CMP
United Business Media

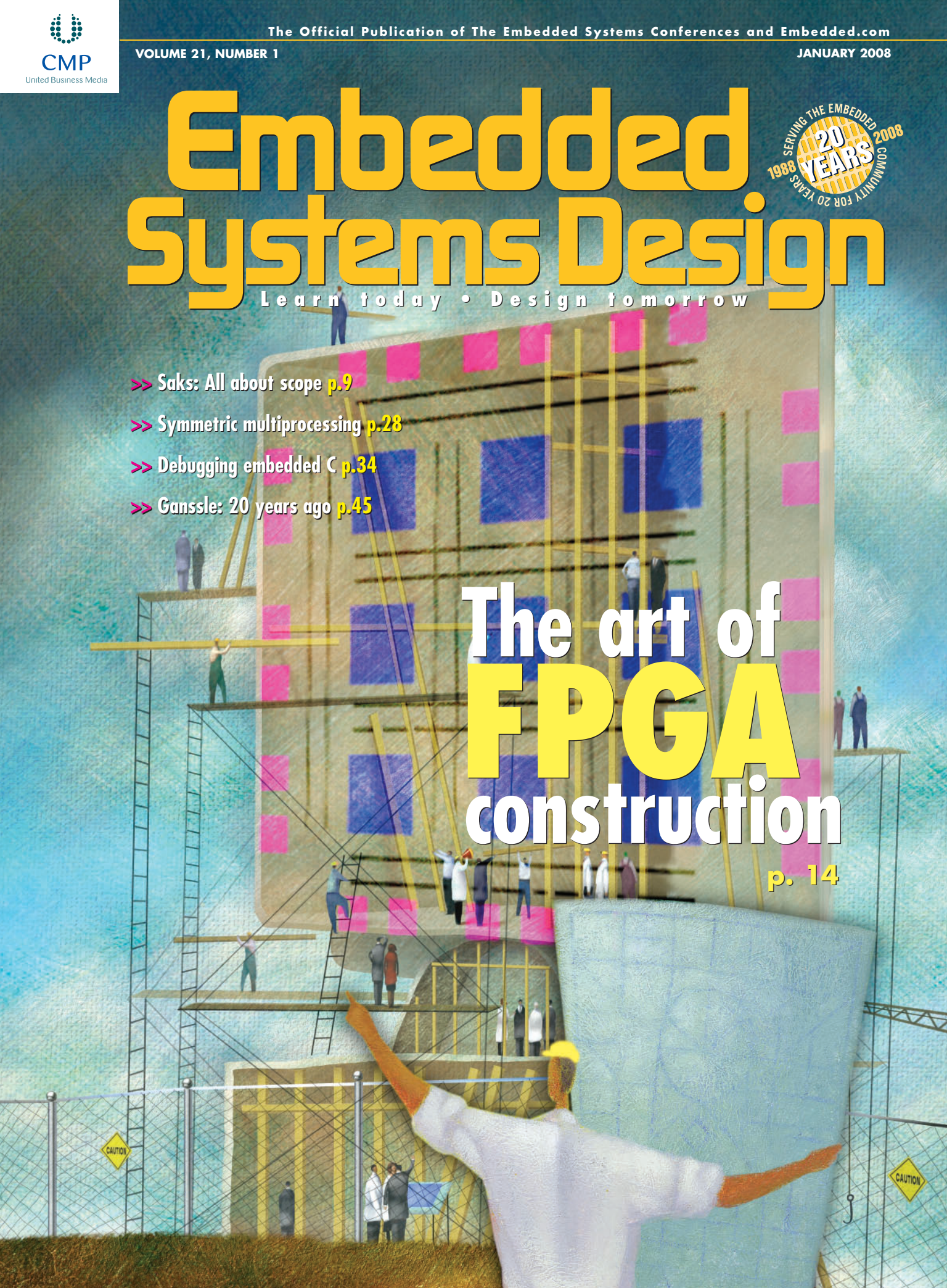
The Official Publication of The Embedded Systems Conferences and Embedded.com

VOLUME 21, NUMBER 1 JANUARY 2008

Embedded Systems Design

Learn today • Design tomorrow

SERVING THE EMBEDDED COMMUNITY FOR 20 YEARS 1988 20 YEARS 2008



The art of FPGA construction

p. 14


90% CUT IN EMISSIONS
AND 50% CUT IN DEVELOPMENT TIME.
THAT'S MODEL-BASED DESIGN.
To meet a tough performance target, the engineering team at Nissan used dynamic system models instead of paper specifications.
The result: 50% time savings, the first car certified to meet California's Partial Zero Emissions Vehicle standard, and a U.S. EPA award.
To learn more, visit mathworks.com/mbd
MATLAB®
&SIMULINK®
The MathWorks
Accelerating the pace of engineering and science
©2005 The MathWorks, Inc.

Need to make sure they line up for your product first?

PROBLEM SOLVED

Green Hills® • SOFTWARE, INC. •

With more than half of the product development cycle consumed by debugging, finding bugs faster means your product will get to market first.

Green Hills Software provides premier tools that pinpoint the most elusive bugs in minutes, instead of hours or days. With the MULTI® development environment's time-saving code analysis tools, errors in code are automatically found, long before the debugging process begins.

MULTI and the TimeMachine™ debugger allow developers to easily find every bug so that shipping a product with known problems becomes a thing of the past.

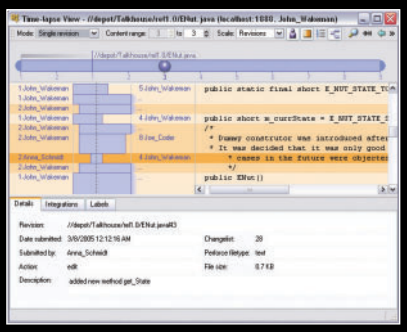
With Green Hills Software's sophisticated technology you'll produce a better product and get it out the door long before your competition.

Call **800-765-4733** or visit us on the web **www.ghs.com** to learn more.

Perforce | *Fast* Software Configuration Management

Introducing Time-lapse View,

a productivity feature of Perforce SCM.



Perforce Time-lapse View

Time-lapse View lets developers see every edit ever made to a file in a dynamic, annotated display. At long last, developers can quickly find answers to questions such as: 'Who wrote this code, and when?' and 'What content got changed, and why?'

Time-lapse View features a graphical timeline that visually recreates the evolution of a file, change by change, in one fluid display. Color gradations mark the aging of file contents, and the display's timeline can be configured to show changes by revision number, date, or changeset number.

Time-lapse View is just one of the many productivity tools that come with the Perforce SCM System.

PERFORCE
SOFTWARE

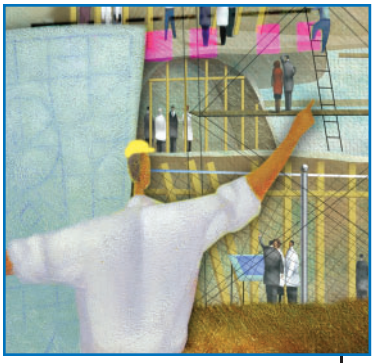
Download a free copy of Perforce, no questions asked, from www.perforce.com. Free technical support is available throughout your evaluation.

THE OFFICIAL PUBLICATION OF THE EMBEDDED SYSTEMS CONFERENCES AND EMBEDDED.COM

Embedded Systems Design

www.embedded.com

JANUARY 2008 VOLUME 21, NUMBER 1

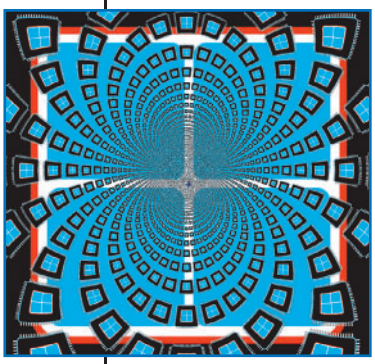


Cover Feature: The art of FPGA construction

BY GINA R. SMITH

Working with FPGAs isn't intimidating when you know the basic techniques and options.

«14



Is symmetric multiprocessing for you?

BY DAVID N. KLEIDERMACHER

Multicore architectures can provide the performance boost you're looking for, but the software is certainly more complicated.

28»

Debugging embedded C

BY ROBIN KNOKE

Has debugging embedded C changed in 20 years? You betcha. But the process will never change: stabilize, isolate, correct, and retest. Here's an article from the 1988 premiere issue of *Embedded Systems Programming*, with some comments from the author, Robin Knoke.

«34

in person

ESC Silicon Valley

San Jose Convention Center
April 14–18, 2008
www.embedded.com/esc/sv/

on-line

www.embedded.com

Web archive:
www.embedded.com/archive

Article submissions:
www.embedded.com/wriguide

Forum discussions:
www.embedded.com/forum

EMBEDDED SYSTEMS DESIGN (ISSN 1558-2493 print; ISSN 1558-2507 PDF-electronic) is published monthly by CMP Media LLC., 600 Harrison Street, 5th floor, San Francisco, CA 94107, (415) 947-6000. Please direct advertising and editorial inquiries to this address.
SUBSCRIPTION RATE for the United States is $55 for 12 issues. Canadian/Mexican orders must be accompanied by payment in U.S. funds with additional postage of $6 per year. All other foreign subscriptions must be prepaid in U.S. funds with additional postage of $15 per year for surface mail and $40 per year for airmail. POSTMASTER: Send all changes to EMBEDDED SYSTEMS DESIGN, P.O. Box 3404, Northbrook, IL 60065-9468. For customer service, telephone toll-free (877) 676-9745. Please allow four to six weeks for change of address to take effect. Periodicals postage paid at San Francisco, CA and additional mailing offices. EMBEDDED SYSTEMS DESIGN is a registered trademark owned by the parent company, CMP Media LLC.

BPA INTERNATIONAL

#include

BY Richard Nass

Acquisitions to enhance coverage

Embedded systems designers can now gain from the experience of their peers, thanks to an abundance of Tear Downs.

CMP, the company that owns this magazine, Embedded.com, and the Embedded Systems Conferences (along with lots of other publications and Web sites, including *EE Times* and TechOnline), recently made two acquisitions. Normally, I wouldn't mention events that occurred on the business side of the house in these pages. However, these two acquisitions could have a great affect on the coverage that you'll see in these pages.

The two acquisitions are Semiconductor Insights (*www.semiconductor.com*) and Portelligent (*www.teardown.com*). If you're not familiar with one or both of these companies, let me shed some light on them.

The Portelligent acquisition was finalized in November. The company's claim to fame is doing Tear Downs. By doing that, they gain intelligence into the design of mobile, wireless, personal, and consumer electronics. With this information, designers can make faster, better, and more cost-effective decisions about their competitive positioning, technology options, investment strategy, intellectual property (IP) position, and marketplace opportunities. Portelligent was formed in 2000 as a spinoff of an Austin-based research consortium.

Richard Nass is editor in chief of Embedded Systems Design. *You can reach him at rnass@cmp.com.*

We've worked with the Portelligent team for years. You may have noticed that the company's Tear Downs have been appearing in our pages and on Embedded.com for some time now, as well as in *EE Times* and on TechOnline. You may also recognize the Portelligent name from the Prius Tear Downs we performed live at the Embedded Systems Conferences. The company had a big hand in that project.

The acquisition of Semiconductor Insights, which occurred last July, has a similar meaning to our group. Semiconductor Insights is also known for its Tear Downs, but they perform them at the IC level rather than at the system level. For example, the company was the first to tear apart and analyze Intel's latest microprocessor, the Penryn 45-nm device.

Semiconductor Insights also serves as a global IP and patent technical advisor. They have the ability to perform technical investigations of patents, ICs, and electronic systems. One division of the company benchmarks competing devices, improves time to market, and solves technical problems, while a second division helps technology companies and legal professionals evaluate, develop, and monetize their IP.

Together, the two companies will offer a combined searchable database of over 40,000 components and ICs, which is an invaluable resource for designers.

