



PROGRAMMING
THE **CELL**
PROCESSOR

For Games, Graphics,
and Computation

MATTHEW SCARPINO

Foreword by DR. DUC VIANNEY, *Technical Solution Architect, IBM*

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: www.informit.com/ph

Library of Congress Cataloging-in-Publication Data:

Scarpino, Matthew, 1975-

Programming the Cell processor : for games, graphics, and computation / Matthew Scarpino. — 1st ed.

p. cm.

ISBN 0-13-600886-0 (hardback : alk. paper) 1. Microprocessors—Programming. 2. Computer architecture. 3. Computer games—Programming. I. Title.

QA76.6.S376 2008

794.8'1526—dc22

2008031247

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-13-600886-6

ISBN-10: 0-13-600886-0

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing, October 2008

Editor-in-Chief

Mark Taub

Acquisitions Editor

Bernard Goodwin

Managing Editor

Kristy Hart

Project Editor

Betsy Harris

Copy Editor

Keith Cline

Indexer

Erika Millen

Proofreader

Editorial
Advantage

**Technical
Reviewers**

Duc Vianney

Sean Curry

Hema Reddy

Michael Kistler

Michael Brutman

Brian Watt

Yu Dong Yang

Daniel Brokenshire

Nicholas Blachford

Alex Chungen Chow

Publishing

Coordinator

Michelle Housley

Interior Designer

Gary Adair

Cover Designer

Chuti Prasertsith

Senior Composer

Gloria Schurick

Foreword

The Cell Broadband Engine Architecture® (Cell/B.E.) defines a new processor structure that extends the 64-bit Power Architecture® technology and incorporates unique features that support critical real-time response of highly parallel, computationally intensive code. The first implementation of the architecture has resulted in a single device of heterogeneous processors mixed with simultaneous multithreading and special computational “accelerator” cores for more performance and efficiency gains. The processor comprises a vector Power Processor Element (PPE) with two levels of cache and eight independent Synergistic Processor Elements (SPEs), each with its own multilevel storage organization. In addition to processor-level parallelism, each processing element has Single Instruction Multiple Data (SIMD) units that can process from 4 words up to 16 characters per processing cycle, and globally consistent Direct Memory Access (DMA) engines that provide a rich set of DMA commands for seamless communication between all Cell/B.E. processing elements. The dual thread, 64-bit PPE controls and runs the operating system, manages system resources, and coordinates work sent to the SPEs, while the SPEs are dedicated only to the compute thread and its data.

Although the Cell/B.E. processor was initially intended for application in game consoles and media-rich consumer-electronics devices, a much broader use of the architecture is starting to emerge to handle streaming workloads found in today’s applications across such industry segments as health care, life sciences, petroleum exploration, financial services, digital media, electronics, government, and education. To fully harness the power of Cell/B.E., developers need to use different programming models and to write multithreaded code that ensures the SPEs are well utilized and work is equally off-loaded across the SPEs.

This book is the first commercially published book written for those who want to program the Cell/B.E. It provides comprehensive learning material for developing multithreaded code that allows an application structured for parallelism to execute much more efficiently on those SPE accelerators. It starts with the IBM Software Development Kit (SDK) for Multicore Acceleration Version 3.0, which consists of run-time tools (such as the Linux kernel), development tools, GNU toolchain, software libraries and frameworks, performance tools, a system simulator, sample source codes and benchmarks—all of which fully support the capabilities of the Cell/B.E. The book discusses the device’s architecture, including the PPE vector processing unit, SPE SIMD capabilities, and DMA communication mechanisms. The book also details SPU programming and exploitation of SPU storage by software module overlay and software managed cache. It includes an in-depth section on libraries, ranging from the basic vector and matrix libraries to BLAS, MASS, FFT, and Monte Carlo libraries.

Multicore processing is the future trend of the computing industry. Cell/B.E. offers a general-purpose, programmable multicore processor available to developers through high level programming languages such C/C++ and Fortran. However, in order to fully leverage the processor parallel capabilities, the application needs to be properly designed in both code and data. This book describes the processor in enough details and contains

many practical code examples to get you off the ground to write your first Cell/B.E. application. Such experience should give you a competitive edge when working with other multicore systems, or as a foundation to work on Cell/B.E. homogeneous and/or hybrid clusters that are being deployed to support a wide range of high performance computing workloads.

Duc J. Vianney, Ph. D.

Technical Solution Architect¹

Education Lead—Cell/QUASAR Ecosystem & Solutions Enablement

IBM Systems & Technology Group, Industry Systems Division

¹ What is the IBM definition of “Technical Solution Architect”?

“This role assumes overall technical responsibility for the success of solution construction, implementation, and system integration in a technology, industry, or business specialty. They deliver new and complex high quality solutions to clients in response to varying business requirements. They utilize in-depth knowledge of leading-edge products and technology in conjunction with industry and business skills to influence clients.”

Preface

In August 2006, I attended the Girvan Workshop for the Cell Broadband Engine, and it's an experience I'll never forget. For two solid days, IBM engineers explained the processor's architecture, tools, and the many software libraries available for building Cell applications. I was stunned, not only by the processor's extraordinary capabilities, but also by how much there was to learn: spulets and CESOF files, AltiVec and Single Instruction, Multiple Data (SIMD) math, debuggers and simulators. I did my best to comprehend it all, but most of the material flew over my head with a fierce *whoosh*.

When Sony released the PlayStation 3, I grabbed the first console off the shelf and ran home for a more thorough investigation. It was daunting at first. Then as now, IBM's Software Development Kit provided a vast number of documents that covered three essential subjects: development tools, software libraries, and the processor itself. The docs were helpful, but there was no overlap or coordination between them. This is a serious problem because any practical Cell developer needs to understand these subjects as an integrated whole.

It took time before the whooshing sound dissipated, but when it did, I genuinely understood how to program the Cell's PowerPC Processor Unit (PPU) and Synergistic Processor Units (SPUs). It wasn't that hard, really—just regular C/C++ and a set of communication mechanisms. Yet the blogs and discussion sites disagreed: To them, Cell programming was much too complex for normal developers to understand. However, they hadn't really given the Cell a chance; they saw the disjointed pieces, but not how they fit together.

This book is my best attempt to reduce the *whoosh* associated with Cell development. My goal is to tie together the Cell's tools, architecture, and libraries in a straightforward progression that appeals to the intuition. And I've included many code examples throughout each of the main chapters so that you can follow the material in a hands-on fashion. To download these examples, go to www.informit.com/title/9780136008866.

You can't accomplish anything without coding tools, so the first part of this book deals the Software Development Kit (SDK). Chapter 2 explains how to obtain the SDK, Chapter 3 describes the GCC-based buildchain, and Chapter 4 shows how the `ppu-gdb/spu-gdb` debuggers work as well as IBM's Full System Simulator. Chapter 5 covers much of this ground again in presenting the Cell SDK integrated development environment.

