing probe and database sequences at different matrix locations, and a compass variable to show which partial result is used to compute the score at a certain location.

Once the code profiling is done, the computationally intense embAlignPathCalcSW call is mapped to the FPGA network using VHDL, while the rest of the code is run on the processor. Calls to the computationally intense embAlignPathCalcSW function in the C code of the water.c file are then replaced with corresponding application program interface (API) calls to the FPGA network. These APIs transfer data between the FPGA network and the processor, such that the calculation of the scores in the path matrix is done inside the FPGA network. All other parts of the code, including backtracking, are executed on the processor

in software. A flow diagram of these steps is shown in Figure 1.

Hardware Implementation

When comparing a target sequence (T) from a database with a probe sequence (P), the scores and the compass values of the path matrix in the embAlignPathCalcSW function are calculated using a systolic array of basic processing elements (PEs).

Figure 2 shows the systolic array implementation of the path matrix algorithm for Smith-Waterman with three PEs. Each PE passes the score and compass values it calculates to the successive PE, which in turn uses these values to calculate its path and compass values. At each clock cycle the path and the compass values are stored in the block RAM. At the end of the computation, each block RAM has the corresponding row of the path matrix stored in it.

The output of the computationally intensive function involves huge amounts of data transfer (the order of probe length times the target length). As a result, the improvements achieved in computing performance are compromised by communication latency. To decrease communication latency, two additional functions (embAlignScoreCalcSW and embAlignWalkSWMatrix) were moved to the FPGA from software. These functions do the backtracking and calculate the score and alignment sequence for the given probe and target. The functions operate on the path matrix and calculate the maximum path value. Then, starting at the location of the maximum path value, the functions back-

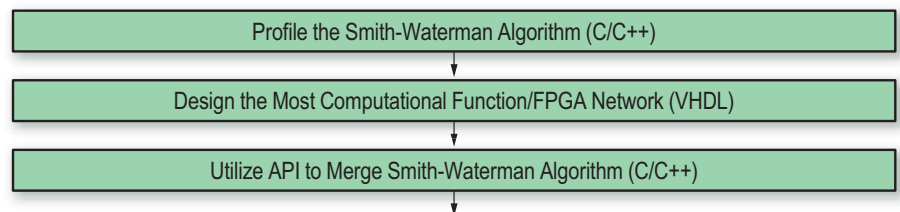


Figure 1 – Design flow for accelerating water.c on the FPGA-processor hybrid computer

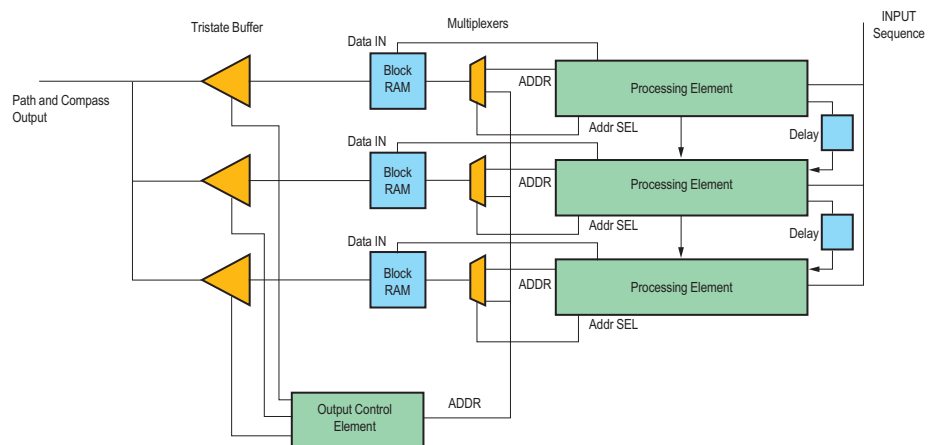


Figure 2 – Systolic array of three PEs for the path matrix calculation

track through the path values in the path matrix based on the compass values to determine the best alignment. The output of the FPGA is now the score and the alignment sequence, which is only about the size of the probe sequence, thus greatly reducing communication latency.

Grid-Enabling the Hybrid Computing Platform

The first step to grid-enable our resource was to install the components of the grid software utility package called Globus Toolkit 4.0 (GT4). The GT4 is an open-source reference implementation of key grid protocols. The toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability.

The core of GT4, the open grid services architecture (OGSA), is an integration of grid technologies and Web service technologies. Above the OGSA, the job submission and control is performed by the grid resource allocation and management (GRAM). GRAM provides a single interface for requesting and using remote system resources for the execution of "jobs." The Globus security interface enables the authentication and authorization of users and groups in the grid through the exchange of signed certificates. Certificates are created and maintained by a tool called SimpleCA in GT4.

Installation and Integration with the UNC-Charlotte VisualGrid

To test our installation in a real grid, we took advantage of a new grid project called VisualGrid, a collaborative project between UNC-Charlotte, UNC-Asheville, and the Environmental Protection Agency. The FPGA-based hybrid computing platform is added as a client to VisualGrid through the master CA server.

Results

We implemented a prototype acceleration task comparing a 40-nucleotide probe sequence with target database sizes of as many as 1,048,576 targets of an approximate length of 850 nucleotides both in software (processor) and an FPGA hybrid computing platform. The probe sequence and the target

database reside in the main memory of the host computer (dual Opteron-based Sun Java W1100z workstation).

For each call to the `embAlignPathCalcSW`, `embAlignScoreCalcSW`, and `embAlignWalkSWMMatrix` functions from `water.c`, a 32-bit probe and target combination is sent over the PCI interface to the PCI FIFO on the Nallatech BenNUEY motherboard at a rate of 33 MHz. From the PCI FIFO, the data is transferred to a 32-bit, 512-word-long input FIFO on the Virtex-II

times of the two implementations. For small databases, the processing times of the processor and the hybrid computing platforms are comparable. However, as databases get larger, the processor computing time rises exponentially, while the hybrid computing platform shows a linear increase. For a database size of 1,048,576 strings, the hybrid computing platform is 44 times faster than execution on a processor. Such large databases are common in bioinformatics applications.

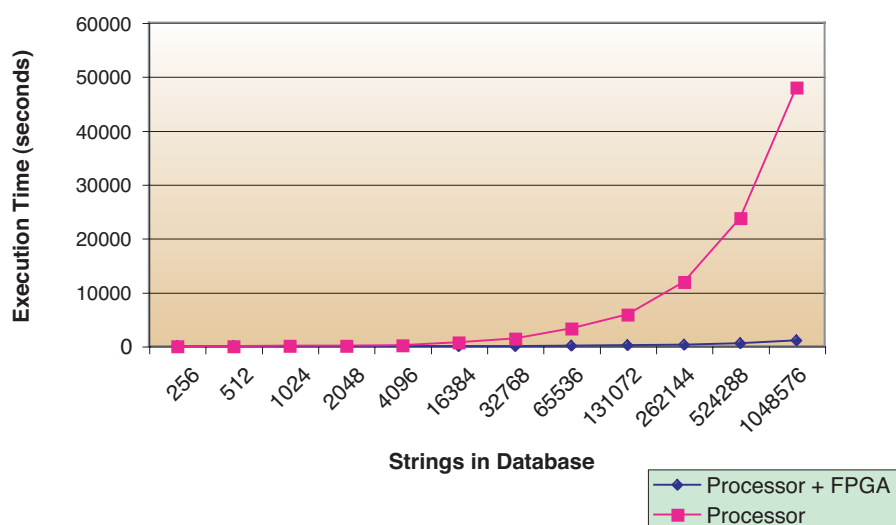


Figure 3 – Execution time for `water.c` for different sizes of sequence database strings running on a processor and a processor + FPGA hybrid system

Pro FPGA of the BenNUEY motherboard. The systolic array reads the data from this FIFO. The 40-processing-element-deep systolic array operates at 40 MHz.