Richard Nass
rnass@cmp.com

Embedded Systems Design

Editor in Chief
Richard Nass
(201) 288-1904
rnass@cmp.com

Managing Editor
Susan Rambo
srambo@cmp.com

Contributing Editors
Michael Barr
John Canosa
Jack W. Crenshaw
Jack G. Ganssle
Dan Saks
Larry Mittag

Art Director
Debee Rommel
drommel@cmp.com

European Correspondent
Colin Holland
colin.holland@btinternet.com

Embedded.com Site Editor
Bernard Cole
bccole@acm.org

Production Manager
Pete Scibilia
pscibili@cmp.com

Director of Audiences Services
Kristi Cunningham
kcunningham@cmp.com

Subscription Customer Service
P.O. Box 2165, Skokie, IL 60076
(800) 577-5356 (toll free), Fax: (847) 763-9606
embeddedsystemsdesign@halldata.com
www.embeddedsystemsdesigncustomerservice.com

Back Issues
Kelly Minihan
(800) 444-4881 (toll free),
Fax: (785) 838-7566

Article Reprints, E-prints, and Permissions
PARS International Corp.
102 West 38th Street, Sixth Floor
New York, NY 10018
(212) 221-9595, Fax: (212) 221-9195
reprints@parsintl.com
www.magreprints.com.quickquote.asp

Publisher
David Blaza
(415) 947-6929
dblaza@cmp.com

Editorial Review Board
Michael Barr
Jack W. Crenshaw
Jack G. Ganssle
Bill Gatliff
Nigel Jones
Niall Murphy
Dan Saks
Miro Samek

CMP
United Business Media

Corporate

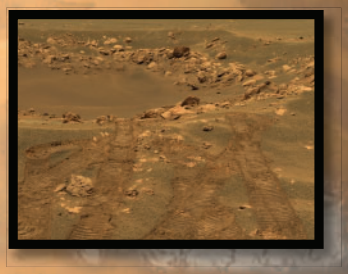
David Levin	*Chairman*
Scott Mozarsky	*Chief Financial Officer*
Tony Uphoff	*President, CMP Business Technology Group*
Robert Faletra	*President, CMP Channel*
Paul Miller	*President, CMP Electronics Group*
Philip Chapnick	*President, CMP Game, Dobb's, ICMI Group*
Anne Marie Miller	*Corporate Senior Vice President, Sales*
Marvlieu Jolla Hall	*Senior Vice President, Human Resources*
Marie Myers	*Senior Vice President, Manufacturing*
Alexandra Raine	*Senior Vice President, Communications*

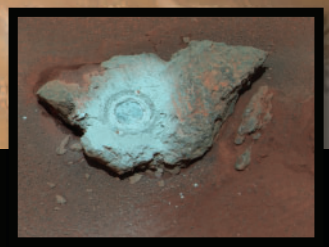
SMILE, MARS!

ThreadX® RTOS manages camera software critical to NASA mission

"We found ThreadX to be a proven solution based on its demonstrated success for the Deep Impact mission, so using it for the HiRISE instrument aboard the MRO was a logical decision. ThreadX delivered a first-rate performance for us and helped the MRO mission return extraordinary high-resolution images from Mars."

– Steve Tarr, HiRISE Software Lead, Ball Aerospace & Technologies Corp.





MRO spacecraft depicted in Mars orbit: NASA

Images courtesy NASA: http://mars.jpl.nasa.gov/mro

The Mission

When they wrote the embedded software that controls the cameras aboard the Mars Reconnaissance Orbiter (MRO), a team of Ball Aerospace and Technology Corp. engineers led by Steve Tarr knew they only had one chance to get it right. If there was a serious flaw anywhere in the software, the $720 million spacecraft might have no more value than a digital camera dropped in a bathtub. Tarr and his team wrote 20,000 lines of code and used Express Logic's ThreadX RTOS. The software has worked flawlessly, resulting in history-making photographs such as the one to the left that shows the Opportunity rover traversing the surface of Mars.

THREADX

The Technology

With its intuitive API, rock-solid reliability, small memory footprint, and high-performance, ThreadX delivered the goods for NASA's MRO. ThreadX is in over 450 million electronic devices from NASA's MRO to HP's printers and digital cameras. Which RTOS will you choose for YOUR next project?

Small Memory Footprint • Fast Context Switch • Fast Interrupt Response
Preemption-Threshold™ Technology • Picokernel™ Design • Event Chaining™
Broad Tools Support • Supports All Leading 32/64-bit Processors • Easy to Use
Full Source Code • Royalty-Free

Order today on Amazon
Real-Time Embedded Multithreading
Using ThreadX and ARM
by Edward L. Lamie

For a free evaluation copy, visit www.rtos.com • 1-888-THREADX

expresslogic

EMBEDDED

Unlock your future

Enter the New Era of Configurable Embedded Processing

Adapt to changing algorithms, protocols and interfaces, by creating your next embedded design on the world s most flexible system platform. With the latest processing breakthroughs at your fingertips, you can readily meet the demands of applications in automotive, industrial, medical, communications, or defense markets.

Architect your embedded vision

- *Choose MicroBlaze™ the only 32-bit soft processor with a configurable MMU, or the industry-standard 32-bit PowerPC® architecture*
- *Select the exact mix of peripherals that meet your I/O needs, and stitch them together with the new optimized CoreConnect™ PLB bus*

Build, program, debug . . . *your* way

- *Port the OS of your choice including Linux 2.6 for PowerPC or MicroBlaze*
- *Reduce hardware/software debug time using Eclipse-based IDEs together with integrated ChipScope™ analyzer*

Eliminate risk & reduce cost

- *No worry of processor obsolescence with Xilinx Embedded Processing technology and a range of programmable devices*
- *Reconfigure your design even after deployment, reducing support cost and increasing product life*

Order your complete development kit today, and unlock the future of embedded design.

Get the Complete Embedded Solution

www.xilinx.com/processor

XILINX®

www.xilinx.com/processor

At the Heart of Innovation

Remembering transistor history

Nice article (Jack Ganssle, "The transistor: sixty years old and still switching," December 2007, p. 53). One minor point: before the galena and cat's whiskers, there was the "coherer," a strange concoction of metal filings between two electrodes. It exhibited a large resistance drop when subjected to an RF signal from the antenna that would sound a bell and vibrate the device back to its high-resistance state (for CW only, of course, not telephony.) A Google search on "coherer" yields some fascinating references, including DIY coherers for the curious—they actually work! After the coherer came the "magnetic detector"—IIRC, a moving band of iron.

—Roger Jones

My father worked for RCA as an engineer from the '50s through the '70s. One year at the open house during the Christmas holidays, they gave us all an inhouse history of RCA. The founder of RCA, David Sarnoff, was a teenage telegrapher for the Marconi company and was on duty when *RMS Titanic* went down. He manned his key for many hours, compiling lists of the survivors, the missing, and the dead. This story, along with many others, encouraged me to enter this field. How far we have come in just the 60 years of the transistor and the 100 years or so of electronics? What will the next 100 years hold?

—Thomas Mazowiesky

While I was working at Bell Laboratories in the late 1970s, a story was going around about the invention of the transistor. It seems that one day Bill Shockley was sitting in his office reading a magazine when a technician ran in to his office screaming "Mr. Shockley! Mr. Shockley! I connected *p* material to *n* material to *p* material, injected some current, and I am getting amplification!" Shockley looked over the top of his magazine and said "Congratulations! You just discovered the Shockley Effect!"

—Ed Wozniak

"Congratulations! You just discovered the Shockley Effect!"

In 1968, I was a sophomore engineering physics major at the University of Illinois. Bardeen was teaching E&M. Articulate, lucid, and very friendly—he put the material at the right level and pulled us along through a course that was universally dreaded. Academia could use a few thousand more professors with his ability to teach and engage young minds.

—David Barr

It's not all about Linux

The statement that "an estimated 70% of new semiconductor devices are Linux-enabled" seems quite impossible (Hadi Nahari and Jim Ready, "Employ a secure flavor of Linux," October 2007, p. 20). A lot of LED and diodes are being made, even today. Even if "semiconductor devices" is changed to read "microprocessor," it seems very unlikely to be true, given the quantities of low-end controllers shipped in small gadgets. Can you point to any justification for this statement?

—Craig Cherry

It's not all about operating systems

Mr. Carbone could not possibly have an "ax to grind," could he? (John Carbonne, "Embedded OS trends points to Linux . . . sometimes," online Guest Editor column, 12/11/07.) After all, he hails from Express Logic, which sells the Thread-X RTOS. I have seen over a dozen surveys that show the exact opposite of the one cited in this article—where the use of Linux is on the rise in embedded devices and (especially) in small handheld devices that must display rich-content from the Internet.

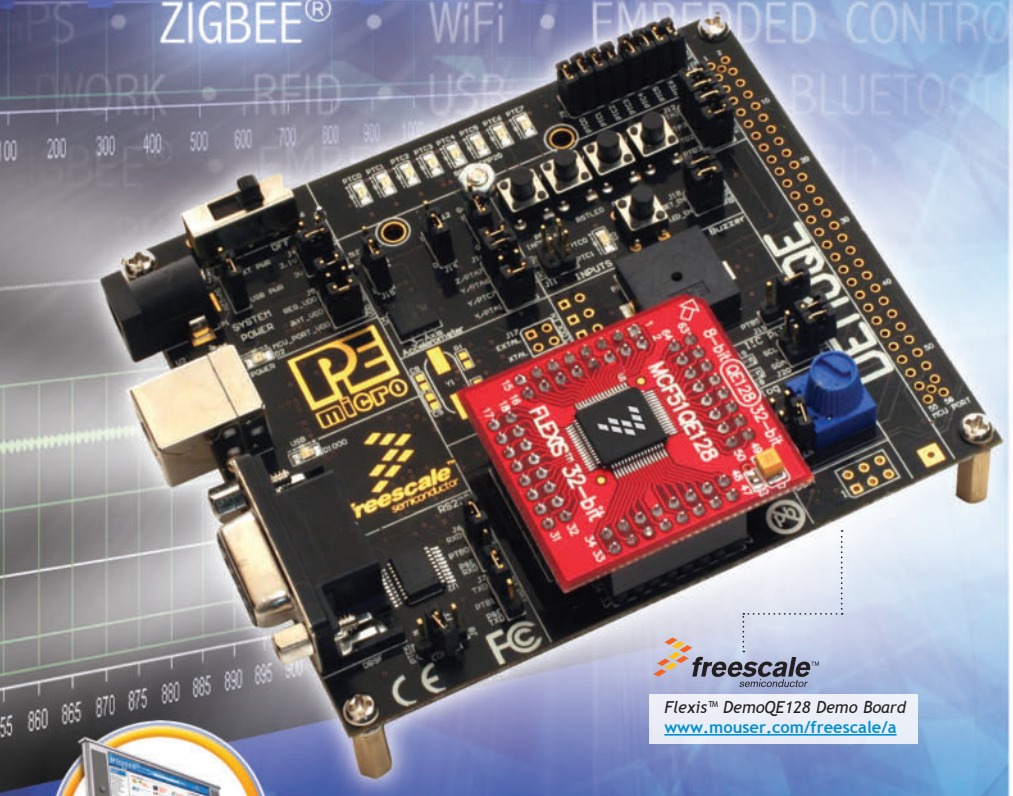
Linux as applied to embedded systems has a steep learning curve. Once you're past that, it's smooth sailing. But, Linux has its place. Personally, I like OpenBSD, because of its focus on correctness and security. OpenBSD will only run (at this time) on machines that have an MMU, so it will not fit well on certain desirable platforms. There's a place for assembly, for C, for a FORTH-based system, for a "home grown" RTOS, for a commercial RTOS, for a free RTOS, and of course for Linux (and its BSD sisters and brothers). Remember the old saying: "If all you have is a hammer, then every problem looks like a nail." One should not fall in love with any one technique.

Linux is not for tiny microcontrollers—you need at least a 32-bit machine with at least tens of megabytes of memory. (However, the trend in microcontrollers is to migrate to 32-bits and memory is becoming very inexpensive). Linux is great if what you want is a standard (and open) platform with extensive networking support, and lots of already written and ready-to-go standard applications. That's why Linux is becoming very popular on upscale cellular phones and handheld PCs. If all you need to do is read a sensor and send a packet somewhere over TCP/IP, there are better solutions, (and not all of them cost a lot of money like the Thread-X RTOS).

CONTINUED ON PAGE 43

The Newest Embedded Technologies

WiFi • USB • BLUETOOTH™ • WLAN • NETWORK • USB • RFID • WLAN • BLUETOOTH™ • EMBEDDED CONTROL • NETWORK • RFID • USB • WLAN • GPS • BLUETOOTH™ • BLUETOOTH™ • ZIGBEE® • USB • NETWORK • RFID • USB • WLAN • BLUETOOTH™ • ZIGBEE® • WiFi



freescale™ semiconductor

Flexis™ DemoQE128 Demo Board
www.mouser.com/freescale/a

New Products from:

VDIP1 Module
www.mouser.com/ftdi/a

FTDI Chip

RCM4000
RabbitCore® Modules
www.mouser.com/rabbitsemi/a

RABBIT Semiconductor

MatchPort™ b/g
Embedded Device Server
www.mouser.com/lantronix/a

LANTRONIX®

The ONLY New Catalog Every 90 Days

Experience Mouser's time-to-market advantage with no minimums and same-day shipping of the newest products from more than 335 leading suppliers.