After the tools have been covered, Part II describes the Cell's central processing core: the PPU. Chapter 6 discusses the overall PPU structure and the basic programming libraries available. Chapter 7 covers the PPU's most important capability: controlling the SPUs in code with the SPE Runtime library. Chapter 8 introduces the AltiVec instructions that it uses to process vectors, and Chapter 9 builds on Chapter 8, covering advanced topics of PPU vector processing.

Part III is the largest and most fascinating of the book's sections, and describes the SPU in depth. Chapter 10 provides an overall introduction, and Chapter 11 explains the instructions that the SPU uses to process vectors. These instructions are similar to

Altivec's, but provide many new capabilities. Chapters 12 and 13 describe the communication mechanisms available for PPU-SPU data transfer, including direct memory access (DMA), mailboxes, and signals. Chapter 14 delves into advanced SPU topics, such as overlays, software caching, and SPU isolation. The last chapter in Part III, Chapter 15, explains how to program the SPU in assembly.

Part IV is the first of the parts that focuses on the SDK's libraries. Chapter 16 covers the vector and matrix libraries, including the Large Matrix library and the BLAS library. Chapter 17 covers the two fast Fourier transform (FFT) libraries made available in the SDK and Chapter 18 describes the Multiprecision library and the Monte Carlo API.

Part V builds on Part IV and shows how the computation libraries provide for graphic processing. Chapter 19 deals with basic 2D graphic development using the Linux frame buffer. Chapter 20 improves upon this, explaining how to build 3D graphics with the new Gallium OpenGL library. Chapter 21 builds upon Chapter 20 by showing how Ogre3D uses OpenGL to build games, and Chapter 22 shows how these games can be accessed and packaged using the COLLADA standard for digital content interchange.

This book is by no means an exhaustive reference on the Cell, but it should give you enough information to make you familiar and comfortable with many of its aspects. From there, you'll be in a perfect position to conduct further investigations on your own.

A note on typography:

When a line of code is too long to fit on one line of text, it is wrapped to the next line. In this case, the continuation is preceded with a code-continuation arrow `↳`. For example

```
unsigned int numElements = dae->getDatabase()->getElementCount(NULL,  
↳"animation", NULL);
```

Introducing the Cell Processor

In September 2007, the *Guinness Book of World Records* announced the new record holder for the world's most powerful distributed computing system. It wasn't a traditional cluster of high-performance computers, but a worldwide network composed of regular PCs and PlayStation 3 consoles (PS3s). Called Folding@Home, this distributed computing system simulates protein folding to analyze how diseases originate.

Before the PS3s joined in, the network was capable of only .25 petaflops (250,000,000,000,000 floating-point operations per second). But when the Folding@Home client started running on PS3s, the computation speed *quadrupled* in six months, making Folding@Home the first distributed computing system to break the 1 petaflop barrier.

Table 1.1 clarifies the significance of the PS3s in the Folding@Home network. There aren't nearly as many consoles as PCs, but they provide more computational power than the Windows/Mac/Linux computers combined.

The PS3's tremendous computational power is provided by the Cell Broadband Engine, commonly called the Cell processor or just the Cell. Developed by the STI Alliance (Sony, Toshiba, and IBM), the Cell combines the general-purpose capability of IBM's PowerPC architecture with enough number crunching to satisfy even the most demanding gamers and graphic developers.

Table 1.1 **Folding@Home Performance Statistics (Recorded on April 16, 2008)**

OS Type	Current Pflops	Active CPUs	Total CPUs
Windows	.182	190,892	1,986,517
Mac OS X/PowerPC	.007	8,478	114,326
Mac OS X/Intel	.023	7,428	45,480
Linux	.047	27,796	286,172
PlayStation 3	1.235	40,880	492,491

What does this mean for you? It means you can enjoy the best of both worlds: computational flexibility and power. On one hand, you can install a common operating system, such as Linux, on the Cell and execute applications as conveniently as if they were running on a PC. On the other hand, you can implement computationally intense algorithms at speeds that far exceed regular CPUs and even compete with supercomputing devices.¹ More incredibly still, you can do both *at the same time*.

The Cell makes this possible through an on-chip division of labor: The operating system runs on a single PowerPC Processor Element (PPE), and the high-speed calculation is performed by a series of Synergistic Processor Elements (SPEs). These two types of cores are specifically designed for their tasks, and each supports a different set of instructions.

Taken individually, these processing elements are easy to understand and straightforward to program. The hard part is coordinating their operation to take the fullest advantage of their strengths. To accomplish this, a coder must know the Cell-specific programming commands and have a solid knowledge of the device's architecture: its processing elements, interconnections, and memory structure.

The purpose of this book is to cover these subjects in enough depth to enable you to create applications that maximize the Cell's capabilities. Much of this treatment delves into the processor's architecture, but only the aspects that you can use and configure in code. Some topics may seem overwhelming for those not used to thinking like computer architects, but don't be concerned: Everything will be explained as the need arises. And the goal of this book is always *software*.

The goal of this chapter is to explain, at a basic level, what the Cell processor is and how it works. The discussion begins with a description of the Cell's background, including its history and capabilities, and proceeds to introduce the processor's basic architecture.

1.1 Background of the Cell Processor

The Cell is so unlike its predecessors that it helps to know about why it was created and the corporate forces that shaped its design. When you see why the STI Alliance spent so much time and effort on the Cell, you'll have a better idea why learning about it is worth your own.

History of the Cell

Sony finished development of the PlayStation 2 in 1999 and released it the following year. Despite its tremendous success, then-CEO Nobuyuki Idei was nervous: How could

¹ See Table 1.2 for a comparison between the Cell, Cray's X1E, AMD's Opteron, and Intel's Itanium2 processors.

Sony's next-generation console top the PS2? What more could they accomplish? The answer, he decided, was twofold: The next offering had to integrate broadband multimedia capability and provide dramatic improvements in graphical processing. These lofty goals required entirely new hardware, and to make this possible, he conferred with IBM's then-CEO, Louis Gerstner. Together they shaped the concept that would ultimately lead to the Cell Processor.

The chief of Sony Computer Entertainment, Ken Kutaragi, fleshed out the hardware requirements and made demands that went far beyond the state of the art. Envisioning each processor as a building block of a larger, networked entity, Ken Kutaragi called the device the Cell. In keeping with Nobuyuki-san's original intention, the project became called the Cell Broadband Engine (CBE). This remains the official name of the Cell processor.

Toshiba expressed an interest in using the Cell in their consumer electronics, and in 2001, Sony, Toshiba, and IBM announced the formation of the STI Alliance. Their stated intention was to research, develop, and manufacture a groundbreaking processor architecture. They formed the STI Design Center in Austin, Texas, to turn the CBE's requirements into reality.