After the systolic array completes processing on a single string, the output values from the block RAM are used to calculate the alignment. The alignment results are then written back to a different block RAM. Finally, the host processor reads the output alignment string from the block RAM over the PCI interface.

The hybrid computing platform was accessed through the VisualGrid with jobs submitted through the UNC-Charlotte VisualGrid Portal. The EMBOSS `water.c` program ran in software on the workstation and separately on the FPGA-based hybrid computing platform. Figure 3 shows the comparison between the run

Conclusion

We have demonstrated the computing potential of a grid-enabled hybrid computing platform for sequence-matching applications in bioinformatics. A careful partitioning of the computing task between the processor and the FPGAs on the hybrid platform circumvented the potential I/O bottleneck associated with large bioinformatics databases. The ability to submit jobs over the grids enables ready integration and use of the hybrid computing platforms in bioinformatics grids. We are currently involved in integrating the hybrid computing platform on the large inter-university SURAGrid (www.sura.org/SURAGrid). 🌈

This research was partly funded by NSF Award Number 0453916, and by Nallatech Ltd.

II DRR Delivers High-Performance SDR

Innovative Integration utilizes Virtex-II Pro FPGAs for a software-defined radio receiver system.

by Dan McLane
Vice President
Innovative Integration Inc.
dmclane@innovative-dsp.com

Amit Mane
Senior Engineer
Innovative Integration Inc.
amane@innovative-dsp.com

Pingli Kao
Senior Engineer
Innovative Integration Inc.
bkao@innovative-dsp.com

Innovative Integration (II) Inc. has developed a powerful solution for multi-channel software-defined radio (SDR). The II digital radio receiver (DRR) has a wide-bandwidth IF front-end digitizer integrated with Xilinx® Virtex™-II Pro FPGAs. This configuration allows you to realize the full flexibility of scalable programmable design with high-performance signal processing hardware.

Using MATLAB's Filter Design and Analysis tool (FDATool) and Innovative's FrameWork Logic software, you can easily design and optimize your desired filters.

You can customize and optimize the II DRR for multiple applications using the II SDR reference design. The reference design incorporates a MATLAB Simulink environment with Xilinx System Generator for DSP. In the System Generator tool, data is bit- and cycle-true and reflects the performance of the real system. You can easily modify the characteristics of the system by changing the parameters in the blocksets. You can then verify the blocksets in real-time simulations. Four Texas Instruments

the magnitude of the ripples produced by the CIC filter.

Figure 1 shows the entire SDR signal processing for the DDC. The channel filtering is built in the MATLAB Simulink environment using the SDR blockset. The input signal is tuned to the target frequency using a mixer and a direct digital synthesizer (DDS). The signal then flows through the CIC, CFIR, and PFIR.

You can implement the SDR directly in hardware by using the Xilinx System

When the specified filter is simulated, the passband ripple is within -0.8 dB, as shown in Figure 3a. Likewise, the magnitude is down to -40 dB at 0.544 MHz and below -90 dB after 1.365 MHz, as shown in Figure 3b.

The channel filter design verified in the simulation is bit- and cycle-true – so the DDC design matches the theoretical expected response shown in Figure 2.

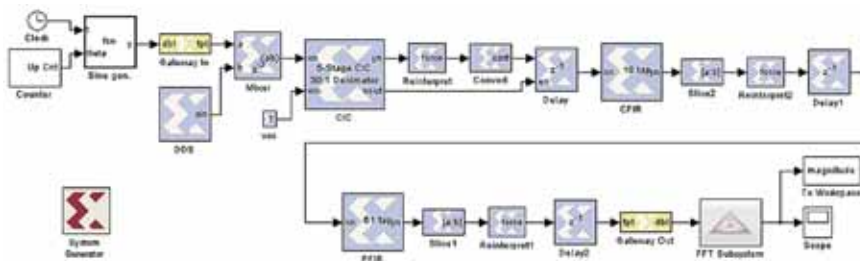


Figure 1 – Simulink block diagram of DDC using Xilinx System Generator software

(TI) 1 GHz TMS320C6416 DSPs supply ample calculating power for advanced operations such as signal encoding, decoding, and compression.

Anatomy of a Digital Down Converter

In the II DRR system, a digital down converter (DDC) decimates the RF to IF signal, providing signal compensation and shaping. The DDC comprises a cascaded integrator-comb (CIC) filter, a compensation filter (CFIR), and a programmable filter (PFIR).

The CIC filter is useful to realize large sample rate changes in digital systems. A CIC filter is a “multiplier-less” structure that comprises only adders, subtractors, and registers, which facilitates hardware implementation. The compensation filter flattens the passband frequency response. The programmable low-pass filter lowers

Generator tool to generate the signal processing logic that is fit into the Virtex-II Pro FPGA using Xilinx ISE™ software. The whole system can be designed and downloaded to hardware in hours, which effectively shortens time to market.

Designing Channel Filters

Using MATLAB's Filter Design and Analysis tool (FDATool) and Innovative's FrameWork Logic software, you can easily design and optimize your desired filters.

Consider a GSM system where the filter specification is:

$$Fs/2=32.5 \text{ MHz}, F_{pass}=0.49 \text{ MHz}, \\ F_{stop1}=0.542 \text{ MHz}, F_{stop2}=1.35 \text{ MHz}$$

The sampling frequency is 130 MHz and the decimation factor is 120. Figure 2 shows the theoretical system response from 0 Hz to 65 MHz.

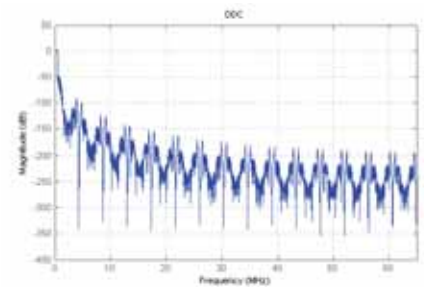


Figure 2 – Theoretical frequency response of DDC of R = 120 from FDATool program

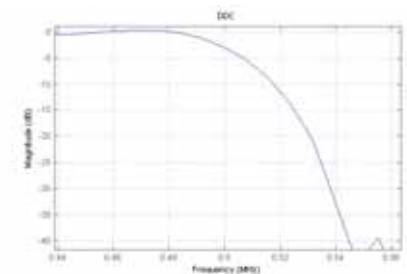


Figure 3(a) – Frequency response at the corner of $F_{pass} = 0.490 \text{ MHz}$

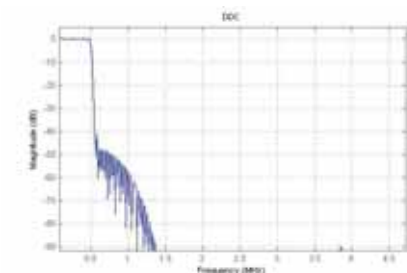


Figure 3(b) – Frequency response within 0.5 MHz

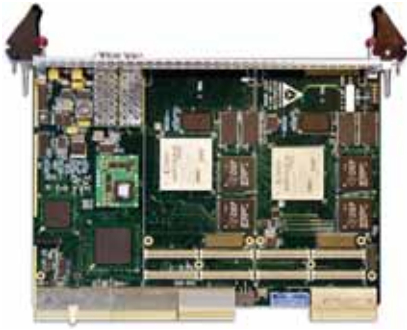


Figure 4 – Quadia DSP/FPGA card with dual Xilinx FPGAs and four TI DSPs

The II DRR system was tested to show channel filter response, system frequency response, and signal quality – and demonstrated excellent results. The results closely matched the theoretical performance predicted by the MATLAB simulations. Each output channel has a tuning resolution of 10 kHz and a noise floor of around -90 dB. The system dynamic range was shown to be greater than 80 dB.

Customize Your Applications

More than 40 percent of the Virtex-II Pro logic blocks and four high-throughput



Figure 5 – UWB PMC module with dual 250 MSPS A/Ds

II DRR Implemented in Hardware

The II DRR system comprises one Quadia DSP card and two ultra-wideband (UWB) PMC I/O modules that can perform digital down conversions and channel filtering on 40 channels simultaneously. The high-speed, front-end signal processing for the DRR is implemented in Virtex-II Pro XC2VP40 FPGAs on the Quadia card and UWB modules, as shown in Figures 4 and 5.