MOUSER® ELECTRONICS
a tti company

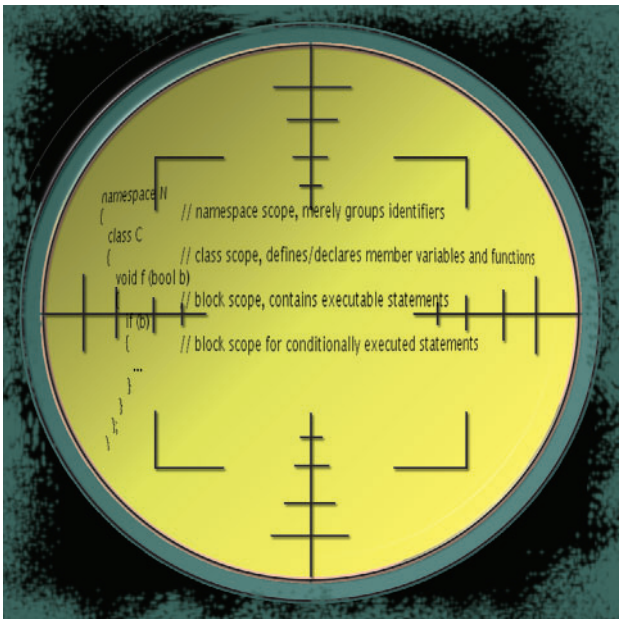
The Newest Products For Your Newest Designs

(800) 346-6873

www.mouser.com
Over 900,000 Products Online

BY Dan Saks

Storage class specifiers and storage duration



Storage class specifiers don't specify scope but combine with scope to determine storage duration. Here's the second part in a series on scope, storage allocation, and linkage.

A *declaration* is a source code construct that associates attributes with names. A declaration either introduces a name into the current translation unit or redeclares a name introduced by a declaration that appeared earlier in the same translation unit. A declaration might also be a *definition*, which provides not just some of the attributes of a name, but rather all the information the compiler needs to create the code for that name.

Among the attributes that a name may have are its type, scope, storage duration, and linkage. Not every name has all of these attributes. For example, a function name has a type, a scope, and a linkage, but no storage duration. A statement label name has only a scope.

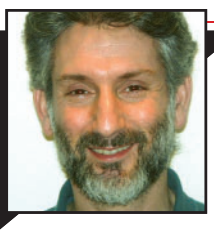
An object's *type* determines the object's size and memory address alignment, the values the object can have, and the operations that can be performed on that object. A function's *type* specifies the function's parameter list and return type. I've discussed the concept of data types in prior columns.[1, 2]

I devoted my November column to the concept of scope as it applies to C and C++.[3] In essence, the *scope* of a name is that portion of a translation unit in which the name is visible. C and C++ each support several different kinds of scope, summarized in the sidebar entitled "Scope regions in C and C++" (see page 10).

Scope is closely related to, but nonetheless distinct from, the concepts of storage duration and linkage. The *storage duration* for an object determines how and when the storage for that object comes and goes. *Linkage* determines whether declarations in different scopes can refer to the same object or function. It's easy to confuse these concepts because they're so intertwined.

Much of the confusion stems from the complex semantics of storage class specifiers, keywords such as `extern` and `static`. For example, the precise meaning of `static` depends on the scope in which it appears. Sometimes, declaring an object `static` affects the object's storage duration. It can also affect the object's linkage. Understanding these distinctions can help you program more effectively.

This month, I'll explain the syntax of storage class specifiers and the concept of storage duration in C and C++. I'll also show you how they're related to the concept of scope.



Dan Saks is president of Saks & Associates, a C/C++ training and consulting company. For more information about Dan Saks, visit his website at www.dansaks.com. Dan also welcomes your feedback: e-mail him at dsaks@wittenberg.edu.

STORAGE CLASS SPECIFIERS

Storage class specifiers are keywords you can use in decla-

SCOPE REGIONS IN C AND C++

C and C++ each support five different kinds of scope regions. Although the C and C++ standards use different names for some regions and different verbiage to define those regions, the two languages support essentially the same five regions:

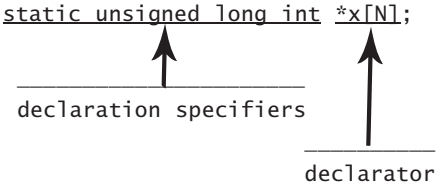
- In C, a name has *file scope* if it's declared in the outermost scope of a translation unit. C++ extends the concept of file scope to the broader concept of *namespace scope*. In C++, a name has namespace scope if it's declared either in a namespace definition or in what C calls file scope. The C++ standard refers to the C concept of file scope as *global namespace scope*, of just *global scope*.
- A name (other than a statement label) has *block scope* if it's declared within a function definition or a block nested therein.
- A name has *function prototype scope* if it's declared in the function parameter list of a function declaration that is not also a definition.
- Each statement label has *function scope*, which spans the body of the function containing the label.
- A name in C++ has *class scope* if it's declared within the brace-enclosed body of a class definition. Classes in C++ include structures and unions, so a member of a C++ structure or union has class scope as well. The C standard doesn't have a corresponding notion of structure scope, but rather says that each structure or union has a separate name space for its members. Despite the different verbiage in their respective standards, C and C++ look up structure and union members in much the same way.

rations to control storage duration and linkage. First I'll show you how they fit into the syntax. Then I'll explain their impact on semantics.

Every declaration in C and C++ has two principal parts: a sequence of zero or more declaration specifiers, and a sequence of zero or more declarators, separated by commas.

For example:

```
static unsigned long int *x[N];
```



A *declarator* is the name being declared, possibly surrounded by operators such as `*`, `[]`, `()`, and (in the case of C++) `&`. In the previous example, `*x[N]` is a declarator indicating that `x` is an "array of `N` pointers to ..."

Each object in C and C++ has one of the following three storage durations: static, automatic, and dynamic.

something, where that something is the type specified in the declaration specifiers.

A declarator may contain more than one identifier. The declarator `*x[N]` contains two identifiers, `x` and `N`. Only one of those identifiers is the one being declared and it's called the *declarator-id*. The other(s), if any, must have been declared previously. The declarator-id in `*x[N]` is `x`.

(The term *declarator-id* comes from the C++ standard. The C standard makes do without it, but I find it to be a useful concept.)

Some of the declaration specifiers leading up to a declarator can be *type specifiers* such as `int`, `unsigned`, `long`, `const`, or a user-defined type name. They can also be *storage class specifiers* such as `extern` or `static`, or *function specifiers* such as `inline`.

The type specifiers contribute to the type of the declarator-id; the other specifiers provide non-type information that applies directly to the declarator-id. For example:

```
static unsigned long int *x[N];
```

declares `x` as an object of type "array of `N` pointers to `unsigned long int`". The keyword `static` specifies `x`'s storage class.

NEW VERSION AVAILABLE

NI LabVIEW.

Limited Only by Your Imagination.

Build and program robots with LEGO® MINDSTORMS® NXT using software powered by NI LabVIEW

Communicate via multiple protocols including Bluetooth

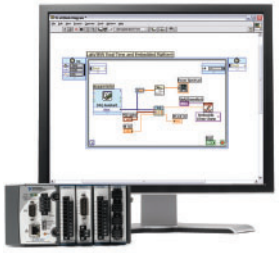
Graphically program concurrent, real-time applications

Develop your human machine interface (HMI) display

Independently control multiple servo motors

Target 32-bit microprocessors and FPGAs

Real-Time and Embedded | Signal Processing | High-Performance Test | Industrial Control



PRODUCT PLATFORM

LabVIEW Real-Time Module

LabVIEW FPGA Module

LabVIEW Microprocessor SDK

NI CompactRIO Embedded Hardware Platform

When the LEGO Group needed parallel programming and motor control tools intuitive enough for children, it selected graphical software powered by NI LabVIEW. With LabVIEW graphical system design, domain experts can quickly develop complex, embedded real-time systems with FPGAs, DSPs, and microprocessors.

>> Expand your imagination with technical resources at **ni.com/imagine** **866 337 5041**

NATIONAL INSTRUMENTS

The C standard lists five storage class specifiers: `auto`, `extern`, `register`, `static`, and `typedef`; however, C considers `typedef` to be a storage class specifier for syntactic convenience only. C++ doesn't consider `typedef` as a storage class, so I won't either.

The C++ standard lists `mutable` as another storage class specifier, but this, too, is more for syntactic convenience than anything else. Unlike the other storage class specifiers, `mutable` has no impact on storage duration or linkage. I don't consider it a storage class specifier for the purpose of this discussion.

A declaration need not have any storage class specifier and can have no more than one.

Allocating storage for an object with static storage duration typically costs nothing at run time . . .

STORAGE DURATION

The *storage duration* of an object determines the lifetime of the storage for that object. That is, it determines that part of program execution during which storage for the object must exist. Programmers often use the term *storage allocation* instead of *storage duration*, but both the C and C++ standards favor the latter. Only objects have storage duration. Enumeration constants, functions, labels, and types don't.

Each object in C and C++ has one of the following three storage durations: static, automatic, and dynamic. (The C standard lists the third kind of storage duration as "allocated" rather than "dynamic" but then never uses the term after that. I'll call it dynamic.)

An object declared at file scope (in C) or namespace scope (in C++), or declared with the storage class specifier `extern` or `static`, has *static storage duration*. The lifetime of the storage for that object is the entire time that the program is executing.

An object declared at block scope, and without the storage class specifier `extern` or `static`, has *automatic storage duration*. The lifetime of the storage for that object begins upon entry into the block immediately enclosing the object's declaration and ends upon exit from the block. Entering an enclosed block or calling a function suspends, but doesn't end, the execution of a block.

When a program allocates storage for an object by calling an allocation function, such as `malloc` in C or an `operator new` in C++, that object has *dynamic storage duration*. The lifetime of the object's storage lasts until the program passes the address of that object to a corresponding deallocation function, such as `free` in C or an `operator delete` in C++.

Table 1 shows how C and C++ determine the storage duration for an object based on the storage class specifier in the object's declaration and the scope in which the declaration appears. For example, the first row (below the column headings) says that an object declared with no storage class specifier at block scope has automatic storage duration, but if it appears at file scope in C or at namespace scope in C++, it has static storage duration. If it appears as a structure or class member, then it has the storage duration of the structure or class object of which it's a member.

None of the entries in Table 1 specify dynamic storage allocation. Unlike objects with static or automatic storage duration, a program can't *declare* any objects with dynamic storage duration. A program can *create* them by calling an allocation function; it just can't declare them.

THE MECHANICS OF STORAGE ALLOCATION

The exact manner in which static storage is allocated and deallocated depends on the target platform. However, allocating storage for an object with static storage duration typically costs nothing at run time because the compiler, linker, and loader together determine the size and address of the object before the program starts running.

Storage duration for objects in C and C++.

Storage class specifier	At block scope	At file scope (in C) or namespace scope (in C++)	As a structure member (in C) or at class scope (in C++)
none	automatic	static	storage allocated as part of enclosing object
`auto`	automatic	*invalid*	*invalid*
`extern`	static	static	*invalid*
`register`	automatic	*invalid*	*invalid*
`static`	static	static	*invalid* in C; static in C++

Table 1

From the running program's perspective, an object with static storage duration is always there.

Typical C and C++ programs allocate automatic storage on a run-time stack, often the same stack that they use for storing function-call return addresses. Allocating storage for a local object isn't free, but it's usually dirt cheap—just one machine instruction. For example, in:

```
int foo(int v)
     {
     int m;
     ...
     return m;
     }
```

function `foo` has a single local object, `m`. The compiler determines `m`'s size from its type, typically 2 or 4 bytes. When it compiles `foo`, the compiler simply generates an instruction such as:

```
sub sp, 4
```

as one of the first instructions in the function body to carve room for an `int` on the stack. (This example assumes that an `int` object occupies 4 bytes and that the stack grows downward from higher addresses to lower addresses.)