Cell Processor Timeline

1999: Sony CEO Nobuyuki Idei proposes a partnership with IBM to design the successor to the PlayStation 2 processor.

2001: Sony Computer Entertainment, Toshiba, and IBM (STI) announce an alliance to develop a processor for broadband multimedia applications. They form the STI Design Center in Austin.

2004: First processor manufactured at IBM's facility in East Fishkill, New York, and tested successfully at frequencies above 4GHz. IBM and Sony Computer Entertainment formally announce the Cell, describing its 64-bit PowerPC core and the multiple SPEs.

2005: Sony unveils the PlayStation 3 at the 2005 Electronic Entertainment Expo.

2006: Sony Computer Entertainment releases the PlayStation 3—the first commercial offering of the Cell Processor. IBM provides the first CBE Software Development Kit (SDK) and announces its new Roadrunner supercomputer design.

2008: IBM's Roadrunner supercomputer (12,960 Cell processors, 12,960 Optrons) becomes the world's fastest supercomputer, reaching a peak processing speed of 1.026 petaflops.

As the Cell's chief architect, Jim Kahle saw that the CBE's requirements couldn't be met with a traditional single-core processor—the demand for power would be too great. Instead, he chose a more power-efficient design that incorporated multiple processing units into a single chip. The final architecture consisted of nine cores: one central processing element and eight dedicated elements for high-speed computation.

At the time of this writing, the STI Design Center has grown to more than 400 engineers. Dr. H. Peter Hofstee, one of the Cell's founding designers, holds the positions of chief scientist and chief architect of the SPE. In a recent presentation, he listed the main goals that drove the Cell's design:

1. Outstanding performance on gaming and multimedia applications
2. Real-time responsiveness to the user and the network
3. Applicability to a wide range of platforms

In 2004, IBM's semiconductor manufacturing plant in East Fishkill produced the first Cell prototype. The STI engineers installed Linux and tested the processor at speeds beyond the commonly stated range of 3 to 4GHz. The prototype passed. Over the next year, Sony and IBM worked feverishly to integrate the device within Sony's next-generation console, and expectant gamers caught their first glimpse of the PlayStation 3 at the 2005 Electronic Entertainment Expo (E3).

November 2006 marked the full commercial release of the PS3, and the tales of long lines and barely sane consumers will amuse retail personnel for years to come. In addition to its powerful Cell processor brain, the new console provided resolution up to 1080p and a Blu-ray drive for high-definition video.

That same year, IBM released its first CBE Software Development Kit (SDK) to enable developers to build applications for the Cell. The SDK provides compilers for both types of processing elements, a combined simulator/debugger, numerous code libraries, and an Eclipse-based development environment. A great deal of this book is concerned with the SDK and how you can use it to build applications.

In mid-2008, the first Cell-based supercomputer, called the IBM Roadrunner, was tested in the Los Alamos National Laboratory. Containing 12,960 Cell processors and 12,960 Opterons, the Roadrunner reached a processing speed of 1.026 petaflops and has become the fastest of the supercomputers on the TOP500 list. Its speed more than doubles that of the second-place supercomputer, BlueGene/L, at .478 petaflops.

Potential of the Cell Processor for Scientific Computing

In 2005, the Lawrence Berkeley National Laboratory studied the Cell's computational performance and recorded their findings in the report *The Potential of the Cell Processor for Scientific Computing*. They simulated a number of different algorithms and compared the Cell's processing speed to that of similar processors: the AMD Opteron, Intel's Itanium2, and Cray's X1E. Table 1.2 tabulates their results.

Table 1.2 **Results of the Lawrence Berkeley National Laboratory Study (All Values in Gflops/s)**

Algorithm	Cell Processor	Cray X1E	AMD Opteron	Intel Itanium2
Dense matrix multiply (single precision)	204.7	29.5	7.8	3.0
Dense matrix multiply (double precision)	14.6	16.9	4.0	5.4
Symmetric sparse matrix vector multiply (single precision) ¹	7.68	—	.80	.83
Symmetric sparse matrix vector multiply (double precision) ¹	4.00	2.64	.60	.67
Nonsymmetric Sparse Matrix Vector Multiply (Single Precision) ¹	4.08	—	.53	.41
Nonsymmetric sparse matrix vector multiply (double precision) ¹	2.34	1.14	.36	.36
2-D fast Fourier transform (single precision) ²	40.5	8.27	.34	.15
2-D fast Fourier transform (double precision) ²	6.7	7.10	.19	.11

¹Sparse matrix results averaged from samples in the SPARSITY suite of matrices.

²2-D FFTs performed with 2K x 2K array sizes.

There are two points to keep in mind. First, the results refer to the computation speed in billions of flops (floating-point operations per second), not the amount of time needed to perform the algorithm. Second, because the first-generation Cell's multipliers are single precision, the first-generation Cell performs much better with single-precision values than with double-precision values. But the second generation provides hardware multiplication of double-precision values.

To an engineer interested in signal processing and computational mathematics (like myself), the results are nothing short of astounding. The study justifies the outrageous marketing claims: The Cell really provides supercomputer-like capability for nearly the cost and power (approximately 50 to 60 W) of a regular CPU.

1.2 The Cell Architecture: An Overview

In Randall Hyde's fine series of books, *Write Great Code*, one of his fundamental lessons is that, for optimal performance, you need to know how your code runs on the target processor. Nowhere is this truer than when programming the Cell. It isn't enough to learn the C/C++ commands for the different cores; you need to understand how the elements communicate with memory and one another. This way, you'll have a bubble-free instruction pipeline, an increased probability of cache hits, and an orderly, nonintersecting communication flow between processing elements. What more could anyone ask?

Figure 1.1 shows the primary building blocks of the Cell: the Memory Interface Controller (MIC), the PowerPC Processor Element (PPE), the eight Synergistic Processor Elements (SPEs), the Element Interconnect Bus (EIB), and the Input/Output Interface (IOIF). Each of these is explored in greater depth throughout this book, but for now, it's a good idea to see how they function individually and interact as a whole.