The Quadia card has four TI 1 GHz TMS320C6416 DSPs that are used for baseband processing. The DRR has 40 independently tuned channels that deliver captured in-phase and quadrature (IQ) data to the DSPs. The FPGAs implement 16 MB data queues for each channel with flow control to the DSPs. This configuration allows the DRR to efficiently process the large data rates of the DRR system.

DSP chips are available for your custom applications. The software provided for the II DRR system supports the complete configuration of your system. The software also supports using the DSPs on the Quadia board for channel baseband processing.

The DRR system supports either a single pair of DSPs or a full configuration with four DSPs and two UWB modules. It also supports logic and DSP program downloads to all of the devices in the system through a host PCI. This allows dynamic reconfiguration of the system for multi-protocol applications.

C++ development libraries on both the host and target DSPs are provided to perform hardware and data-flow management. Most system operations – including data transfer from target to host, loading system

logic, and DSP common object file format (COFF) file downloading – can be performed with a single function call using Innovative's Malibu host library. DSP functions, including device drivers for the DRR interface and controls, are included in Innovative's Pismo library and run under TI's DSP/BIOS RTOS.

An advantage of the Malibu library is that the code to control the baseboards is standard C++ code, and the library is portable between different compilers. The Malibu library supports Borland C++ Builder 6 and Microsoft Visual Studio 2003.

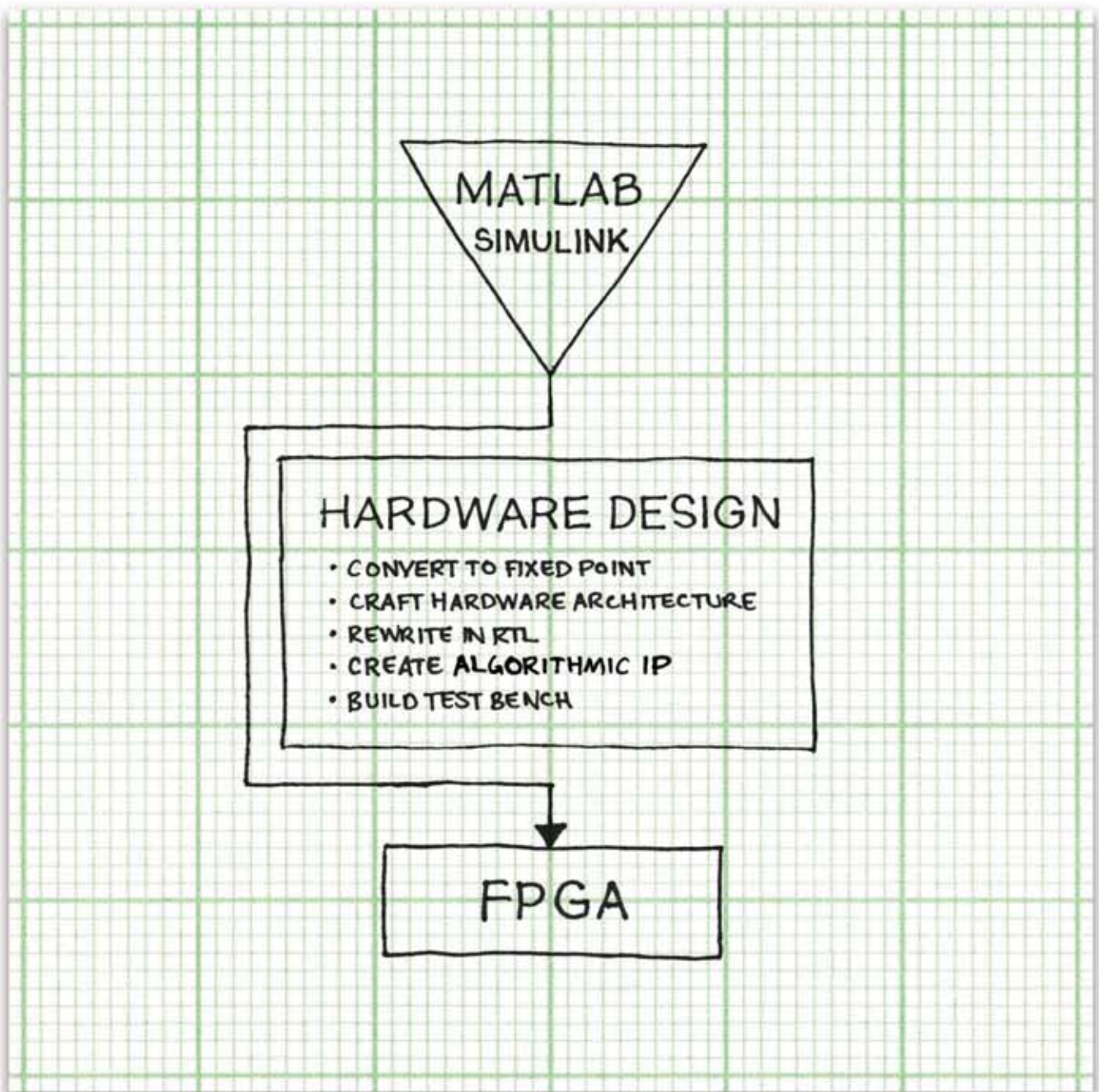
The configuration software allows the setup of each channel on the DRR. You can configure each channel to have its own A/D input, tuning frequency, gain, and spectral inversion. Tuning frequencies are saved relative to a base frequency. This base frequency can be measured in calibration and loaded into the program to allow selection of precise tuning frequencies. All configurations and settings may be saved and reloaded for convenience.

Once the configuration is set, you can download selected target programs to the designated DSPs. A global trigger to the system for all channels begins data flow in the system. The DSPs continuously process data from the DRR and deliver the data to the host.

Conclusion

Innovative Integration's digital radio receiver system is a practical multi-channel SDR application that takes you step-by-step from requirements to implementation. Using MATLAB Simulink DSP tools to define specific digital signal processing, you can directly – and quickly – implement your SDR model into Xilinx Virtex-II Pro FPGAs and TI DSPs using Xilinx System Generator and ISE software. The final hardware is realized on Quadia DSP cards and UWB PMC modules using Innovative's comprehensive FrameWork Logic system and DSP software support.

Everything you need to produce a working SDR application based on the II DRR system model is available through the Innovative Integration website at www.innovative-dsp.com.



NOW, THERE'S A FLOW YOU COULD GET USED TO.

XtremeDSP Sidestep tedious hardware development tasks with Xilinx System Generator for DSP and AccelDSP design tools. When combined with our library of algorithmic IP and hardware platforms, Xilinx XtremeDSP Solutions get you from design to verified silicon faster than ever.

Visit www.xilinx.com/dsp today and speed your design with AccelDSP and System Generator for DSP.

XILINX[®]
www.xilinx.com/dsp

Eliminate Packet Buffering Busywork

The Packet Queue IP core simplifies packet buffering and aggregation, letting you focus on high-level system design.



by Russ Nelson
Senior Design Engineer
Xilinx, Inc.
russ.nelson@xilinx.com

Stacey Secatch
Staff Design Engineer
Xilinx, Inc.
stacey.secatch@xilinx.com

United Parcel Service and FedEx are arguably two of the most sophisticated package delivery services in the world, but they still do not build their own trucks. They know that their strategic advantage lies in figuring out how to move packages around the world efficiently.

Take a lesson from these two successful companies – let the new Xilinx® Packet Queue LogiCORE™ IP be the delivery vehicle for your packets so that you can focus on your strategic advantage instead of wasting time and money on the mechanics of packet buffering.