Allocating automatic storage for several local objects costs more stack space, but no more run time, than allocating storage for just one. For example, in:

```
int foo(int v)
     {
     int m;
     double d;
     ...
     return m + n;
     }
```

the function has two local objects, `m` and `d`. In this case, when it compiles the function, the compiler determines the size of `m`, still 4, and the size of `d`, say 8. Rather than generate a separate instruction to allocate storage for each object, the compiler simply adds up the sizes and uses the sum in a single instruction, such as:

```
sub sp, 12
```

A function may also declare local objects in nested blocks. For example, in:

```
int foo(int n)
     {
     char *p;
     ...
     if (p != NULL)
          {
          int v;
          ...
          }
     return n;
     }
```

Allocating automatic storage for several local objects costs more stack space, but no more runtime, than allocating storage for just one.

function `foo` has a block nested within the if statement. That block declares a local object `v`. In this case, the lifetime of the storage for `v` begins upon entry into the nested block and ends upon exiting the block. However, many compilers will generate code for `foo` to allocate the storage for `v` along with all the other local objects upon entering the function and deallocate the storage for `v` upon exiting `foo`. Thus, a compiler might generate code that extends the actual lifetime of the storage for a local object, but it's very hazardous for programs to try to exploit these longer lifetimes.

Dynamic allocation is typically much slower than automatic allocation. It often involves executing tens of instructions, possibly more than a hundred. Nonetheless, you can use it to manage memory very economically, and so it may be worth the price.

LINKAGE ON THE HORIZON

As I mentioned earlier, not only can a declaration specify type, scope, and storage duration, it can also specify linkage. I thought linkage would be the subject of this column until I started writing and realized that I needed to cover storage duration first. I'll get there yet. ■

ENDNOTES:

1. Saks, Dan, "A New Appreciation for Data Types," *Embedded Systems Programming*, May, 2001, p. 59.
2. Saks, Dan, "Cast with caution," *Embedded Systems Design*, July, 2006, p. 15.
3. Saks, Dan, "Scope regions in C and C++," *Embedded Systems Design*, November, 2007, p. 15.

The art of FPGA construction

BY GINA R. SMITH

Working with FPGAs isn't intimidating when you know the basic techniques and options.

Over the last several years, the use of FPGAs has greatly increased in military and commercial products. They can be found in primary and secondary surveillance radar, satellite communication, automotive, manufacturing, and many other types of products. While the FPGA development process is second nature to embedded systems designers experienced in implementing digital designs on an FPGA, it can be confusing and difficult for the rest of us. Good communication is important when technical leads, supervisors, managers, or systems engineers interface with FPGA designers.

The key to good communication is having an understanding of the development process. A solid understanding will help you comprehend and extract relevant information for status reports, define schedule tasks, and allocate appropriate resources and time. There have been many times when my FPGA knowledge has allowed me to detect and correct errors, such as wrong part numbers or misuse of terms and terminology found in requirements and other documents.

Regardless of your final product, FPGA designers must follow the same basic process. The FPGA development stages are design, simulation, synthesis, and implementation, as shown in Figure 1. The design process involves converting the requirements into a format that represents the desired digital function(s). Common design formats are schematic capture, hardware description language (HDL), or a combination of the two. While each method has its advantages and disadvantages, HDLs generally offer the greatest design flexibility.


CAUTION
CAUTION

The FPGA development process can be divided into four functions: design, synthesis, simulation, and implementation.

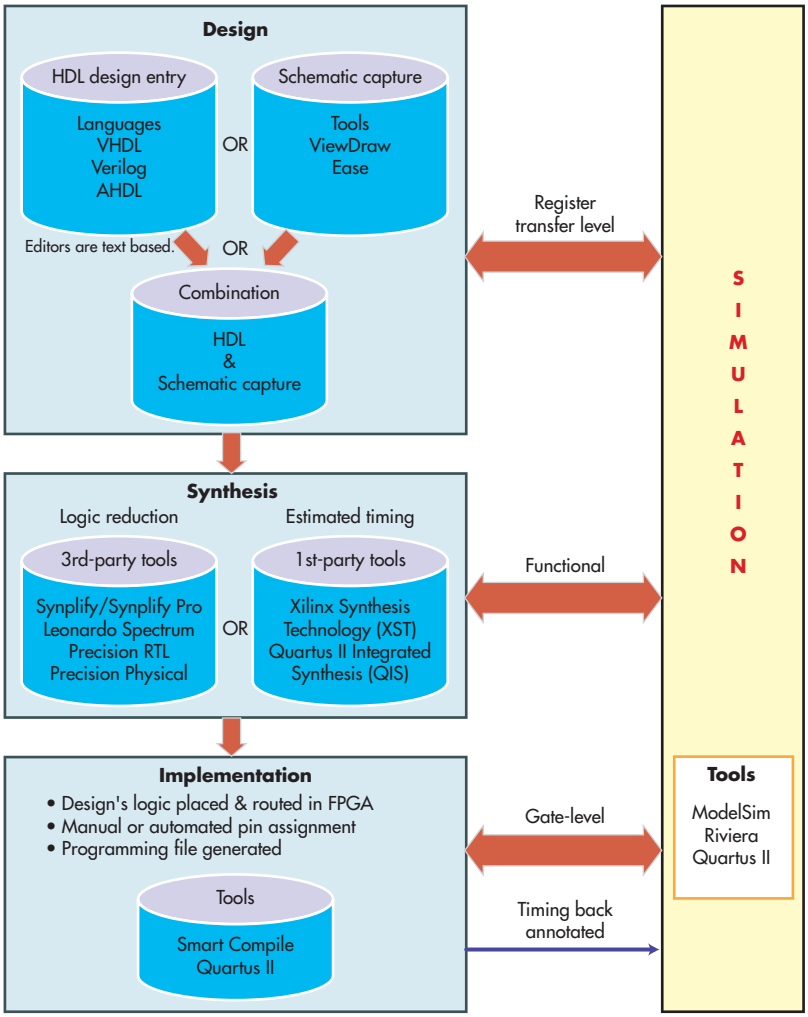


Figure 1

SCHEMATIC CAPTURE

Schematic capture, the graphical depiction of a digital design, shows the actual interconnection between each logic gate that produces the desired output function(s). Many of these logic-gate symbols involve proprietary information making them available to the designer only through the specific vendor's component library. Schematic capture designs that mainly consist of proprietary symbols make the design unrecognizable by competitors' FPGA development tools. The proprietary nature of this type of design makes it vendor dependent, and the entire design process must be repeated if a different vendor is used.

Examples of schematic capture tools are Viewlogic's ViewDraw and HDL's EASE. The main advantage of schematic capture is that the graphical representation is easy to understand. However, its major drawback is an increase in cost and time to reproduce a design for different vendors due to the design's proprietary nature.

HDL METHOD

Hardware description languages (HDLs) use code to represent digital functions. "Firmware" often refers to the resulting HDL code. HDLs are a common and popular approach to FPGA design. You can create the source code with any text editor. Special HDL editors like CodeWright and Scriptum (a free HDL text editor by HDL Works) offers features such as HDL templates and highlighting reserved words not found in ordinary text editors. HDLs can be generic (supported by multiple simulation and synthesis tool sets) like Verilog or VHDL (Very High Speed IC HDL), or vendor-specific like Altera's Hardware Description Language (AHDL), which is only recognizable by Altera's design tool set.

There are two writing styles for HDL designs, structural or behavioral. *Structural firmware* is the software equivalent of a schematic capture design. Like schematic capture, structural designs instantiates or uses vendor-specific components to construct the desired digital functions. This type of HLD firmware is vendor-dependent like its graphical counterpart and has the same disadvantages. Like schematic capture designs, repeating the design process is necessary for different vendors.

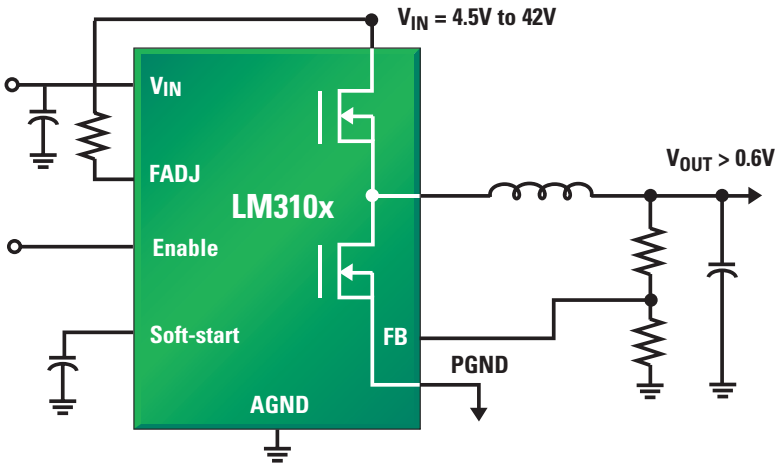
Behavioral HDL firmware describes digital functions in generic or abstract terms that are generally vendor independent. This provides enough flexibility for code reuse in different vendor's FPGAs so little or no code modification is required. Advantages of behavioral designs are its flexibility and time and cost-savings, and it offers little to no vendor dependence. For designs that require vendor specific resources, such as RAM, only those components must change for different vendors.

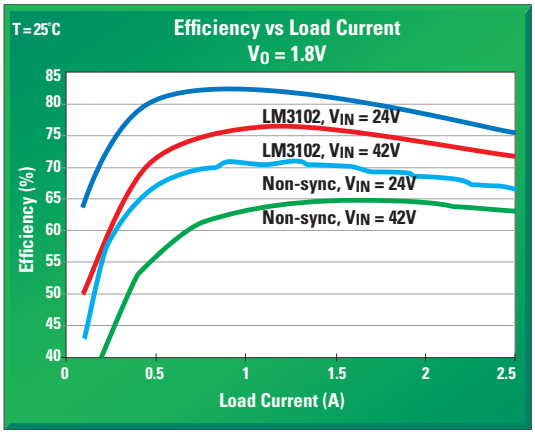
VHDL and Verilog are the most popular HDL languages. VHDL files consist of three main parts: *library declaration*, *entity declaration*, and *architecture section*. While not required by VHDL, an optional heading section should be included. This section should contain pertinent information, such as the designer's name, filename, a brief summary of the code, and a re-

2.5A, 42V SIMPLE SWITCHER® Synchronous Step-Down Regulators

national.com/switcher

Constant-on-Time (COT) LM310x Regulators from the PowerWise® Family Need No Loop Compensation and Are Stable with Ceramic Capacitors





Product ID	V_{IN} Range (V)	Current (A)	V_{FB} (V)	Frequency (MHz)	Packaging
LM3100	4.5 to 36	1.5	0.8	Up to 1	eTSSOP-20
LM3102	4.5 to 42	2.5	0.8	Up to 1	eTSSOP-20
LM3103	4.5 to 42	0.75	0.6	Up to 1	eTSSOP-16

LM310x Features

- COT control provides lightning-fast transient response
- Stable with ceramic capacitors
- Near-constant frequency operation from unregulated supplies
- No loop compensation reduces external component count
- Pre-bias startup
- Discontinuous Conduction Mode (DCM) operation for a light load
- Enabled in National's WEBENCH® online design environment

Applications

Embedded systems, industrial controls, automotive telematics and body electronics, point-of-load regulators, storage systems, and broadband infrastructure

For FREE samples, datasheets, and online design tools visit:

national.com/switcher

Or call: 1-800-272-9959

National Semiconductor

The Sight & Sound of Information

Listing 1 The various sections of a VHDL source file are illustrated here.