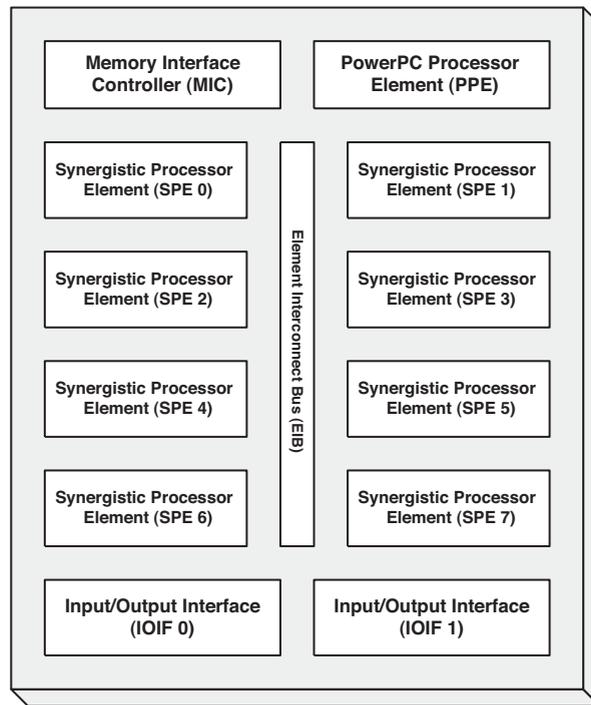


Figure 1.1 The top-level anatomy of the Cell processor

The Memory Interface Controller (MIC)

The MIC connects the Cell's system memory to the rest of the chip. It provides two channels to system memory, but because you can't control its operation through code, the discussion of the MIC is limited to this brief treatment. However, you should know that, like the PlayStation 2's Emotion Engine, the first-generation Cell supports connections only to Rambus memory.

This memory, called eXtreme Data Rate Dynamic Random Access Memory, or XDR DRAM, differs from conventional DRAM in that it makes eight data transfers per clock cycle rather than the usual two or four. This way, the memory can provide high data bandwidth without needing very high clock frequencies. The XDR interface can

support different memory sizes, and the PlayStation 3 uses 256MB of XDR DRAM as its system memory.

The PowerPC Processor Element (PPE)

The PPE is the Cell's control center. It runs the operating system, responds to interrupts, and contains and manages the 512KB L2 cache. It also distributes the processing workload among the SPEs and coordinates their operation. Comparing the Cell to an eight-horse coach, the PPE is the coachman, controlling the cart by feeding the horses and keeping them in line.

As shown in Figure 1.2, the PPE consists of two operational blocks. The first is the PowerPC Processor Unit, or PPU. This processor's instruction set is based on the 64-bit PowerPC 970 architecture, used most prominently as the CPU of Apple Computer's Power Mac G5. The PPU executes PPC 970 instructions in addition to other Cell-specific commands, and is the only general-purpose processing unit in the Cell. This is why Linux is installed to run on the PPU and not on the other processing units.

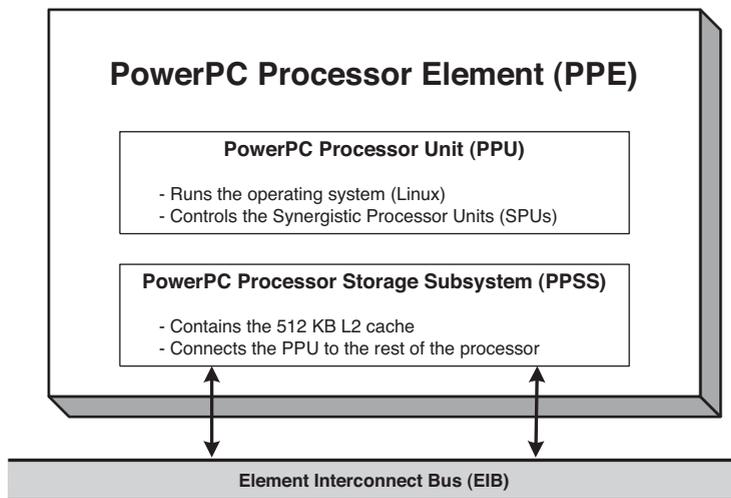


Figure 1.2 Structure of the PPE

But the PPU can do more than just housekeeping. It contains IBM's VMX engine for Single Instruction, Multiple Data (SIMD) processing. This means the PPU can operate on groups of numbers (e.g., multiply two sets of four floating-point values) with a single instruction. The PPU's SIMD instructions are the same as those used in Apple's image-processing applications, and are collectively referred to as the AltiVec instruction set. Chapter 8, "SIMD Programming on the PPU, Part 1: Vector Libraries and Functions," is dedicated to AltiVec programming on the PPU.

Another important aspect of the PPU is its capacity for symmetric multithreading (SMT). The PPU allows two threads of execution to run at the same time, and although each receives a copy of most of the PPU's registers, they have to share basic on-chip execution blocks. This doesn't provide the same performance gain as if the threads ran on different processors, but it allows you to maximize usage of the PPU resources. For example, if one thread is waiting on the PPU's memory management unit (MMU) to complete a memory write, the other can perform mathematical operations with the vector and scalar unit (VXU).

The second block in the PPE is the PowerPC Processor Storage Subsystem, or PPSS. This contains the L2 cache along with registers and queues for reading and writing data. The cache plays a very important role in the Cell's operation: not only does it perform the regular functions of an L2 cache, it's also the only shared memory bank in the device. Therefore, it's important to know how it works and maintains coherence. Chapter 6, "Introducing the PowerPC Processor Unit (PPU)," covers this topic in greater depth.

The Synergistic Processor Element (SPE)

The PPU is a powerful processor, but it's the Synergistic Processor Unit (SPU) in each SPE that makes the Cell such a groundbreaking device. These processors are designed for one purpose only: high-speed SIMD operations. Each SPU contains two parallel pipelines that execute instructions at 3.1GHz. In only a handful of cycles, one pipeline can multiply and accumulate 128-bit vectors while the other loads more vectors from memory.

SPUs weren't designed for general-purpose processing and aren't well suited to run operating systems. Instead, they receive instructions from the PPU, which also starts and stops their execution. The SPU's instructions, like its data, are stored in a unified 256KB local store (LS), shown in Figure 1.3. The LS is not cache; it's the SPU's own individual memory for instructions and data. This, along with the SPU's large register file (128 128-bit registers), is the only memory the SPU can directly access, so it's important to have a deep understanding of how the LS works and how to transfer its contents to other elements.

The Cell provides hardware security (or digital rights management, if you prefer) by allowing users to isolate individual SPUs from the rest of the device. While an SPU is isolated, other processing elements can't access its LS or registers, but it can continue running its program normally. The isolated processor will remain secure even if an intruder acquires root privileges on the PPU. The Cell's advanced security measures are discussed in Chapter 14, "Advanced SPU Topics: Overlays, Software Caching, and SPU Isolation."

Figure 1.3 shows the Memory Flow Controller (MFC) contained in each SPE. This manages communication to and from an SPU, and by doing so, frees the SPU for crunching numbers. More specifically, it provides a number of different mechanisms for interelement communication, such as mailboxes and channels. These topics are discussed in Chapters 12, "SPU Communication, Part 1: Direct Memory Access (DMA)," and 13, "SPU Communication, Part 2: Events, Signals, and Mailboxes."