Packet Queue joins FIFO Generator and Block Memory Generator in an impressive

portfolio of Xilinx no-cost memory cores. Packet Queue buffers packetized data, multiplexes one or more channels together, provides advanced features such as retransmission, discarding of unwanted or corrupted packets, and includes interfaces for complete scheduling control.

Using Xilinx CORE Generator™ software (as shown in Figure 1), you can configure and generate an optimized, pre-engineered solution to your protocol bridging, aggregation, or packet-buffering application. And because Packet Queue provides an optimized and supported solution at no cost, you will realize real savings in terms of NRE and time to market. In this article, we'll highlight Packet Queue's features and show how it can enhance two sample networking designs.

Channelized Data Transfer

Seldom does the output from a system look identical to the input. This is particularly the case in networking applications, where data is often encapsulated and re-encapsulated in different protocols

as it travels through the network. Packet Queue is particularly suited for these systems. It supports as many as 32 input channels, transparently performing data-width conversion while simultaneously migrating data from each independent input clock domain to the output clock domain. Packet Queue's sideband data feature is also extremely versatile, enabling unusual data widths and additional control or status signaling.

Shared Memory Architecture

Packet Queue reduces your design cost, not only as a no-cost core but also by reducing FPGA resources versus traditional implementations. Packet buffering memory is segmented and allocated to packets dynamically as they are written to the core. This enables sharing of memory between data from all channels. You can therefore configure the overall size of the core to represent the peak requirements of the system, as opposed to the sum of the peak requirements of each channel.

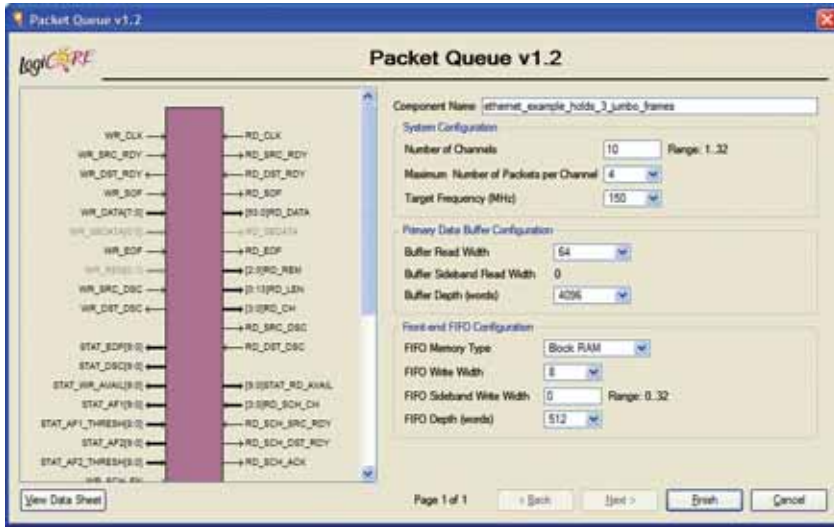


Figure 1 – Packet Queue GUI showing potential configuration for Ethernet example.

Overall memory needs will vary greatly from design to design. Packet Queue enables you to pick the solution that is right for your system’s specifications, resulting in memory savings that translate directly to a lower unit cost for you.

Retransmit and Discard

If your application requires the ability to recover from receiving corrupted packets or to resend packets corrupted during transmission, Packet Queue provides simple yet powerful options. Packet Queue’s input logic supports the discard of an incoming packet at any point before packet completion by asserting a single signal. This functionality can also be used to discard unwanted packets, such as those with invalid cyclic redundancy checks (CRCs). Similarly, you can interrupt a packet being transmitted with a single signal, enabling retransmission at a later time. Together, these two simple yet powerful features allow Packet Queue to easily interface with unreliable links.

Complete Scheduling Control

The most powerful feature of Packet Queue isn’t something included in the core – it’s something *not* included in the core. As a designer of a networking system, your competitive advantage lies in your scheduling algorithms. Packet Queue provides status and control ports for directing multiplexing

between data channels, letting you decide how to transfer data based on the requirements of your system – whether that means round robin, weighted round robin, fixed priority, or anything else you can design. For example, you can choose to give priority to channels that require a higher quality of service (QoS) to reduce queuing latency.

Aggregation Example

Our first example application is that of an Ethernet bridge, between ten 1-Gb ports and a single 10-Gb port, as shown in Figure 2. Packet Queue can assist you with aggregation. Several of its features come into play here:

- Multiple channels – this configuration uses 10 of Packet Queue’s 32 available input channels. You can also configure more input ports for oversubscription of the 10-Gb link.
- Clock conversion – the Xilinx 1-Gb Ethernet MAC LogiCORE solution (GEMAC) provides data at a frequency of 125 MHz. The Xilinx 10-Gb Ethernet MAC LogiCORE solution (10GEMAC) expects data at 156.25 MHz. Packet Queue seamlessly resynchronizes data from one clock domain to the other, allowing each input GEMAC core to exist in its own clock domain.
- Width conversion – the GEMAC core provides 8-bit data, while the 10GEMAC core expects 64-bit data.

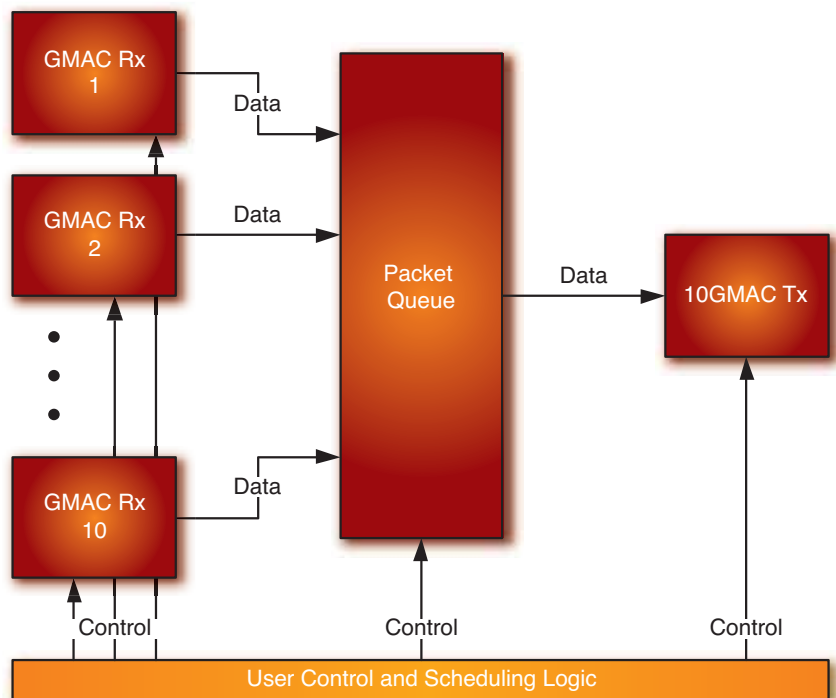


Figure 2 – Packet Queue aggregation of 10 Ethernet channels

Packet Queue automatically converts between the different widths.

- Scheduling control – Packet Queue’s versatile scheduling interfaces enable you to implement any algorithm you desire to control data flow through the core. A simple system might use a round-robin scheme, while a more complex implementation could include QoS, giving priority to particular ports based on the type of data expected.
- Shared memory architecture – the dynamic memory allocation used by Packet Queue is particularly suited to Ethernet because it supports variable frame sizes, including jumbo frames. Because memory is shared between channels, the total memory needed is reduced, while still handling bursty data. Simply determine the peak outstanding data and apply it to the shared memory size instead of the size of each channel. For example, the core as configured in the GUI shown in Figure 1 supports as many as three stored jumbo frames at any given time, in addition to smaller standard frames.