Optional heading section

```
--************************ Header Section ***********************************************
--  Name                :       Beckie Smith
--  Date                :       January 28, 2005
--  Filename            :       Door_monitor.vhd
--
--  Description:
--  This circuit is responsible for enabling an external door chime circuit 10 clock
--  cycles or about 500ns after door_status goes high.
--
--  Revision History
--  Date            Initials      Description
--  2-17-05           BCS         Changed chime delay from 1 minute to 500ns.
--
--
*******************************************************************************************
```

Library declaration

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE ieee.std_logic_signed.all;
```

Entity declaration

```
ENTITY monitor IS PORT(
  reset                     : IN std_logic;           -- set internal gates to initial state
  door_status               : IN std_logic;           -- closed = low
  clock_20mhz               : IN std_logic;           -- input clock_20mhz 20HZ
  door_chime_en             : OUT std_logic);         -- signal used to sound door chime
END monitor;
```

Architecture section

```
ARCHITECTURE door_monitor OF monitor IS
  SIGNAL start_500ns_timer   : std_logic;                    -- enables 500ns timer
  SIGNAL reset_start_timer   : std_logic;                    -- reset 500ns timer
  SIGNAL timer_500ns         : std_logic_vector(3 DOWNTO 0); -- 500ns counter
  SIGNAL chime_enable        : std_logic;                    -- sets external door chime

BEGIN
  door_chime_en             <= chime_enable;          -- signal used to sound door chime

checking_door_status:  PROCESS (reset, clock_20mhz, reset_start_timer)
-- This process detects when the door has been opened and then starts the
-- 500ns timer.
BEGIN
  IF (reset                  = '1') OR (reset_start_timer = '1') THEN
    start_500ns_timer       <= '0';                          -- clear 500ns timer
  ELSIF(rising_edge (clock_20mhz)) THEN
    IF door_status           = '1' THEN                      -- door is opened
      start_500ns_timer     <= '1' after 2 ns;               -- enable 500ns timer
    END IF;
  END IF;
END PROCESS;

set_alarm_enable: PROCESS (reset, clock_20mhz)
-- This process set the alarm enable 500ns after the door has been opened
BEGIN
  IF reset                   = '1' THEN
    chime_enable            <= '0';                          -- clear chime enable
    reset_start_timer       <= '0';                          -- clear timer reset signal
    timer_500ns             <= (OTHERS => ('0'));            -- clear chime timer
  ELSIF (rising_edge (clock_20mhz)) THEN
    IF start_500ns_timer     = '1' THEN                      -- door has been opened
      timer_500ns           <= timer_500ns + '1' after 2 ns; -- start counting 500nsec
      IF timer_500ns         = "1010" THEN                   -- test for 500ns
        chime_enable        <= '1' after 2 ns;               -- set door chime enable
        reset_start_timer   <= '1' after 2 ns;               -- reset door chime timer
      END IF;
    ELSE
      chime_enable          <= '0' after 2 ns;               -- don't set door chime circuit
    END IF;
  END IF;
END PROCESS;
END door_monitor;
```

Concurrent statement

Sensitivity list

Sequential statements

Injected delay

vision history. Listing 1 shows an example of a VHDL file's behavior. Because HDLs are similar to software, firmware designers should follow some of software development rules.

HDL GUIDELINES

1. Use comments to provide code clarity.
2. Indicate active low signals by `n`, `_n`, `_b`, `*`at the end of the name.
3. Signal names should be relatively short but descriptive. For example:
   - A good signal name would be `CEn` for an active low chip enable.
   - A bad signal name would be `active_low_chip_enable`.
   - Use underscores in name description for clarity.
   - Synchronize signals to change on a clock edge.
   - Process, routes, modules, and so forth, should perform a single function.
   - Use formatting, such as tabs and spaces, to provide readability of code.
   - Include a header section for each file or module. Suggestive header information designer's name, file description, and revision or history record.

VHDL SYNTAX RULES

Now for some VHDL specifics, including data types:

- `Std_logic` can have values of high 1, low 0, unknown *X*, uninitialized *U*, high impedance *Z*, weak unknown *W*, weak 0 *L*, weak 1 *H*, and don't care - to represent a single data bit.
- `Std_logic_vector` can have the same values as `std_logic`; however it represents multiple bits.
- A bit can only have a value of high 1 or low 0, and it represents one data bit.
- Boolean represents true or false.
- Comments are denoted by double dash marks `--`.
- Comments continue after `--` until a carriage return.
- Each statement ends with a semicolon `;` .
- VHDL is not case sensitive.
- No specific format is required.
- Reserved words aren't valid signal names.
- Signal names must start with a letter; numbers are not acceptable.

Library declaration

The *library declaration* is the first section in the source file. This is where you place the library and package callout statements. *Libraries* and *packages* define and store components, define signal types, functions, procedures, and so forth. Packages and libraries are standardized, such as the IEEE library, and defined by a user (designer) or vendor. The IEEE library offers several packages, such as standard, textio, and std_logic_1164. Each of these packages defines various types, attributes, procedures, files, and so on. Here's an abbreviated list of selected IEEE packages:

When a designer has specific constants, formulas, processes, and procedures that are used by multiple modules or submodules within their design, he or she can create a custom package.

- *standard* defines types (such as boolean, bit, time, and integer), subtypes (such as natural and positive), and the attribute `foreign`.
- *textio* package defines types (such as line and text), files (such as input and output), and procedures (such as read, readline write, and writeline).
- *Std_logic_1164* package defines types (such as std_ulogic and std_ulogic_vector) and functions (such as nand, and, or, nor).

The *work library* serves as a place to add or delete designs. Designs stored in the work library get analyzed during synthesis and simulation. Various tools handle libraries in different ways. Therefore, users should consult the tool's documentation for correct use. To use what's in a library or package, the library must be made visible by using the keywords `Library` and `Use` clause. The IEEE std_logic_1164 package contains the types used in Listing 1. Therefore, the `LIBRARY IEEE;` statement makes it visible and `USE IEEE.std_logic_1164.all;` tells the tools to use all the contents in the std_logic_1164 package.

When a designer has specific constants, formulas, processes, and procedures that are used by multiple modules or submodules within their design, he or she can create a custom package. By doing this, the functions in the user-defined package can be shared with other designers and projects. A *user-defined library/package* is an easy way to repeatedly use specific HDL elements in multiple files with the luxury of only defining its elements once. Assuming a designer creates a package called my_package and stores this package in the library called `Test`, the following command would make the package visible, thereby allowing its contents to be used in the source file.

```
LIBRARY Test;
USE Test.my_package.ALL;
```

User/designer-defined packages are similar to those supplies by vendors, such as Xilinx, whose packages contain elements such as RAMs, counters, and buffers. Xilinx's "vcomponents" package contains constants, attributes, types, and components that become available once the library and package are visible to the design. The package contains components like AND3, which is a three-input AND gate, and NAND3, a three-input NAND gate. The "vcomponent" package provides timing information, the I/O port names (used to instantiate components in design),

Multiple source files are created for each function and are interconnected through a hierarchical file structure.

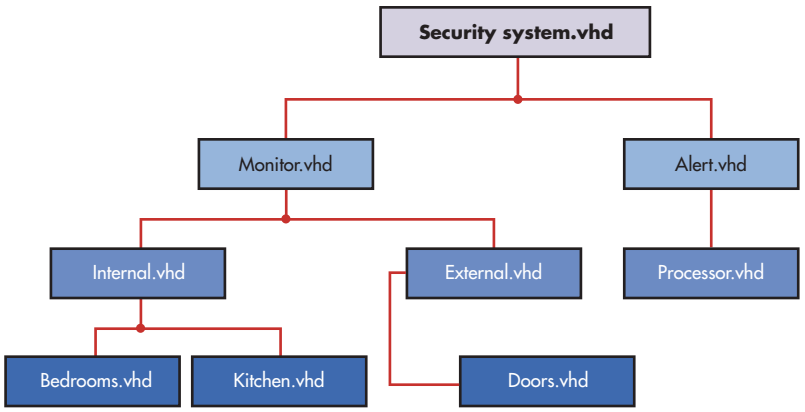


Figure 2

and other information used by synthesis and simulation tools. The vendor's package becomes visible in the same manner as the standard and user-defined libraries. To use the elements in Xilinx's "vcomponent" package, designers must make the library visible. For example, the following command makes the "Xilinx" library with vcomponent package visible to the design:

```
LIBRARY Xilinx;
USE Xilinx.vcomponents.ALL;
```

Once all the libraries and packages are visible, this section is complete.

Entity declaration

The *entity declaration section* immediately follows the library declaration. Each entity has an assigned name; Monitor is the entity name of the VHLD code in Listing 1. Just as the library declaration section makes libraries and packages visible to the design, the entity section makes the I/Os visible to other source files and the design and can represent the I/Os as physical FPGA pins. VHDL designs can contain one source file or a hierarchy of multiple files. *Hierarchical file structures* consist of several files connected through the signals declared in their entities. Figure 2 shows a simplified hierarchical file structure for a home security system.

> . . . if the design is only one file, the top-level entity declaration defines all of the I/O that represents physical FPGA pins.

On the other hand, if the design is only one file, the top-level entity declaration defines all of the I/O that represents physical FPGA pins. All I/O signals defined in this section must have unique names, indicated signal direction (input or output), and number of bits reserved for the signal. From Listing 1, `reset` is an input, only one data-bit wide and is a `std_logic` data type. The keyword `END` followed by the entity's name signifies the end of the entity. All entities must be associated with an architecture section.

Architecture section

The *architecture section*, which contains the circuit description, is the body of the VHDL source code. The libraries, packages, and signals work together to develop the desired functions. Like the entity, each architecture must have an assigned name. The format for declaring the architecture is the reserved word `Architecture` followed by its name `Door_monitor`, then the reserved word `Of`, then the entity's name `Monitor`. Signals not defined in the entity section are defined in this section.

The *signal assignment format* consists of the reserved word `Signal` followed by the signal name and then the data type (such as `std_logic` and `std_logic_vector`), as in Listing 1. Like names defined in the entity, each signal name must be unique and have a data type. This section is also for declaring constants, variables, and other data types.

Signals can be thought of as wires used to connect functions and store values. After defining all the design's signals, the designer is ready to develop the code that describes the desired functions. The reserved word `Begin` signifies the start of the next subsection, which combines the concurrent and sequential statements. *Concurrent statements* update or change value at anytime. The signal assignment immediately following the first reserved word `BEGIN` in Listing 1 is an example of a concurrent statement. *Sequential statements* update or change value when signals in the sensitivity list (see Listing 1) change state. Signals in "processes" are sequential statements. Most processes have a sensitivity list, process name, and circuit description (HDL code) between reserve words `BEGIN` and `END PROCESS`. The process name precedes the reserved word `Process`, and the sensitivity list is enclosed in the parenthesis.

Listing 1 contains two processes. The first is `checking_door_status`, which has a sensitivity list that contains three signals: `reset`, `clock_20mhz`, and `reset_start_timer`. The second process is `set_alarm_enable`, which

KEIL™
An ARM® Company

Microcontroller Development Tools

ARM Microcontroller Solution

ARM Powered® Microcontrollers – available from many silicon vendors; offer high computing performance along with rich peripherals. Turn ARM Microcontrollers into your solution for cost-sensitive powerful applications – **with Keil Development Tools**.

C/C++ Development Kit

The RealView **Microcontroller Development Kit** (MDK) is the complete software development environment for ARM7/9 and Cortex-M1/M3.

MDK is easy to learn and use, yet powerful enough for the most demanding embedded ARM application. The integrated Device Database® simplifies tool configuration and includes more than 250 ARM Powered Microcontroller variants for your embedded project.

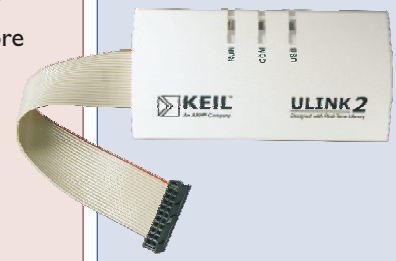
RealView® Microcontroller Development Kit

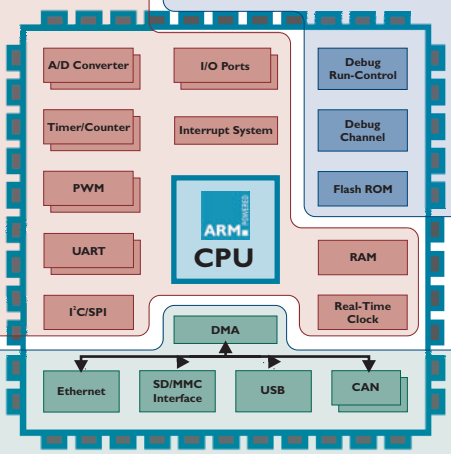
- RealView C/C++ Compiler
- RTX RTOS Kernel Library
- µVision Device Database & IDE
- µVision Debugger & Analysis Tools
- Complete Device Simulation

Examples and Templates

JTAG Debugger

ULINK2® connects to the JTAG or 2-wire debug interface and supports on-the-fly debugging and Flash programming.





RealView® Real-Time Library

- RTX RTOS Source Code
- TCPnet Networking Suite
- Flash File System
- USB Device Interface
- CAN Interface

Examples and Templates

RTOS and Middleware

The RealView **Real-Time Library** (RL-ARM) solves the real-time and communication challenges of your ARM project and expands **MDK** with essential components for sophisticated communication and interface peripherals.

Learn more about RealView **MDK**, **RL-ARM**, and **ULINK2**. Download a free evaluation version from www.keil.com/demo or call **1-800-348-8051**.

Cx51

Keil Cx51 is the de-facto industry standard for all classic and extended 8051 device variants. C51 Version 8.5 includes the latest devices such as XC800, ADE7169, and C8051F4xx - F6xx.

More information: www.keil.com/c51

C166

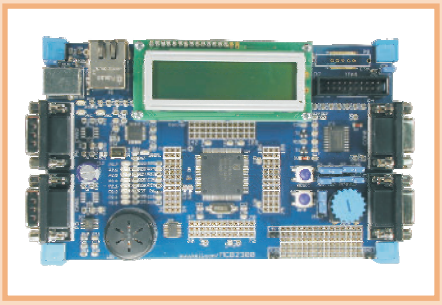
Keil C166 is the complete software development environment for Infineon C166, XC166, XC2000 and ST Microelectronics ST10 with debug and programming support via ULINK2.

More information: www.keil.com/c166

ARM

RealView MDK combines the best-in-class ARM C/C++ Compiler, the genuine Keil µVision IDE/Debugger/Simulator, and the royalty-free RTX RTOS Kernel.

More information: www.keil.com/arm



Keil MCB evaluation boards come with code size limited tools and extensive example projects that help you get up and running quickly with your own embedded application.

More information: www.keil.com/boards

www.keil.com

Listing 2 VHDL Testbench is used to provide stimulus to the VHDL source code.

Optional heading section