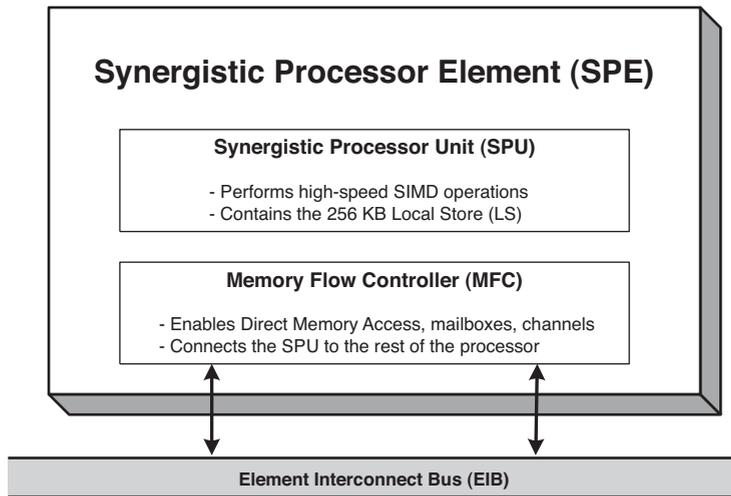


Figure 1.3 Structure of the SPE

The MFC's most important function is to enable direct memory access (DMA). When the PPU wants to transfer data to an SPU, it gives the MFC an address in system memory and an address in the LS, and tells the MFC to start moving bytes. Similarly, when an SPU needs to transfer data into its LS, it can not only initiate DMA transfers, but also create lists of transfers. This way, an SPU can access noncontiguous sections of memory efficiently, without burdening the central bus or significantly disturbing its processing.

The Element Interconnect Bus (EIB)

The EIB serves as the infrastructure underlying the DMA requests and interelement communication. Functionally, it consists of four rings, two that carry data in the clockwise direction (PPE > SPE1 > SPE3 > SPE5 > SPE7 > IOIF1 > IOIF0 > SPE6 > SPE4 > SPE2 > SPE0 > MIC) and two that transfer data in the counterclockwise direction. Each ring is 16 bytes wide and can support three data transfers simultaneously.

Each DMA transfer can hold payload sizes of 1, 2, 4, 8, and 16 bytes, and multiples of 16 bytes up to a maximum of 16KB. Each DMA transfer, no matter how large or small, consists of eight bus transfers (128 bytes). As Chapter 12 explains, DMA becomes more efficient as the data transfers increase in size.

The Input/Output Interface (IOIF)

As the name implies, IOIF connects the Cell to external peripherals. Like the memory interface, it is based on Rambus technology: FlexIO. The FlexIO connections can be configured for data rates between 400MHz to 8GHz, and with the high number of connections on the Cell, its maximum I/O bandwidth approaches 76.8GB/s. In the

PlayStation 3, the I/O is connected to Nvidia's RSX graphic processor. The IOIF can be accessed only by privileged applications, and for this reason, interfacing the IOIF lies beyond the scope of this book.

1.3 The Cell Broadband Engine Software Development Kit (SDK)

This book uses a hands-on approach to teach Cell programming, so the development tools are very important. The most popular toolset is IBM's Software Development Kit (SDK), which runs exclusively on Linux and provides many different tools and libraries for building Cell applications.

IBM provides the SDK free of charge, although some of the tools have more restrictive licensing than others. For the purposes of this book, the most important aspect of the SDK is the GCC-based toolchain for compiling and linking code. The two compilers, `ppu-gcc` and `spu-gcc`, compile code for the PPU and SPU, respectively. They provide multiple optimization levels and can combine scalar operations into more efficient vector operations. Chapter 3, "Building Applications for the Cell Processor," explores these and the rest of the SDK's build tools in depth.

The SDK also includes IBM's Full-System Simulator, tailored for Cell applications. This impressive application runs on a conventional computer and provides cycle-accurate simulation of the Cell processor, keeping track of every thread and register in the PPU and SPUs. In addition to basic simulation and debugging, it provides many advanced features for responding to processing events. Chapter 4, "Debugging and Simulating Applications," covers the simulator in thorough detail.

The SDK contains many code libraries to ease the transition from traditional programming to Cell development. It provides most standard C/C++ libraries for both the PPU and SPU, POSIX commands for the PPU, and a subset of the POSIX API on the SPU. Many of the libraries are related to math, but others can be used to profile an SPU's operation, maintain a software cache, and synchronize communication between processing units.

All of these tools and libraries can be accessed through the Cell SDK integrated development environment (IDE). This is an Eclipse-based graphical user interface for managing, editing, building, and analyzing code projects. It provides a powerful text editor for code entry, point-and-click compiling, and a feature-rich interface to the Cell debugger. With this interface, you can watch variables as you step through code and view every register and memory location in the Cell. Chapter 5, "The Cell SDK Integrated Development Environment," explains the Cell IDE in detail.

1.4 Conclusion

Some time ago, I had the pleasure of programming assembly language on a multicore digital signal processor, or DSP. The DSP performed matrix operations much, much faster than the computer on my desk, but there were two problems: I had to write all the routines for resource management and event handling, and there was no file system to organize the data. And without a network interface, it was hard to transfer data in and out of the device.

The Cell makes up for these shortcomings and provides many additional advantages. With SIMD processing, values can be grouped into vectors and processed in a single cycle. With Linux running on the PPE, memory and I/O can be accessed through a standard, reliable API. Most important, when all the SPEs crunch numbers simultaneously, they can process matrices at incredible speed.

The goal of this book is to enable you to build applications with similar performance. As with the DSP, however, it's not enough just to know the C/C++ functions. You have to understand how the different processing elements work, how they're connected, and how they access memory. But first, you need to know how to use the tools. The next chapter explains how to acquire and install IBM's SDK.

Symbols

\ (backslash), 41
(pound sign), 37
\$\$ variable, 50
\$\$+ variable, 50
\$\$? variable, 50
\$\$@ variable, 50
\$\$^ variable, 50
3D games, 529
4x4 matrices, 416

A

a instruction, 382
a-form (absolute addressing), 373
ABI (Application Binary Interface), 401
 calling C/C++ functions from
 assembly code, 405-406
 writing assembly-coded functions, 402
 example, 404-405
 function declaration, 402
 stack management, 402-403
absdb instruction, 382-384
absi4 function, 263
absolute addressing (a-form), 373
Accelerated Library Framework, 617
accelerator memory buffers, 619-620
accelerator stages
 accelerator environment
 functions, 637
 implementing with functions, 634-637
 kernel API export definition
 section, 637
 macros, 635
 overview, 632
accessing
 device files, 493
 ELF (Executable and Linking Format)
 files in code, 604-607
 frame buffer, 494