Buffering Example

Our second example application, shown in Figure 3, uses Packet Queue to provide

transmit buffering and retry capabilities for the Xilinx Serial RapidIO LogiCORE solution (SRIO). The RapidIO protocol allows for as many as 31 outstanding (unacknowledged) packets, with the possibility of retransmitting any or all of the outstanding packets. Packet Queue is ideally suited to provide this amount of buffering, as well as providing other features to make the integration as simple and seamless as possible. These features include:

- LocalLink Interfaces – both Packet Queue and SRIO use the Xilinx LocalLink standard for their data interfaces. This enables you to connect the two cores’ data interfaces together without any glue logic. No glue logic means faster development and less time spent debugging.
- Retransmit packets – Packet Queue’s retransmit capability meshes quite well with the requirements of RapidIO. The SRIO core provides two interfaces through which packet retransmit requests are made. When there is a problem with the packet currently on the transmit interface, SRIO asserts its “discontinue” signal and the packet is retransmitted by the Packet Queue without any further intervention by your control logic. When retransmission of

more than one packet is required, your control logic then translates the request for all of the failed packets currently located in the Packet Queue and schedules them for resend as normal. Following packet acknowledgment, your logic then instructs Packet Queue to free the memory that was allocated.

- Sideband data and channelization – RapidIO optionally allows packets with different priorities. Packet Queue supports this in either of two ways, giving you the flexibility to pick the implementation that best suits your system requirements. A simpler way is to configure the core with a single bit of sideband data that is mapped to the SRIO core’s critical request input. Doing so causes the critical request flag to pass through the Packet Queue along with the associated packet.

The more complex (but more powerful) method for implementing priorities is to use multiple Packet Queue channels, with prioritized and critical request packets written to separate channels. This strategy allows higher priority packets to pass lower priority packets in the Packet Queue, but requires you to implement more complex scheduling and retransmit control logic.

Conclusion

The Xilinx Packet Queue LogiCORE IP is a simple yet powerful solution for protocol bridging, aggregation, and packet buffering systems. In addition to the examples we’ve discussed, Packet Queue is great for bridging SPI-3 to SPI-4.2, SPI-4.2 to XAUI, PCI to PCI Express, and countless other protocol combinations. As networking protocols multiply and networks carry more diverse types of data, the need for these types of systems will only grow. Packet Queue provides a cost-effective solution for your designs and lets you deliver your product to your customers earlier.

For more information about the Packet Queue, visit www.xilinx.com/systemio/packet_queue. Feedback and suggestions for new features are always welcome. To contact the Packet Queue team, e-mail packet_queue@xilinx.com.

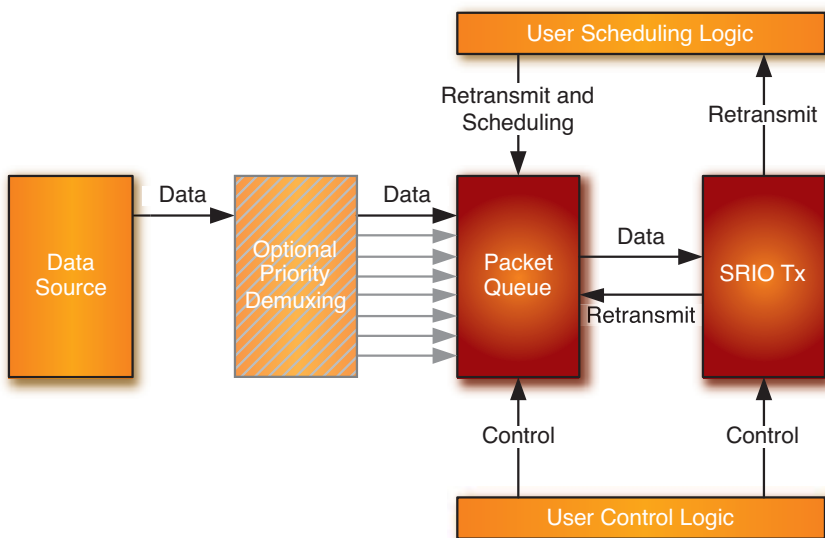


Figure 3 – Serial RapidIO transmit buffering with optional priority reordering

Onward to Glory

Complete Software Radio Solution on a single 6U card



Quixote

Features

- ▶ 6 Million gate Virtex-II FPGA
 - Scalable Software Defined Radio
 - Embedded Processing via MatLab
- ▶ TMS320C6416 DSP
- ▶ 105 MHz, 14 bit 2 Ch. Analog I/O
- ▶ 32 MB RAM
- ▶ 32/64-bit cPCI bus
- ▶ PMC expansion site
- ▶ STAR Fabric interface

Get your data sheets now!

www.innovative-dsp.com/quixote

sales@innovative-dsp.com
805.520.3300 phone • 805.579.1730 fax

Innovative Integration
... real time solutions!

XILINX EVENTS

Xilinx participates in numerous trade shows and events throughout the year. This is a perfect opportunity to meet our silicon and software experts, ask questions, see demonstrations of new products, and hear other customer success stories.

For more information and the current schedule, visit www.xilinx.com/events/.

North America

Aug. 8-10	NI Week 2006	Austin, TX
Aug. 16	Next-Gen Networking Chips	San Jose, CA
Sept. 26-28	MAPLD 2006	Washington, DC
Oct. 16-18	Convergence 2006	Detroit, MI
Oct. 23-25	MILCOM 2006	Washington, DC

Europe

Sept. 27-29	RADECS	Athens, Greece
Sept. 7-13	IBC	Amsterdam, Holland

Japan

Sept. 28	Hi Speed Interface Workshop	Tokyo, Japan
----------	-----------------------------	--------------

Stay Ahead!



PCI based development system also available

with the PMC-FPGA05 Range of Virtex-5 PMCs

Xilinx Virtex-5 LX FPGA
A new generation of performance

Analog I/O, Camera Link, LVDS, FPDP-II & RS422/485 options
Fast, integrated I/O without bottlenecks

Multiple banks of fast memory
DSP & I/O optimized memory architecture

PCI-X Interface with multiple DMA controllers
More than 1GB/s bandwidth to host

Libraries and Example Code
Easy to use with head-start time-to-market

Processing and FPGA - Input/Output - Data Recording - Bus Analyzers

For more information, please visit virtex5.vmetro.com or call (281) 584 0728

VMETRO
innovation deployed

Product Selection Matrix



	CLB Resources				Memory Resources				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)	
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	-4 -5	-4	0.6M	
XC3S150E	250K	26 x 34	2,448	5,508	4896	38K	12	216K	12	5/326	4	NO	68	172	LVTTL, LVCMOS3.3/2.5/1.8/ 1.5/1.2, PCI 3.3V – 32/64-bit	-4 -5	-4	1.4M	
XC3S500E	500K	34 x 46	4,656	10,476	9312	73K	20	360K	20	5/326	4	NO	92	232	33/60MHz PCI-X 100MHz, SSTL1.8/2.5, HSTL1.1/1.8, HSTL III/1.8	-4 -5	-4	2.3M	
XC3S1200E	1,200K	46 x 60	8,672	19,512	17344	136K	28	504K	28	5/326	8	NO	124	304	Differential LVDS2.5, Bus LVDS2.5, mini-LVDS, RSDS, LVPECL	-4 -5	-4	3.8M	
XC3S1600E	1,600K	58 x 76	14,752	33,192	29504	231K	36	648K	36	5/326	8	NO	156	376		-4 -5	-4	5.9M	
Spartan-3 and Spartan-3L Families – 1.2 Volt (see note 2)																			
XC3S500	500K	16 x 12	768	1,728	1,536	12K	4	72K	4	24/280	2	YES	56	124	Single-ended	-4 -5	-4	.4M	
XC3S1000	200K	24 x 20	1,920	4,320	3,840	30K	12	216K	12	24/280	4	YES	76	173	LVTTL, LVCMOS3.3/2.5/1.8/ 1.5/1.2, PCI 3.3V – 32/64-bit	-4 -5	-4	1.0M	
XC3S4000	400K	32 x 28	3,584	8,064	7,168	56K	16	288K	16	24/280	4	YES	116	264	33MHz SSTL2 Class I & II, SSTL18 Class I, HSTL Class I, III, HSTL1.8 Class I, II & III, GTL, GTL+	-4 -5	-4	1.7M	
XC3S10000	1,000K	48 x 40	7,680	17,280	15,360	120K	24	432K	24	24/280	4	YES	175	391		-4 -5	-4	3.2M	
XC3S15000	1,500K	64 x 52	13,312	29,952	26,624	208K	32	576K	32	24/280	4	YES	221	487		-4 -5	-4	5.2M	
XC3S20000	2,000K	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_srx2.5, RSDS, LD2.5, LVPECL	-4 -5	-4	7.7M	
XC3S40000	4,000K	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	
XC3S50000	5,000K	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.