```
--************************ Header Section ******************************************
--   Name                 :            Beckie Smith

--   Date                 :            January 28, 2005

--   Filename             :            tb_door_monitor.vhd
--
--   Description:
--   This testbench is used to verify door_chime_en signal is set high 500ns after
--   door_status goes high.
--
--   Revision History
--   Date                 Initials     Description
--
--**********************************************************************************
```

Library declaration

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

Entity declaration

```
  ENTITY testbench  IS
  END testbench;
```

Architecture section

```
ARCHITECTURE tb_monitor OF testbench IS
COMPONENT monitor  PORT(
   reset                : IN std_logic;            -- power on reset
   door_status          : IN std_logic;            -- door closed = low
   clock_20mhz          : IN std_logic;            -- 20MHz clock
   door_chime_en        : OUT std_logic);          -- external door chime enable
END COMPONENT;

SIGNAL reset            : std_logic   := '1';      -- reset initially set high
SIGNAL door_status      : std_logic   := '0';      -- door initially closed
SIGNAL clock_20mhz      : std_logic   := '0';      -- 20MHz clock starts low
SIGNAL door_chime_en    : std_logic;

CONSTANT clock_20mhz_time  : time       := 25.0 ns;  -- half 20MHz clock period

BEGIN
display: monitor
PORT MAP (
   reset                => reset,
   door_status          => door_status,
   clock_20mhz          => clock_20mhz,
   door_chime_en        => door_chime_en);

reset                   <= '0' AFTER 50.00 ns;

create_clk:  PROCESS
-- This process generates the 20MHz input clock
BEGIN
WAIT FOR clock_20mhz_time;
   clock_20mhz          <= NOT clock_20mhz;
END PROCESS;