- kernel source code, 612-613
- SPU software cache
 - safe software cache functions, 352-353
 - unsafe software cache functions, 353-354
- unaligned memory, 195-197
 - adding unaligned arrays, 198-200
 - vec_lvsl function, 197
 - vec_perm function, 197
- acknowledging events, 327**
- acosd2 function, 270**
- acosf4 function, 191, 270**
- actions, trigger actions**
 - associating with trigger events, 71-73
 - definition, 70
 - implementing, 70-71
- add-on packages**
 - installing, 611
 - overview, 609
 - ps3-utils, 610-611
 - table of available packages, 610
- addChild function, 550**
- addition**
 - addition functions
 - PPU, 166-170
 - SPU, 246-250
 - matrix addition, 642-645
 - multiprecision addition, 466-468
 - vectorized addition, 195-197
 - adding unaligned arrays, 198-200
 - vec_lvsl function, 197
 - vec_perm function, 197
- addition instructions (assembly instructions), 381-384**
- addition property (FFT), 450**
- addressing modes, 373**
- addTime function, 555**
- addx instruction, 382**
- Adleman, Leonard, 472**
- Advance Cycle button (SystemSim), 60**
- affinity, 147-148**
- ah instruction, 382**
- ahi instruction, 382**
- ai instruction, 382**
- ALF (Accelerated Library Framework)**
 - accelerator memory buffers, 619-620
 - accelerator stages
 - accelerator environment functions, 637
 - implementing with functions, 634-637
 - kernel API export definition section, 637
 - macros, 635
 - overview, 632
 - ALF runtime, 618
 - datatypes, 626
 - events, 646-647
 - initializing ALF environment
 - alf_init function, 620-621
 - alf_num_instances_set function, 623
 - alf_query_system_info function, 621
 - ALF system query fields, 622
 - overview, 620
 - ppu_alf_simple.c example, 622-623
 - libspe and, 618-619
 - matrix addition and subtraction
 - partitioning accelerator data, 644-645
 - partitioning host data, 642-643
 - overview, 617
 - task descriptors
 - adding task contexts to, 625-626
 - creating, 623
 - customizing, 624-625
 - fields, 624-625

- tasks
 - creating, 626-627
 - definition, 618
 - ending, 631
 - launching, 631
 - task dependency, 645-646
 - task instances, 618
 - work blocks, 618
- text transfer and display, 638-642
- work blocks
 - adding parameter contexts to, 628
 - adding to tasks, 631
 - buffer storage, 629
 - creating, 627-628
 - data transfer lists, 628-630
- ALF_ACCEL_DTL_BEGIN macro, 635**
- ALF_ACCEL_DTL_END macro, 635**
- ALF_ACCEL_DTL_ENTRY_ADD macro, 635**
- alf_exit function, 631**
- alf_init function, 620-621**
- alf_instance_id function, 637**
- alf_num_instances function, 637**
- alf_num_instances_set function, 623**
- ALF_QUERY_* fields, 622**
- alf_query_system_info function, 621**
- alf_task_create function, 626-627**
- alf_task_depends_on function, 645**
- ALF_TASK_DESC_* fields, 624**
- alf_task_desc_ctx_entry_add function, 625**
- alf_task_desc_destroy function, 626**
- alf_task_desc_set_int64 function, 634**
- ALF_TASK_EVENT_* types, 646-647**
- alf_task_event_handler_register function, 646-647**
- alf_task_finalize function, 631**
- alf_task_wait function, 631**
- alf_wb_create function, 627-628**
- alf_wb_dtl_begin function, 629**
- alf_wb_dtl_end function, 629-630**
- alf_wb_dtl_entry_add function, 629-630**
- alf_wb_enqueue function, 631**
- alf_wb_parm_add function, 628**
- allocating memory dynamically**
 - functions, 233-234
 - heap, 233
 - spu_heaptest.c application, 234
- Altivec library, 160**
- altivec.h library, 102**
- analysis, frequency analysis**
 - DFT (discrete Fourier transform), 442-447
 - FFT (fast Fourier transform), 448-461
 - frequency-domain representation, 440-441
 - overview, 440
 - signals and sampling, 441-442
 - time-domain representation, 440-441
- and instruction, 386**
- andbi instruction, 386**
- andc instruction, 386**
- andhi instruction, 386**
- andi instruction, 386**
- animation in Ogre3D, 555**
- AnimationState class, 555**
- AoS (array of structures), converting to SoA, 200-203, 278-280**
- Application Binary Interface. See ABI**
- applications, 501. See also *individual applications***
 - building
 - make command, 44
 - makefiles, 44, 59
 - overview, 35-36
 - for PPU (PowerPC Processor Unit), 36-43
 - with ppu-gcc, 42-43
 - for SPU (Synergistic Processor Unit), 43
 - controlling execution of, 56