Platform Flash Features

	XC3F01S	XC3F02S	XC3F04S	XC3F08P	XC3F16P	XC3F32P
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
VCC0 (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
VCC1 (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	VOG48 VOG48	VOG48 VOG48	VOG48 VOG48
Availability	Now	Now	Now	Now	Now	Now

Package Options and User I/O¹

Pin	Area ²	Spartan-3E (1.2V)						Spartan-3 (1.2V)											
		XC3S100E	XC3S250E	XC3S500E	XC3S1200E	XC3S1600E	IO ³	108	172	232	304	376	XC3S500	XC3S1000	XC3S1500	XC3S2000	XC3S4000	XC3S5000	
PQFP Packages (PQ) – wire-bond plastic QFP (0.5 mm lead spacing)																			
208	30.6 x 30.6 mm			158	158														
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														
TQFP Packages (TQ) – thin QFP (0.5 mm lead spacing)																			
144	22.0 x 22.0 mm			108	108														
Chip Scale Packages (CP) – wire-bond chip-scale BGA (0.5 mm ball spacing)																			
132	8 x 8 mm			92	92														
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													
FGA Packages (FG) – wire-bond fine-pitch BGA (1.0 mm ball spacing)																			
256	17 x 17 mm																		
280	19 x 19 mm			232	250	250													
400	21 x 21 mm																		
456	23 x 23 mm																		
488	23 x 23 mm																		
676	27 x 27 mm																		
900	31 x 31 mm																		
1156	35 x 35 mm																		
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.

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/pp/
- 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/gsl/



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		Speed			Clocking	
	Maximum I/O	I/O Banking	Min. Pin-to-pin Logic Delay (ns)	Commercial Speed Grades (fastest to slowest)	Industrial Speed Grades (fastest to slowest)	IQ Speed Grade	Global Clocks	Product Term Clocks per Function Block									
CoolRunner-II Family – 1.8 Volt																	
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																	
XCR3032XL	750	32	48	3.3/5	3.3	36	5	5	-5-7-10	-7-10	-10	4	16				
XCR3064XL	1,500	64	48	3.3/5	3.3	68	6	6	-6-7-10	-7-10	-10	4	16				
XCR3128XL	3,000	128	48	3.3/5	3.3	108	6	6	-6-7-10	-7-10	-10	4	16				
XCR3256XL	6,000	256	48	3.3/5	3.3	164	7.5	7.5	-7-10-12	-10-12	-12	4	16				
XCR3384XL	9,000	384	48	3.3/5	3.3	220	7.5	7.5	-7-10-12	-10-12	-12	4	16				
XCR3512XL	12,000	512	48	3.3/5	3.3	260	7.5	7.5	-7-10-12	-10-12	-12	4	16				

Package Options and User I/O

CoolRunner-II		CoolRunner XPLA3											
Part	Area ¹	XC2C32A	XC2C64A	XC2C128	XC2C256	XC2C384	XC2C512	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 (TF) – 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		Speed				Clocking			
	800	1,600	3,200	6,400	36	72	144	288	2.5/3.3	2.5/3.3	1.8/2.5/3.3	36	1	5	5	-5 -7	-7	NA	3	18
XC9500XV Family – 2.5 Volt																				
XC9536XV	800	1,600	3,200	6,400	36	72	144	288	2.5/3.3	2.5/3.3	1.8/2.5/3.3	36	1	5	5	-5 -7	-7	NA	3	18
XC9572XV	1,600	3,200	6,400		72	144	288		2.5/3.3	2.5/3.3	1.8/2.5/3.3	72	1	5	5	-5 -7	-7	NA	3	18
XC95144XV	3,200	6,400			144	288			2.5/3.3	2.5/3.3	1.8/2.5/3.3	117	2	5	5	-5 -7	-7	NA	3	18
XC95288XV	6,400				288				2.5/3.3	2.5/3.3	1.8/2.5/3.3	192	4	6	6	-6 -7 -10	-7 -10	NA	3	18
XC9500XL Family – 3.3 Volt																				
XC9536XL	800	1,600	3,200	6,400	36	72	144	288	2.5/3.3/5	2.5/3.3/5	2.5/3.3	36	1	5	5	-5 -7 -10	-7 -10	-10	3	18
XC9572XL	1,600	3,200	6,400		72	144	288		2.5/3.3/5	2.5/3.3/5	2.5/3.3	72	1	5	5	-5 -7 -10	-7 -10	-10	3	18
XC95144XL	3,200	6,400			144	288			2.5/3.3/5	2.5/3.3/5	2.5/3.3	117	2	5	5	-5 -7 -10	-7 -10	NA	3	18
XC95288XL	6,400				288				2.5/3.3/5	2.5/3.3/5	2.5/3.3	192	4	6	6	-6 -7 -10	-7 -10	NA	3	18
XC9500 Family – 5 Volt																				
XC9536	800	1,600	3,200	6,400	36	72	144	288	5	5	5	36	10	10	10	-5 -6 -10 -15	-7 -10 -15	-15	3	18
XC9572	1,600	3,200	6,400		72	144	288		5	5	5	72	10	10	10	-7 -10 -15	-10 -15	-15	3	18
XC95108	2,400	4,800			108	216			5	5	5	108	10	10	10	-7 -10 -15 -20	-7 -10 -15 -20	NA	3	18
XC95144	3,200	6,400			144	288			5	5	5	133	10	10	10	-7 -10 -15	-10 -15	NA	3	18
XC95216	4,800				216				5	5	5	166	10	10	10	-10 -15 -20	-10 -15 -20	NA	3	18
XC95288	6,400				288				5	5	5	192	10	10	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	
XC9536XV	XC9572XV	XC9536XL	XC9572XL
XC95288XV	XC95144XV	XC95288XL	XC95144XL
Area ¹			
PLCC Packages (PC) – wire-bond plastic chip carrier (1.27 mm lead spacing)			
44	17.5 x 17.5 mm	34	34
84	30.2 x 30.2 mm		
PQFP Packages (PO) – wire-bond plastic QFP (0.5 mm lead spacing)			
100	23.3 x 17.2 mm		
160	31.2 x 31.2 mm		
208	30.6 x 30.6 mm	168	
VQFP Packages (VO) – very thin QFP (0.5 mm lead spacing)			
44	12.0 x 12.0 mm	34	34
64	12.0 x 12.0 mm	36	52
TQFP Packages (TO) – thin QFP (0.5 mm lead spacing)			
100	16.0 x 16.0 mm	72	81
144	22.0 x 22.0 mm	117	117
Chip Scale Packages (CS) – wire-bond chip-scale BGA (0.8 mm ball spacing)			
48	7 x 7 mm	36	38
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		
FBGA Packages (FG) – wire-bond Fine-line BGA (1.0 mm ball spacing)			
256	17 x 17 mm		192

Note 1: Area dimensions for lead-frame products are inclusive of the leads.