door_status             <= '1' AFTER 200.00 ns;    -- door is opened
END tb_monitor;
```

only has two signals, `reset` and `clock_20mhz`, in its sensitivity list. Signals in a process that update or change following a clock edge are called synchronous signals. `Start_500ns_timer` in the `checking_door_status` process is an example of a synchronous signal. The architecture section closes by using the reserved word `END` followed by the architecture's name.

SIMULATE OR SYNTHESIZE

One or more designers may be responsible for a design. A number of factors influence the numbers designers needed, such as design complexity and size; the designers' skill level; and the designers' schedule and availability. Regardless of the number of designers, after the design is completed, there are a couple of options. He or she may choose to simulate or synthesize the design. There isn't a hard and fast rule stating you must simulate before syn-

thesis. There are advantages to each option, and designers must determine which step is most beneficial. In fact, there may be times when a designer decided to simulate following the completion of the initial design while another time decide to synthesize. Each option lets the designer detect and correct different types of errors.

Simulating the design prior to synthesis allows logic errors and design flaws to be resolved early in the development process. Synthesizing lets the designer resolve synthesis errors prior to logic errors and design flaws. Ideally, the designer would perform minimal simulation, leaving the more stringent testing to a code tester. The original code designer shouldn't test his own code because he's less likely to detect specific design flaws such as:

1. Misinterpretation of requirements; if the designer misunderstood a requirement, he or she will test and evaluate the design based on that misunderstanding.
2. It's more difficult for a person to find his own errors. A third-party generally tests the code more rigorously and is more eager to find bugs than the original designer.

Regardless of who performs the simulations, the process is the same. For the sake of this article, we're going to assume the testing is performed by a code tester, not the original designer.

Simulation is the act of verifying the HDL or graphical digital designs prior to actual hardware validation. The circuit's input-signal characteristics are described in HDL or in graphical terms that are then applied to the design. This lets the code tester observe the outputs' behavior. It may be necessary to modify the source code during simulation to resolve any discrepancies, bugs, or errors.

Simulation inputs or *stimulus* are inputs that mimic realistic circuit I/Os. Stimulus forces the circuit to operate under various conditions and states. The greatest benefit of stimulus is the ability to apply a wide range of both valid and invalid input-signal characteristics, test circuit limits, vary signal parameters (such as pulse width and frequency), and observe output behavior without damaging hardware. Stimulus can be applied to the design in either HDL or graphical/waveform format. Generally, when a tester or designer speaks of a testbench, he's referring to applying stimulus to the design in the form of HDL. Listing 2 shows an example of a VHDL stimulus or testbench file.

The testbench looks similar to the actual VHDL design. Hence, the same VHDL language rules apply. Each tester has a style in which he or she writes a testbench, which can be automatic or manual and can use external files for simulation and analysis. Automatic testbenches can analyze simulation data and provide a final result, output error data, or other important information. Manual testbenches require the tester to manually analyze the data. An example of an automatic testbench would be one that reads valid data from an external file, compares it with simulation data, and writes the final pass/fail results to an external file. External files are useful for duplicating events seen on actual hardware.

Ideally, the designer would perform minimal simulation, leaving the more stringent testing to a code tester. The original code designer shouldn't test his own code because he's less likely to detect specific design flaws

Data can be taken from the hardware, stored in an external file, then read into a testbench and used as the input stimulus. Many simulators accept both waveform and testbenches as input stimulus; consult your simulator user's manual for acceptable formats. Some popular simulators are Mentor Graphics' ModelSim, Aldec's Riviera, and Altera's Quantus II.

There are three levels of simulation: register transfer level (RTL), functional, and gate level. Each occurs at a specific place in the development process. RTL follows the design stage; functional follows synthesis and after implementation is completed the gate level simulation. Generally, the stimulus developed for the RTL simulation is reusable without modification for each level of simulation.

SIMULATION

The initial simulation performed immediately after the design stage is the *RTL simulation*. This involves directly applying the stimulus to the design. RTL simulation only lets designers verify that the logic is correct. No realistic timing information is available to the simulator. Therefore, no serious timing exists for the design. The only timing information that can be available to the simulator is tester generated. Much like input stimulus, a tester can insert simulated or injected delays into the original HDL design, as in Listing 1. Most synthesis tools (discussed later) will ignore these simulated delays.

Applying test stimulus to the synthesized or optimized netlist produced by a synthesis tool is a *functional simulation*. Optimized netlists produced by non-vendors apply estimated delays that produce more realistic simulation output results. The main benefit from performing functional simulation is that it lets the tester verify that the synthesis process hasn't changed the design. Many, but not all, third-party simulation tools accept post-synthesis netlists.

Gate-level simulation involves applying stimulus to the netlist created by the implementation process. All internal timing delays are included in this netlist, which provides the tester with the most accurate design output. Again, many, but not all, third-party

simulation tools can perform gate simulation.

Ideally, each level of simulation is performed at the appropriate development stage. However, if this isn't possible, it's recommended that at a minimum, RTL is performed. As this simulation is performed, it's normal for the original design to require modifications due to logic errors. Each simulation level offers various benefits. RTL uncovers logic errors, the functional level verifies that the pre- and post-synthesis designs are equivalent, and the gate level uncovers timing errors.

Some benefits to spending sufficient time generating quality testbenches and simulation are reduced time troubleshooting hardware (generally, cheaper to testbench troubleshoot than hardware troubleshoot) and a decrease in the chance of damaging hardware resulting in a faster time to market. Opting to omit simulation and testbenching will generally cost the project additional time and money. Lab testing requires collecting and setting up test equipment (such as a logic analyzer and oscilloscope) and depending on the equipment used, the designer may have a limited number of signals available. Or, the desired signal must be made available on an output, which requires additional time. Simulation is valuable and as a guideline, at least 2X the number of hours spent writing the code should be spent developing and testing the code.

DESIGN SYNTHESIS

While some designers prefer to proceed directly to simulation, I prefer to synthesize the design. Synthesis is the process that reduces and optimizes the HDL or graphical design logic. Some third-party synthesis tools are available as a part of the FPGA vendor's complete development package. Synplicity's Synplify and Mentor Graphics' LeonardoSpectrum, Precision RTL, and Precision Physical are examples of third-party synthesis tools. Xilinx offers ISE Project Foundation, which is a complete development application that includes a synthesis tool. Altera has Quartus II Integrated Synthesis, QIS.

Each simulation level offers various benefits. RTL uncovers logic errors, the functional level verifies that the pre- and post-synthesis design are equivalent, and the gate level uncovers timing errors.

Although some FPGA vendors offer synthesis, they still recommend using a third-party's synthesis tools. The synthesis tool must be set up prior to actually synthesizing the design. Synplicity's Synplify goes through a common setup process, as it involves providing the design files (completed during design stage) and information about the FPGA. FPGA information includes the vendor's name, the specific part or family, the package type, and the speed. The synthesis process takes this information and the user-defined constraints and produces the output netlist. A constraints file specifies information like the critical signal paths and clock speeds. After completing set-up, synthesis can begin. General synthesis flow for tools like Synplicity's Synplify involves three steps, creating structural element, optimizing, and mapping. Figure 3 shows a synthesis flow diagram.

The first step in the synthesis process is to take the HDL design and compile it into structural elements. This means that the HDL design is technology independent. Synplify graphically represents this step as the "RTL Schematic View", viewable in Synplify. The next step involves optimizing the design, making it smaller and faster by removing unnecessary logic and allowing signals to arrive at the inputs or output faster. The goal of the optimizing process is the make the design perform better without changing the circuit's functions.

The final step in the synthesis process involves mapping or associating the design to the vendor specific architecture. The mapping process takes the design and maps or connects it using the architecture of the specific vendor. This means that the design connects to vendor-specific components such as look-up tables and registers. The optimized netlist is the output of the synthesis process. This netlist may be produced in one of several formats. Edif is a general netlist format accepted by most implementation tools, while .xnf format is specific to Xilinx and is only recognized by Xilinx's implementation.

In addition to the optimized netlist, many synthesis tools like Syn-

The design serves as the input to the synthesis process, resulting in a netlist that's used as the input to the place and route or implementation tool.

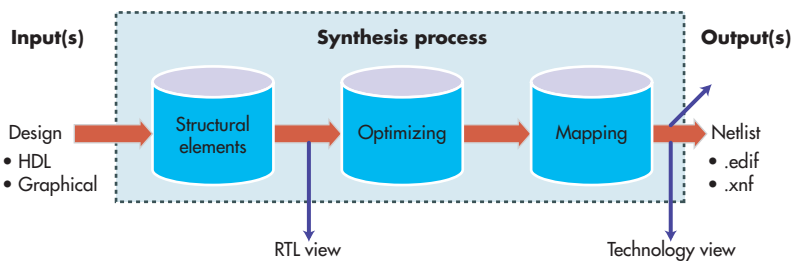


Figure 3

SuperH Flash Microcontroller Reaches Speeds up to 160MHz

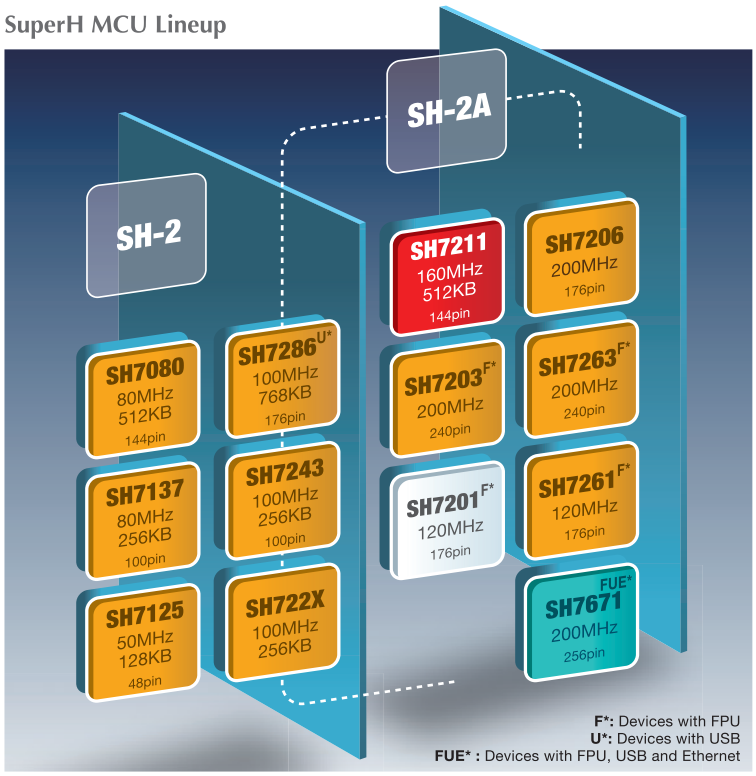
Superscalar performance,
high-speed on-chip flash memory access, and much more

Renesas Technology

No.1* supplier of microcontrollers in the world

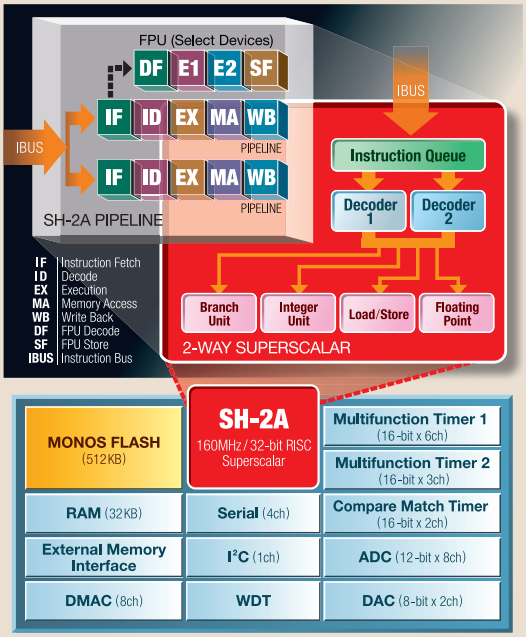
proudly presents the SuperH family of devices. SuperH devices equipped with the SH-2A core offer superscalar performance at speeds of 160MHz, allowing high-speed access to on-chip FLASH memory and up to 200MHz CPU performance. Enhanced features that include on board Floating Point Unit (FPU), Multiply Accumulate Unit (MAC), High-Speed Barrel Shifter and advanced addressing modes deliver DSP-like performance in RISC style architecture without the complicated programming associated with a DSP engine. The SuperH RISC engine and the SH-2A core are establishing new performance standards in the industry, and are ideal for systems that demand real-time, high-precision control and require a combination of high performance CPU with high-speed flash.

SuperH MCU Lineup



HOT Products **SH7211F**

(R5F72115D160FPV)



MONOS FLASH (512KB)	SH-2A 160MHz / 32-bit RISC Superscalar	Multifunction Timer 1 (16-bit x 6ch)
		Multifunction Timer 2 (16-bit x 3ch)
RAM (32KB)	Serial (4ch)	Compare Match Timer (16-bit x 2ch)
External Memory Interface	I²C (1ch)	ADC (12-bit x 8ch)
DMAC (8ch)	WDT	DAC (8-bit x 2ch)

Top Reasons to Select SuperH

480 Dhrystone MIPS at **200** MHz

- **Superscalar Performance**
  - Two instructions are executed per cycle at 160MHz
  - World's fastest embedded FLASH with 12.5ns read access time.
- **Fast Real-Time Control**
  - Register Bank architecture (15 Banks) for context switching enables 37.5nsec (6 cycles) interrupt latency time.
- **High Integration**
  - 512KB On-Chip Flash / 32KB On-Chip RAM
  - Advanced 16-bit PWM timers to drive two motors simultaneously
  - 1.25μsec 12-bit A/D conversion with 3 sample & hold circuits

*Source: Gartner (March 2007) "2006 Worldwide Microcontroller Vendor Revenue" GJ07168

#1 MCU REACH FURTHER

Get Started Today -

Register to qualify for a SuperH MCU Kit

www.america.renesas.com/ReachSH/c

Starter Kit for SH7211F:
R0K572115S000BE
USB Emulator:
HS0005KCU11H

Renesas Technology Corp.

Everywhere you imagine. RENESAS

plify will produce a netlist for gate-level simulation and other report files. Stimulus applied to this netlist instead of the original HDL design produces the functional-level simulation, which lets the designer verify that the synthesis process hasn't changed the design's functions. At this point, synthesis is complete and ready for the implementation process. Each FPGA vendor has its own implementation tool, such as Xilinx's Project Navigator and Altera's Quartus II's.

DESIGN IMPLEMENTATION

The final stage in the FPGA development process is the design implementation, also known as *place and route (PAR)*. If the FPGA vendor has a complete development tool, meaning it can perform synthesis, and the design is synthesized using this tool, little or no set-up is required for PAR. However, if a third-party synthesis tool is used, the implementation tool must be set up, which involves directing the PAR tool to the synthesized netlist and possibly a constraint file. The constraint file contains information such as maximum or minimum timing delays for selected signal(s) and I/O pin assignments.

Pin assignments can be automatic (performed by the tool) or manual (dictated by the designer). Automatic pin assignment is generally the best option for new designs, as it lets the tool more effectively route the design without having fixed pin assignments. It may be necessary to manually assign signals to specific pins to achieve easy board routing, to provide the minimum signal route for timing-critical signals, or be compatible with legacy designs.

There are numerous reasons why manual pin assignments would be necessary. But regardless of the reason, the designer must make this information available to the PAR tool, which is done by creating a user constraint file that's used by the PAR tool. After completing setup, the PAR process can begin. Each PAR tool may have a slightly different approach to design implementation, so consult your PAR documentation. Xilinx's Foundation or Project Navigator performs design implementation in three steps, translate, fit, and generate programming file.

Step one, called translate, involves verifying that the synthesized netlist is consistent with the selected FPGA architecture and there are no inconsistencies in the constraint file. Inconsistencies would consist of assigning two different signals to the same pin, assigning a pin to a power or ground pin, or trying to assign a non-existing design signal to a pin. If the design fails either check, the translate step will fail and the implementation process will be stopped.

The final step is to generate the programming file, which can be stored in flash memory, PROMs, or directly programming into the FPGA.

Translate errors must be corrected and the translation step must be error free before advancing to step two, which is the fit stage. This step involves taking the constraints file and netlist and distributing the design logic in the selected FPGA. If the design is too large or requires more resources or available logic than the selected device offers, the fitter will fail and halt the implementation process. To correct this type of error, replace the current FPGA with a larger one and re-synthesize, and repeat PAR for the design. A successful fit stage is necessary to proceed to generate the programming file stage.