debugging

overview, 53

with spu-gdb, 54-58

ARB (Architecture Review Board), 504

architecture of Cell processor

EIB (Element Interconnect Bus), 9

IOIF (Input/Output Interface), 9-10

MIC (Memory Interface
Controller), 6-7

overview, 5-6

PPE (PowerPC Processor
Element), 7-8

SPE (Synergistic Processor
Element), 8-9

Architecture Review Board (ARB), 504

arithmetic functions

division functions, 468-471

general arithmetic functions, 464-468

modular exponentiation
functions, 471

for PPU

addition/subtraction functions,
166-170

multiplication/division
functions, 170-174

for SPU

addition/subtraction, 246-250

multiplication/division, 250-253

array `get aname` command, 664

array names `aname` command, 664

**array of structures (AoS), converting to SoA,
200-203, 278-280**

array set `aname list` command, 664

array set command, 72

array size `aname` command, 664

arrays

AoS (array of structures), converting
to SoA, 200-203, 278-280

Tcl arrays, 664-665

unaligned arrays, adding, 198-200

asind2 function, 270

asinf4 function, 190, 270

asm function, 324

assemblers

ppu-as, 40

spu-as, 43

assembly language, 39

addition/subtraction instructions,
381-384

advantages/disadvantages, 368

Application Binary Interface
(ABI), 401

calling C/C++ functions from
assembly code, 405-406

writing assembly-coded functions,
402-405

assembly-coded applications

building, 371-372

debugging, 372-373

bit selection and selection mask

creation instructions, 380-381

branch hinting instructions, 392-393

branch instructions, 390-392

channel and control instructions,
394-395

counting and conversion instructions,
400-401

dual-pipeline architecture, 406-408

halt instructions, 393-394

instruction set reference, 649-659

load immediate instructions, 375-376

load/store instructions, 373-375

logic instructions, 386-387

multiplication instructions, 384-386

overview, 367

rotate instructions, 397-399

sections, creating, 369-370

shift instructions, 395-396

shuffling and mask creation
instructions, 377-380

- simple assembly file example, 370-371
- individual intrinsics, 369
- SPU addressing modes, 373
- vector comparison instructions, 388-389
- assembly-coded applications**
 - building, 371-372
 - debugging, 372-373
- assembly-coded functions, writing**
 - example, 404-405
 - function declaration, 402
 - stack management, 402-403
- asset element (DAE files)**
 - contributor subelement, 561-562
 - created subelement, 562
 - example, 562-563
 - modified subelement, 562
 - table of subelements, 561
 - unit subelement, 562
- atanf4 function, 191, 271**
- atomic_add function, 309**
- atomic_add_return function, 309**
- atomic_dec function, 309**
- atomic_dec_return function, 309**
- atomic DMA (direct memory access), 306-308**
- atomic_inc function, 309**
- atomic_inc_return function, 309**
- atomic_read function, 309**
- atomic_set function, 309**
- atomic_sub function, 309**
- atomic_sub_return function, 309**
- attachObject function, 550**
- author element, 562**
- authoring tool element, 562**
- automatic variables, 50**
- avgb instruction, 382-384**

B

- backslash (\), 41**
- basic unary functions**
 - PPU, 180-183
 - SPU, 263-266
- bg instruction, 382-383**
- bgx instruction, 382**
- bi instruction, 390**
- bihnz instruction, 391**
- bihz instruction, 391**
- binz instruction, 391**
- BIOGET_VBLANK request code (ioctl), 495-496**
- bisl instruction, 390**
- bisled instruction, 391-392**
- bit ordering, 101-102**
- bit selection instructions (assembly instructions), 380-381**
- bit-reversal, 452**
- biz instruction, 391**
- BLAS library (libblas), 411, 430-434**
- block devices, 493-494**
- blocking comments, 88**
- blocks, 493**
- bmove512 function, 204**
- boot configuration script, 614-615**
- bootloaders, kboot, 18**
- Box-Muller method, 484**
- br instruction, 390-391**
- bra instruction, 390**
- branch hinting instructions (assembly instructions), 392-393**
- branch instructions (assembly instructions), 390-392**
- branch misses, preventing, 113**
- branch prediction (PPU pipeline), 110-111**
- branch processing (SPU), 236-237**
- brasl instruction, 390**
- break command, 54-55**

breakpoints, 54-55

brhnz instruction, 390

brhz instruction, 390

brnz instruction, 390

brsl instruction, 390

brz instruction, 390

bubbles (pipeline), 108

buffers

accelerator memory buffers, 619-620

buffer storage for work block data, 629

DMA (direct memory access)

double-buffered DMA, 295-296

single-buffered DMA, 293-294

SPU, 236

TLB (Translation Lookaside Buffer), 123-124

vertex buffer objects, 525-527

Build Project command (Project menu), 90

buildhead process, 354

building applications

Linux kernel, 613-614

for PPU (PowerPC Processor Unit)

build process, 37

PPU header files, 36-37

PPU source files, 36

ppu-as assembler, 40

ppu-cpp preprocessor, 37-38

ppu-gcc compiler, 38-43

ppu-ld linker, 40-42

make command, 44

makefiles

advanced example, 51-52

automatic variables, 50

comments, 47-48

dependency lines, 45-46

overview, 44

pattern rules, 50-51

phony targets, 49

shell lines, 46-47

simple example, 48

structure, 45

variables, 47-48

overview, 35-36

with ppu-gcc, 42-43

projects with Cell IDE, 90

simulated applications, 64

for SPU (Synergistic Processor Unit), 43

spu_basic.c application

with Cell SDK IDE, 129-131

from command line, 129

burning Fedora Core disk image to DVD, 17-18

buses, EIB (Element Interconnect Bus), 286-287

butterfly diagrams, 450

buttons in SystemSim, 60-61

bzero function, 204

C

C standard library, 226-227

C++ standard library, 226-227

C/C++ Development Tooling, 59, 85

cache

accessing

safe software cache

functions, 352-353

unsafe software cache functions, 353-354

cache statistics, 357

configuring, 350-352

DCache, 119-120

heapsort, 354, 356-357

instruction cache

controlling, 112

overview, 110

overview, 349-350

- cache_flush function, 352
- cache_lock function, 353
- CACHE_LOG2NSETS attribute, 352
- CACHE_LOG2NWAY attribute, 351
- CACHE_LOG2SIZE attribute, 352
- CACHE_NAME attribute, 350
- cache_rd function, 352
- cache_rd_x4 function, 352
- CACHE_READ_X4 attribute, 352
- cache_rw function, 353
- CACHE_SET_TAGID attribute, 352
- CACHE_STATS attribute, 352
- cache_touch function, 353
- CACHE_TYPE attribute, 350
- cache_unlock function, 353
- cache_wait function, 353
- cache_wr function, 352
- CACHED_TYPE attribute, 350
- call stack, 230
- calling C/C++ functions from assembly code, 405-406
- callthru command, 64
- callthru sink command, 64-65
- callthru source command, 64-65
- Camera class, 536-537
- capture function, 554
- cast_matrix4x4_to_dbl function, 417
- cast_matrix4x4_toflt function, 418
- cbd instruction, 377
- cbtrd2 function, 189, 268
- cbtrf4 function, 189, 268
- cbx instruction, 377
- cdd instruction, 377
- CDT (C/C++ Development Tooling), installing, 85
- cdx instruction, 377
- ceild2 function, 263
- ceilf4 function, 263
- Cell BE Handbook*, 321
- Cell simulators**
 - running executables on, 93-94
 - SystemSim
 - checkpoints, 68-69
 - command window, 61-63
 - configuring, 59-60
 - console window, 63
 - control buttons, 60-61
 - emitters, 73-81
 - overview, 59
 - running, 60-61
 - simulated applications, compiling and running, 64-65
 - starting, 60-61
 - statistics and profiling commands, 66-68
 - trigger actions, 70-73
 - trigger events, 69-73
- cellsdk script, 22-23
- ceq instruction, 388
- ceqb instruction, 388
- ceqbi instruction, 388
- ceqh instruction, 388
- ceqhi instruction, 388
- ceqi instruction, 388
- CESOF format, 601-602
- cflls instruction, 400
- cflltu instruction, 400
- cg instruction, 382-383
- cgt instruction, 388
- cgtb instruction, 388
- cgtbi instruction, 388
- cgth instruction, 388
- cgthi instruction, 388
- cgti instruction, 388, 392
- cgx instruction, 382
- challenges of modern computing, 581
- channel instructions (assembly instructions), 394-395

channels

- definition, 317
- functions, 318-321
- PPU access to MFC registers, 321
- scholar-butler analogy, 317-318
- SPE channels, 318-320

character devices, 493**chd instruction, 377-379****checking errors, 87****checkpoints in SystemSim, 68-69****chx instruction, 377-379****cipher text (CT), 472****classes**