The Programmable Logic Company™

Xilinx CPLD

www.xilinx.com/devices/

ISE™ 8.2i – Powered by ISE Fmax Technology

Platforms	Feature	ISE WebPACK™	ISE Foundation™
	Virtex™ Series	Microsoft Windows 2000 / XP Red Hat Enterprise Linux 3/4 (32 bit)	Microsoft Windows 2000 / XP Sun Solaris 2.8 or 2.9 Red Hat Enterprise Linux 3/4 (32 & 64 bit)
Devices		Virtex: XCV50 - XCV600 Virtex-E: XCV50E - XCV600E Virtex-II: XC2V40 - XC2V500 Virtex-II Pro: XC2VP2 - XC2VP7 Virtex-4: LX: XC4VLX15, XC4VLX25 SX: XC4VFX25 FX: XC4VFX12 Virtex Q: XQV100 - XQV600 Virtex QR: XQVR300, XQVR600 Virtex-E Q: XQV600E	Virtex: All Virtex-E: All Virtex-II/Virtex-II Pro: All Virtex-4: LX: All SX: All FX: All Virtex-5: LX: XC5VLX30 - XC5VLX220 Virtex Q: QR: All Virtex-E Q: All
	Spartan™ Series	Spartan-II/III: All Spartan-3: XC3S50 - XC3S1500 Spartan-3E: All Spartan-3L: XC3S1000L, XC3S1500L XA (Xilinx Automotive) Spartan-3: All	Spartan-II/III: All Spartan-3: All Spartan-3E: All Spartan-3L: All XA (Xilinx Automotive) Spartan-3: All
	CoolRunner™ XPLA3 CoolRunner-II / CoolRunner-IIA XC9500™ Series		All All
Design Entry	HDL Editor / Schematic Editor State Diagram Editor Xilinx CORE Generator™ System RTL & Technology Viewers Architecture Wizards	Microsoft Windows only Yes Yes Yes Yes	Yes Yes Yes Yes Yes
Embedded Design	3rd Party RTL Checker Support Xilinx System Generator for DSP Embedded Design Kit (EDK)	Sold as an Option Sold as an Option	Sold as an Option Sold as an Option
Synthesis	XST – Xilinx Synthesis Technology Mentor Graphics® Leonardo Spectrum™ Mentor Graphics Precision RTL™ Mentor Graphics Precision Physical™ Synplify/Pro/Premier ABEL	Integrated Interface (EDIF Interface on Linux) Integrated Interface EDIF Interface Integrated Interface CPD (Microsoft Windows only)	Yes Yes Yes Yes Yes

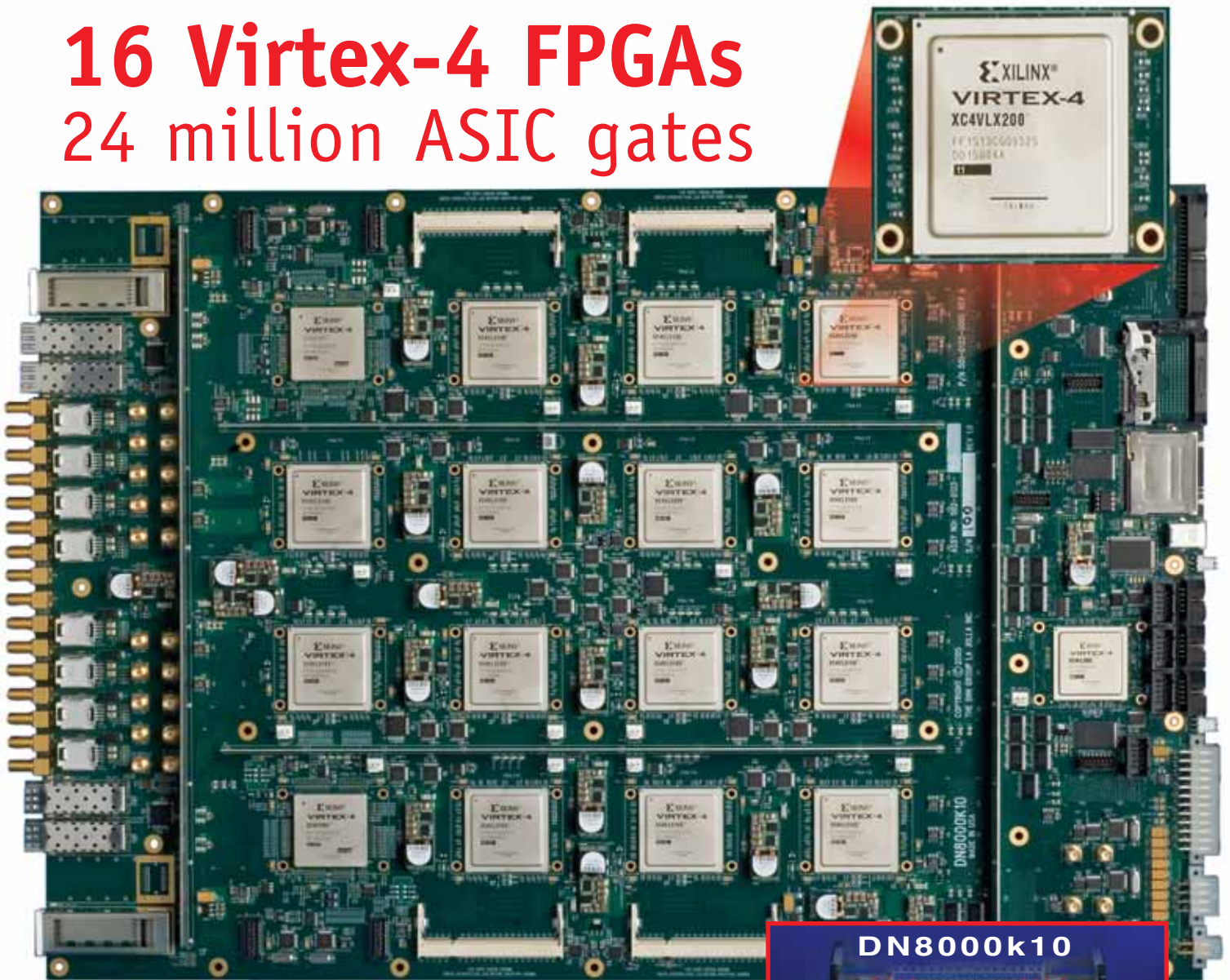
Implementation	Feature	ISE WebPACK™	ISE Foundation™
	FloorPlanner PlanAhead™		Yes Sold as an Option Free downloadable evaluation at www.xilinx.com/planahead
	Timing Driven Place & Route Integrated Timing Closure Environment Partition Support Incremental Design Timing Improvement Wizard Xplorer	No	Yes Yes (Virtex-5 only) Yes Yes Yes Yes
Programming	IMPACT / System ACE™ CableServer IBIS / STAMP / HSPICE™ Models		Yes Yes
Board Level Integration	ELDO Models™ (MGT only)		Yes
	ChipScope™ Pro ChipScope Pro Serial IO Toolkit Graphical Testbench Editor ISE Simulator Lite		Sold as an Option Free downloadable evaluation at www.xilinx.com/chipscopepro Microsoft Windows only Yes
Verification	ISE Simulator ModelSim® XE III Starter ModelSim XE III Static Timing Analyzer FPGA Editor with Probe Chip Viewer XPower (Power Analysis) 3rd Party Equivalency Checking Support SMARTModels for PowerPC™ and RocketIO™ 3rd Party Simulator Support	ISE Simulator is available as an Optional Design Tool for ISE Foundation only Sold as an Option Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes 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



16 Virtex-4 FPGAs

24 million ASIC gates



New High Speed ASIC Prototyping Engine - Now shipping!