All timing information is available and many PAR tools will provide the required files necessary for the simulator to perform a timing simulation. The final step is to generate the programming file, which can be stored in flash memory, PROMs, or directly programming into the FPGA. JTAG and third-party programmers like Data I/O are two programming methods used to store the programming file in memory. The appropriate format depends on the FPGA vendor, the programming method, and the device used to hold the programming.

There are various output formats; consult your documentation for the correct one. In addition to the implementation process creating the programming file, there are several output report files created, such as a pad file. The pad file contains information such as signal pin assignment, part number, and part speed.

BEYOND THE BASICS

This article gives some basic examples of the FPGA development process, so a new embedded systems designer, manager, technical lead from other disciplines, or someone wanting to diversify his or her skills can understand what it takes to develop and implement a digital design in a FPGA. The generic process provided here will vary depending on the FPGA tools since each vendor may perform some of these tasks in a slight different manner.

A good resource for furthering your knowledge is *Essential VHDL RTL Synthesis Done Right* (Sundar Rajan, F.E. Compton Co, 1998). ■

Gina R. Smith is CEO and owner of Brown-Smith Research and Development Laboratory Inc., an engineering services, technical training and consulting company. She is also a senior systems engineer, with responsibility for performing failure mode effect and criticality analysis, requirements analysis and definition, creating physical and functional block diagrams, and evaluating design tool needs. She has a BS in electrical engineering magna cum laude from North Carolina A&T State University and an MS with honors in systems engineering from Johns Hopkins University. Smith can be reached at Gina_R_Smith@BrownSmithRDL.com.

Learn how to combine the AVR® microcontrollers high performance with the lowest possible power consumption on www.atmel.com/avrman

Multicore architectures can provide the performance boost you're looking for, but the software is certainly more complicated.

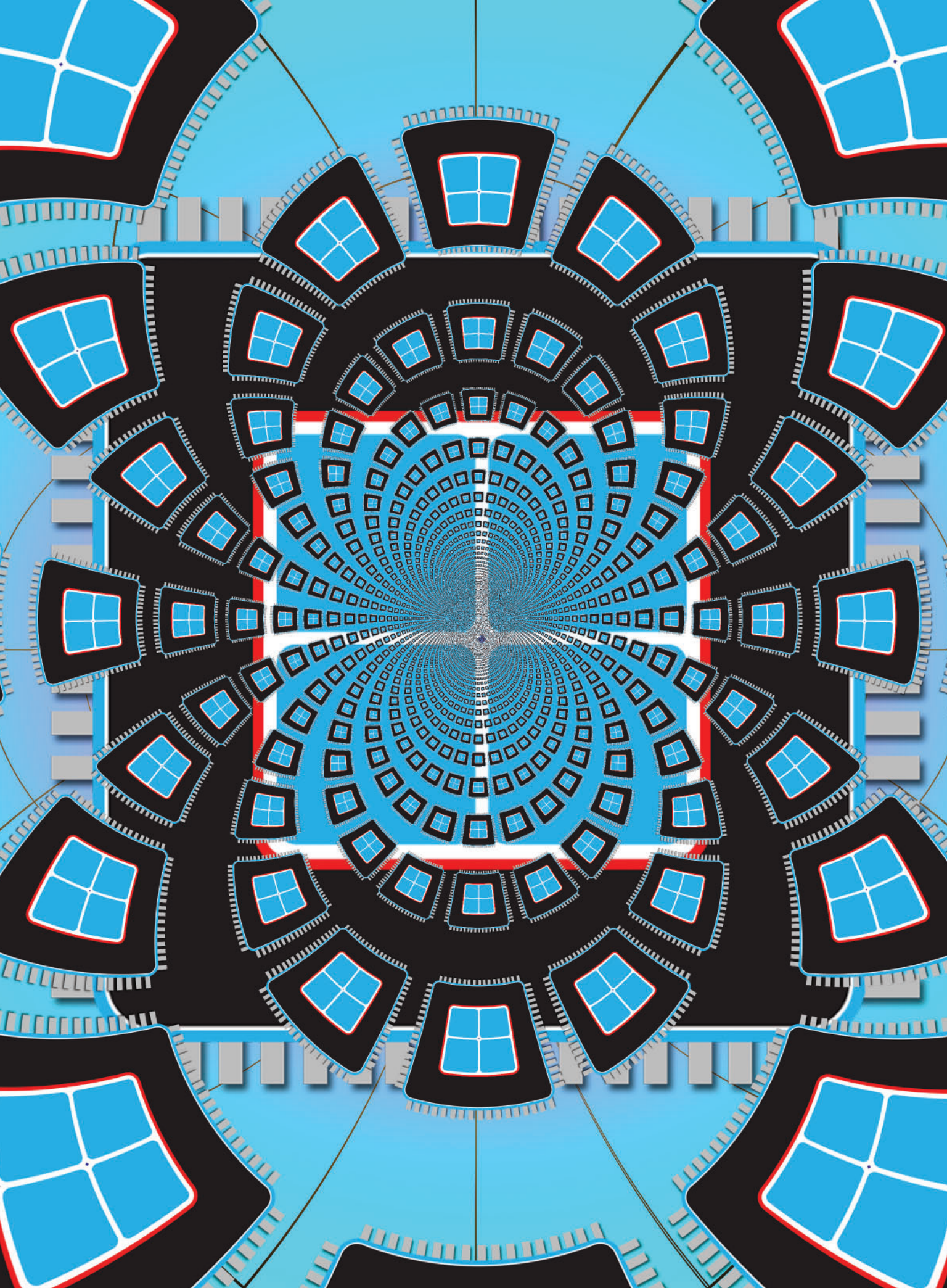
Is symmetric multiprocessing for you?

BY DAVID N. KLEIDERMACHER

For the past thirty years, computing has enjoyed continual boosts in performance, primarily due to increases in clock speed, pipelining efficiency, and cache size. Recently, however, traditional microprocessor optimization has hit the proverbial wall. Although tweaks such as further cache size increases can continue to nudge system performance, it's clear that Moore's gains are behind us. Meanwhile, embedded systems continue to grow in software complexity, with consumers expecting that all the bells and whistles will continue to come in ever shrinking cost, size, weight, and power footprints.

Microprocessor designers have concluded that the best path toward meeting the growing demand for performance with controlled footprint is to employ multicore architectures, in which the main premise is to partition the software and parallelize or offload execution across multiple processing elements. *Symmetric multiprocessing (SMP)* is one such architecture, consisting of homogenous cores that are tightly coupled with a common memory subsystem, as shown in Figure 1. SMP is a *de facto* standard on the desktop, but adoption in embedded applications has been slow, with recent surveys showing only a small percentage of designs using single-chip SMP-capable devices.

So if your design is in need of some extra horsepower, how can you determine whether SMP is a sensible choice? Several key requirements enable you to realize the promise of SMP. First, the software must be partitioned and parallelized to take advantage of the hardware concurrency. Second, operating systems must provide the load-balancing services required to enable distribution of software



An example of a symmetric multicore system is shown.

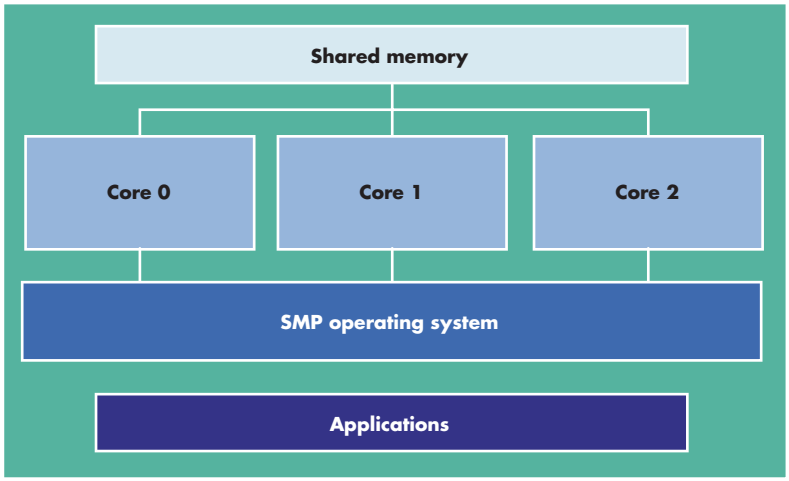


Figure 1

onto the multiple processing elements. And finally, you will need to learn and use development tools specifically tailored to the difficult task of multicore system debugging so you can find concurrency problems quickly and avoid time-to-market delays.

PROGRAMMING FOR CONCURRENCY

If your software has no potential for application-level parallelism (for example, a simple control system), then SMP is not for you. If software has the potential for parallelism but isn't currently multithreaded, then SMP could still be a good fit.

There are two ways to partition and parallelize software to take advantage of multicore concurrency: manual and automatic parallelization. *Manual parallelization* requires the programmer to deduce which parts of the application can be parallelized and write the code such that this parallelism is explicit. For example, the developer can place code into threads that will then be scheduled by an SMP operating system to run concurrently.

Automatic parallelization involves using a tool to discover a program's "parallelizability" and convert the code into an explicitly parallelized program. Some forms of parallelization focus specifically on loops. This approach is sensible: loops tend to be execution bottlenecks and sometimes can be converted into parallelizable iterations. However, many loops aren't parallelizable (even with a very smart compiler), and many applications simply don't benefit from this approach.

If software has the potential for parallelism but isn't currently multithreaded, then SMP could still be a good fit.

Parallelizing compilers do exist, but the embedded software community hasn't found automatic parallelization (autoparallelization, for short) technology to be of general use due to the compilers' focus on data-level parallelism. Certainly, a developer wouldn't take a legacy embedded control application running on a unicore platform and expect a parallelizing compiler to convert the application into something that runs optimally on an SMP. Autoparallelization may indeed boost performance in places, especially when the user can add some hints and directions to aid the compiler (known as semi-automatic parallelization), but a systemwide approach is required in general. Future innovations in autoparallelization could be more effective.

POSIX

POSIX is a collection of open standard APIs specified by the IEEE for operating system services. *POSIX threads*, or *Pthreads*, is the part of the standard that deals with multithreading. The Pthread APIs provide interfaces for run control of threads, synchronization primitives, and interprocess communication mechanisms. While other multithreading standards exist, Pthreads is the most generic, widely applicable standard. Pthreads are supported by a wide range of embedded operating systems such as Integrity, LynxOS, and QNX.

Due to POSIX's ubiquity, a large base of application code exists that can be reused for embedded SMP designs. Another strong advantage of POSIX is its independent conformance validation. The list of POSIX implementations that have been certified conformant to the latest POSIX specification can be found at *http://get.posixcertified.ieee.org/cert_prodlist.tpl?CALLER=index.tpl*. By programming to the POSIX API, developers can write multithreaded applications that can be ported to any multicore platform running a POSIX conformant operating system.

In embedded systems, add-on software components can often be easily mapped to individual threads. For example, a TCP/IP network stack can execute within the context of one POSIX thread; same for a file system server, audio application, and so forth. Because of this, many embedded software systems can take advantage of SMP to improve performance without significant application modifications.

LANGUAGE-LEVEL CONCURRENCY

Because threads are an integral part of the Java and Ada languages, designing multithreaded software in these languages is relatively natural. Java and Ada programs using language-level

threading can map nicely to SMP. Yet C and C++ remain the most popular languages for embedded systems. Surveys in recent years have shown C and C++ (which lack native thread support) accounting for about 80% of embedded software, with no significant downward trend.

If your software base is hopelessly dependent on a real-time operating system (RTOS) that doesn't support SMP, then SMP may not be for you. If you have the freedom to select a new operating system, your best bet at future portability is to select one that supports both POSIX and SMP. An SMP operating system will simply schedule concurrent threads to run on the extra cores in the system. This automatic load balancing is the primary advantage of SMP: adding cores will increase performance, often dramatically, without requiring software modifications.

There's one important exception to the automatic reusability of multithreaded applications on an SMP system. Most SMP operating systems will allow threads at varying priority levels to execute concurrently on the multiple cores. Most real-time embedded software is written for a strictly priority-based preemptive scheduler. Trouble will ensue if the software is using priority as a means of synchronization. For example, software may manually raise a thread's priority to preempt another thread. On an SMP system, this preemption won't occur if the two threads are the highest priority runnable threads on a dual-core system. Embedded designers must analyze their systems to ensure that the SMP scheduling algorithms won't pose a problem.

CORE BINDING

If your embedded system has tight real-time deadlines, than SMP may pose a problem: context switches can be delayed due to the overhead of interprocessor interrupts (IPIs) and cache inefficiency. For example, when an interrupt service routine executes on one core and signals a thread to run, the SMP scheduler may decide to run the thread on a different core, requiring an IPI. If the thread didn't last run on that same core, there will be additional overhead to rewarm the cache with the thread's code and data. SMP operating systems tend to migrate threads, making it difficult to predict whether this overhead will be incurred.

The good news is that most SMP operating systems provide the ability to map interrupts and bind threads to

The high-speed interconnect is the centerpiece of the NUMA system.

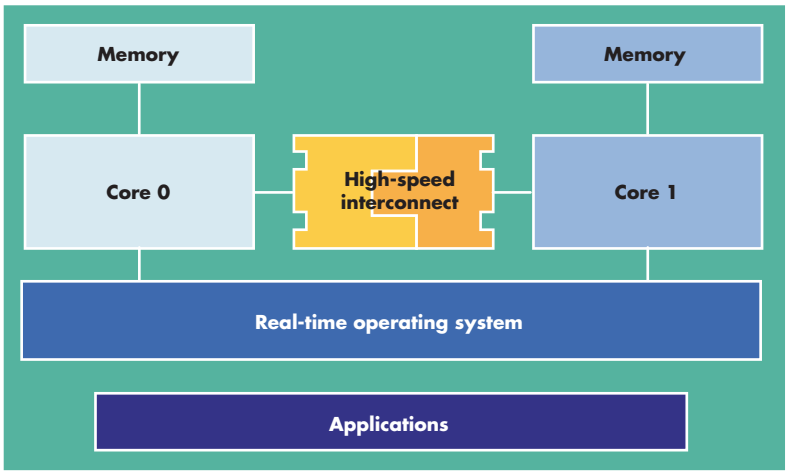


Figure 2

specific cores to specific cores. Thus, real-time performance can be accommodated while other software is optimized across the multiple cores as deemed appropriate by the RTOS. The bottom line: real-time systems can take advantage of SMP, but designers should be prepared to spend time tweaking the system's scheduling parameters.

NUMA FOR EMBEDDED

SMP's single memory-bus architecture may be a poor fit for memory- and I/O-bound applications, relative to compute-intensive systems. The only way to be sure of the payoff is to run the software on an SMP. However, engineers sitting on the SMP fence may be excited about the prospect of NUMA (non-uniform memory access) systems. NUMA is similar to SMP except that the system contains more than one memory source, where the time to access each memory source varies. This architecture is depicted in Figure 2.

NUMA represents a compromise in which code can still be shared and automatically load-balanced in the manner of an SMP. Yet you can optimize memory access times by running threads on the core for which the thread's memory references are local. One way to do this is simply to take advantage of the aforementioned binding capabilities of the SMP operating system. You can locate thread-required memory to a core's local memory bank and bind the thread to the same core. The NUMA-aware operating system may automate this optimization of memory and thread binding. Although NUMA isn't available in mainstream embedded devices, there are rumors about future parts that could provide an intriguing alternative to SMP in the future.

When moving to an SMP platform for the first time, developers must be prepared to use tools required in the multicore development, debugging, and optimization process. Tightly coupled multicore processors often provide a single on-chip debug port (such as JTAG) that enables a host debugger, connected with a hardware probe device, to debug multiple cores simultaneously. With this capability, developers can perform low-level, synchronized run control of the multiple cores. Board bring-up and device-driver development are two common uses of this type of solution.

The development tool lets developers visualize all the system's cores and choose any combination to debug, each optionally in its own window. At the same time, the tool provides controls for synchronized running and halting of the debugged cores.

SMP's single memory-bus architecture may be a poor fit for memory- and I/O-bound applications, relative to compute-intensive systems. The only way to be sure of the payoff is to run the software on an SMP.

RUN-MODE MULTICORE DEBUGGING

Run-mode debugging is also useful for SMP systems, as the cores are never stopped. Rather, the debugger controls application threads using a communications channel (usually Ethernet) between the host PC and a target-resident debug agent.

The SMP operating system typically provides an integrated debug agent (and the associated communications device drivers) that's operating-system–aware and provides flexible options for interrogating the system. For example, one operating system comes with a powerful debug agent that communicates with the debugger, providing the ability to debug any combination of user threads on any core. The user can set specialized breakpoints that enable user-defined groups of threads to be halted when another thread hits the breakpoint. Some classes of bugs require this fine-grained level of control.

By collecting a system's execution history and making it available for playback within debugging tools, even the most difficult multicore bugs become easy to find and fix. If you're new to SMP, choosing a processor with on-chip trace capabilities may be desirable.

Multicore trace capability is just starting to arrive on multicore processors. A major technical challenge that has kept this hardware feature from becoming a reality involves finding a way to keep up with trace data emitted simultaneously from multiple cores. An emerging solution is high-speed serial trace (HSST). HSST replaces the current generation of parallel trace ports by taking advantage of high-speed serial bus technology, which enables higher data throughput with a lower pin count. HSST has been proposed to the Nexus standards committee. In addition, ARM has adopted HSST as part of its CoreSight trace solution.

SMP is a promising technology for improved performance in an attractive cost and power footprint. However, SMP is not a panacea. The application must have the potential for concurrency, and designers may need to manually refactor software to unlock this concurrency. Furthermore, SMP systems are more difficult to manage and debug than unicore designs. This in turn may require switching operating systems and tooling to acquire the load balancing and multicore debugging capabilities that go hand in hand with SMP. ■

David Kleidermacher is chief technology officer at Green Hills Software where he has been designing compilers, software development environments, and real-time operating systems for the past 16 years. David frequently publishes articles in trade journals and presents papers at conferences on topics relating to embedded systems. He holds a BS in computer science from Cornell University, and can be reached at davek@ghs.com.

Embedded Systems CONFERENCE SILICON VALLEY

LEARN TODAY.
DESIGN TOMORROW.

Conference: **April 14 – 18, 2008**
Expo: **April 15 – 17, 2008**
McEnery Convention Center, San Jose, CA

Embedded Systems Conference Silicon Valley delivers a comprehensive technical program focusing 15+ critical topics that affect your designs. Learn how to solve your engineering issues today.

register today at **www.embedded.com/esc/sv**