- Logic prototyping system with 2-16 Xilinx Virtex™-4 FPGAs
 - 14 LX100/160/200 and 2 FX60/100
 - Nearly 24M ASIC gates (LSI measure - not inflated)
- FPGA interconnect is single-ended or LVDS (350MHz & 700mbit/s)
- Synplicity Certify™ models for partitioning assistance
- 4 separate DDR2 SODIMMs (200MHz)
 - 64-bit data width, 200MHz (400mbit/s), 4GB/each
 - Optional modules: Mictor, SSRAM, RLDRAM, FLASH
- Multi-gigabit serial I/O interfaces via XFP, SFP, and SMA's
- Flexible customization via daughter cards
 - 1200 I/O pins, LVDS capable, 350MHz
- Fast and Painless FPGA configuration via Compact FLASH, or USB
- Full support for embedded logic analyzers via JTAG interface: ChipScope™, Identify™

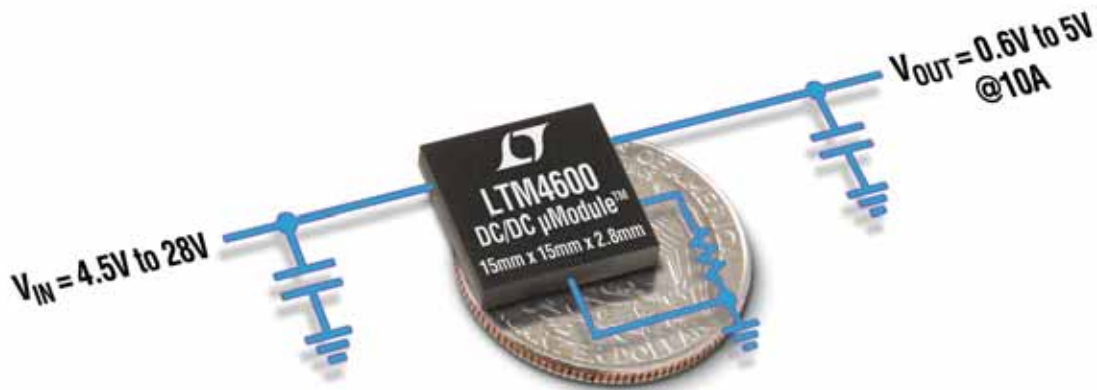


Try our other
Virtex-4 products

The
DiNI
Group

The DINI Group
1010 Pearl Street, Suite 6
La Jolla, CA 92037
(858) 454-3419
www.dinigroup.com

Instant 10A Power Supply



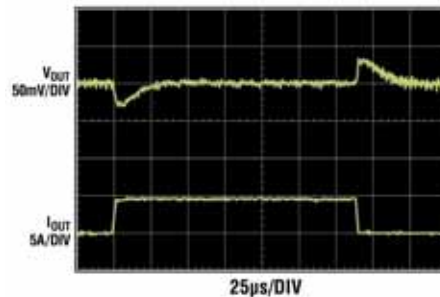
Complete, Quick & Ready.

The LTM[®]4600 is a complete 10A switchmode step-down power supply with a built-in inductor, supporting power components and compensation circuitry. With high integration and synchronous current mode operation, this DC/DC μ Module[™] delivers high power at high efficiency in a tiny, low profile surface mount package. Supported by Linear Technology's rigorous testing and high reliability processes, the LTM4600 simplifies the design and layout of your next power supply.

▼ Features

- 15mm x 15mm x 2.8mm LGA with 15°C/W θ_{JA}
- Pb-Free (e⁴), RoHS Compliant
- Only C_{BULK} Required
- Standard and High Voltage:
LTM4600EV: 4.5V V_{IN} 20V
LTM4600HVEV: 4.5V V_{IN} 28V
- 0.6V V_{OUT} 5V
- I_{OUT}: 10A DC, 14A Peak
- Parallel Two μ Modules for 20A Output

Ultrafast Transient Response 2% Δ V_{OUT} with a 5A Step



V_{IN} = 12V, V_{OUT} = 1.5V, 0A to 5A Load Step
(C_{OUT} = 3 x 22 μ F CERAMICS, 470 μ F POS CAP)

▼ Information

Nu Horizons Electronics Corp.

Tel: 1-888-747-NUHO

www.nuhorizons.com/linear



REE
LTM4600
 μ Module Board
to qualified customers.

LTC, LT, LTM and PolyPhase are registered trademarks and μ Module is a trademark of Linear Technology Corporation. All other trademarks are the property of their respective owners.

**NU HORIZONS
ELECTRONICS CORP.**

**LINEAR
TECHNOLOGY**

High VELOCITY LEARNING



Nu Horizons Electronics Corp. is proud to present our newest education and training program - **XpressTrack** - which offers engineers the opportunity to participate in technical seminars conducted around the country by experts focused on the latest technologies from Xilinx. This program provides higher velocity learning to help minimize start-up time to quickly begin your design process utilizing the latest development tools, software and products from both Nu Horizons and Xilinx.

For a complete list of course offerings, or to register for a seminar near you, please visit:

www.nuhorizons.com/xpresstrack



Courses Available

- **Optimizing MicroBlaze Soft Processor Systems**
 - 4 hour class
 - Covers building a complete customized MicroBlaze soft processor system
- **Video/Imaging Algorithms in FPGAs**
 - 1 day class
 - Verify designs onto actual hardware using Ethernet-based hardware-in-the-loop co-simulation.
- **Introduction to Embedded PowerPC and Co-Processor Code Accelerators**
 - 4 hour class
 - Covers how to build a high performance embedded PowerPC system
- **Xilinx Spotlight: Virtex 5 Architectural Overview**
 - Learn about the new 65 nm family of FPGAs from Xilinx
 - Understand how the new ExpressFabric will improve logic utilization
- **Fundamentals of FPGA**
 - 1 day class
 - Covers ISE 8.1 features
- **ISE Design Entry**
 - 1 day class
 - Covers XST, ECS, StateCAD and ISE simulator
- **Fundamentals of CPLD Design**
 - 1 day class
 - Covers CPLD basics and interpreting reports for optimum performance
- **Design Techniques for Low Cost**
 - 1 day class
 - Covers developing low cost products particularly in high volume markets
- **VHDL for Design Engineers**
 - 1 day class
 - Covers VHDL language and implementation for FPGAs & CPLDs

EXPRESSFABRIC

Ultimate Performance...



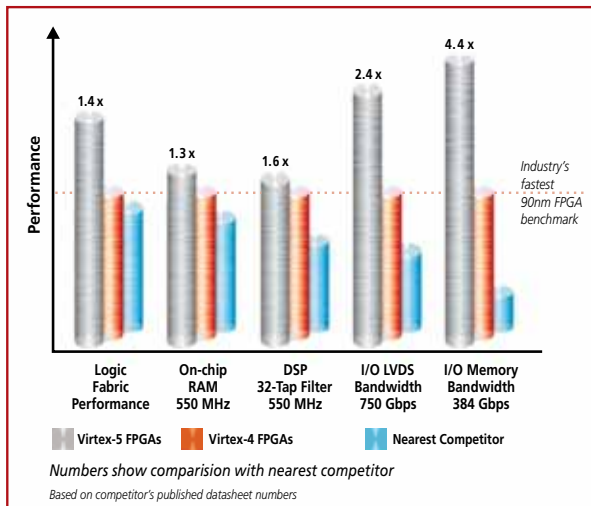
Achieve highest system speed and better design margin with the world's first 65nm FPGAs.

Virtex™-5 FPGAs feature ExpressFabric™ technology on 65nm triple-oxide process. This new fabric offers the industry's first LUT with six independent inputs for fewer logic levels, and advanced diagonal interconnect to enable the shortest, fastest routing. Now you can achieve 30% higher performance, while reducing dynamic power by 35% and area by 45% compared to previous generations.

Design systems faster than ever before

Shipping now, Virtex-5 LX is the first of four platforms optimized for logic, DSP, processing, and serial. The LX platform offers 330,000 logic cells and 1,200 user I/Os, plus hardened 550 MHz IP blocks. Build deeper FIFOs with 36 Kbit block RAMs. Achieve 1.25 Gbps on all I/Os without restrictions, and make reliable memory interfacing easier with enhanced ChipSync™ technology. Solve SI challenges and simplify PCB layout with our sparse chevron packaging. And enable greater DSP precision and dynamic range with 550 MHz, 25x18 MACs.

Visit www.xilinx.com/virtex5, view the TechOnline webcast, and give your next design the ultimate in performance.



The Programmable Logic Company™

www.xilinx.com/virtex5



The Ultimate System Integration Platform