- AnimationState, 555
- Camera, 536-537
- DAE, 573-574
- daeDatabase, 574-576
- Entity, 549
- FrameListener, 556-557
- InputManager, 554
- Keyboard, 554
- Light, 552-554
- Root, 533-536
- SceneManager, 536
- SceneNode, 549-552
- Viewport, 538

clgt instruction, 389**clgtb instruction, 388****clgtbi instruction, 389****clgth instruction, 389****clgthi instruction, 389****clgti instruction, 389****clipcode_ndc function, 415****clipcode_ndc_v function, 415****clip_ray function, 415-416****clz instruction, 400-401****cntb instruction, 400-401****__cntlzw(int) function, 103****code listings, 202****COLLADA (Collaborative Design Activity)**

- DAE (Digital Asset Exchange) format
 - asset element, 561-563
 - characteristics, 560-561
 - library_controllers element, 567-569
 - library_geometries element, 563-567
 - library_materials element, 570-571
- DAE class, 573-574
- daeDatabase class, 574-576
- dae_basic.cpp sample application, 572-573
- DOM (Document Object Model), 577-580
- library installation, 572
- overview, 559-560

color of vertices, 521-523**color depth, 492****command window (SystemSim), 61-63****commands, 21, 44, 59, 611, 665. See also *individual commands*****comments**

- blocking, 88
- comments element, 561-562
- in makefiles, 47-48
- Tcl comments, 662

comments element, 561-562**communication**

- channels
 - definition, 317
 - functions, 318-321
 - PPU access to MFC registers, 321
 - scholar-butler analogy, 317-318
 - SPE channels, 318-320
- DMA (direct memory access), 285
- events, 322
- with isolated SPU, 363-365

- mailboxes
 - overview, 328
 - PPU mailbox communication functions, 331-332
 - SPU mailbox communication, 328-331
 - SPU-SPU mailbox communication, 333
- multiprocessor synchronization
 - MFC multisource synchronization, 341-342
 - multiprocessor DMA ordering, 340-341
- signals
 - many-to-one communication, 336-337
 - notification channels, 333-335
 - notification modes, 336-337
 - overview, 333
 - PPU signaling, 338
 - read operations, 333-334
 - sending from SPE, 335-336
 - SPE synchronization, 338-340
- compare functions (SPU), 256-260**
- compare instructions (assembly instructions), 388-389**
- compile-time embedding, 133-134**
- compilers**
 - dtc (device tree compiler), 612
 - ppu-gcc, 10, 38-43
 - spu-gcc, 10, 43
- compiling simulated applications, 64**
- complete function, 313**
- complete_all function, 313**
- completion variables, 313**
- concurrency example (spu_cashier.c), 313-315**
- cond_broadcast function, 312**
- cond_init function, 312**
- cond_signal function, 312**
- cond_wait function, 312**
- condition register (CR), 107**
- condition variables, 312**
- configuring**
 - double-precision rounding mode with FPSCR (floating-point status and control register), 283-284
 - emitter events, 73-76
 - Linux kernel, 613-614
 - projects, 89-90
 - PS3 to boot kernel, 614-615
 - SPU software cache, 350-352
 - SystemSim, 59-60
- Connectivity tab (Target Environment Configuration dialog), 93**
- connecting to remote systems, 30**
 - with PuTTY, 31
 - with WinSCP, 31-33
- console window (SystemSim), 63**
- contexts**
 - creating, 135-136
 - definition, 125
 - gang contexts, 136
 - affinity, 147-148
 - creating, 147
 - deallocating, 149
 - loading program handles into, 138
 - parameter contexts
 - adding to work blocks, 628
 - data transfer lists, 628-630
 - task contexts, adding to task descriptors, 625-626
- continue command, 56**
- contributor element, 561-562**
- control instructions (assembly instructions), 394-395**
- conversion functions (PPU), 174-177**
- conversion instructions (assembly instructions), 400-401**

converting

- AoS (array of structures) to SoA, 200-203, 278-280
- rectangular coordinates to polar coordinates, 272
- Cooley, J. W., 439**
- Cooley-Tukey FFT, 439**
- copyright element, 562**
- copysign2 function, 193-194, 274**
- copysignf4 function, 193-194, 274**
- cosd2 function, 191, 270**
- cosf4 function, 191, 270**
- coshd2 function, 191, 270**
- coshf4 function, 191, 270**
- count functions (SPU), 256-260**
- count registers (CTRs), 107**
- counting instructions (assembly instructions), 400-401**
- CR (condition register), 107**
- createAndPlace function, 577**
- createAndPlaceAt function, 577**
- createChild function, 550**
- created element, 562**
- createDocument function, 574-575**
- createElement function, 577**
- createEntity function, 549**
- createRenderWindow function, 535**
- createSceneManager function, 535**
- cross_product3 function, 412**
- cross_product3_v function, 413**
- cross_product4 function, 412**
- cryptology, 472-473**
- Cryptography in C and C++*, 463**
- csflt instruction, 400**
- CT (cipher text), 472**
- CTRs (count registers), 107**
- cuftl instruction, 400**
- customizing task descriptors, 624-625**
- cwd instruction, 377, 380**
- cwx instruction, 377, 380**
- Cygwin, 30**

D

- d-form (indexed addressing), 373**
- DAE (Digital Asset Exchange) format**
 - asset element
 - contributor subelement, 561-562
 - created subelement, 562
 - example, 562-563
 - modified subelement, 562
 - table of subelements, 561
 - unit subelement, 562
 - characteristics, 560-561
 - library_controllers element
 - morph subelement, 569
 - overview, 567
 - skin subelement, 568-569
 - library_geometries element
 - overview, 563
 - shapes, 566-567
 - source subelement, 564-565
 - vertices subelement, 565
 - library_materials element
 - overview, 570
 - setparam subelement, 570-571
 - technique_hint subelement, 570
- dae_basic.cpp sample application, 572-573**
- DAE class, 573-574**
- daeDatabase class, 574-576**
- data partitioning, 628**
 - data on accelerator, 644-645
 - data on host, 642-643
- data transfer**
 - data transfer lists, 628-630
 - vectorization
 - libfreevec library, 204-205
 - libmotivec library, 204-205
 - memcpy function, 206-207
 - strcmp function, 207-208
- datatypes, 100-101**
 - ALF datatypes, 626
 - OpenGL datatypes, 512-513

- scalar datatypes, 223–224
 - vector datatypes, 224
- DCache, 119–120**
- deallocating**
 - data structures, 142
 - gang contexts, 149
- Debug Controls button (SystemSim), 61**
- Debug dialog, 94**
- Debug view (Eclipse), 94**
- debugging**
 - assembly-coded applications, 372–373
 - executables with Cell IDE, 94–96
 - overview, 53
 - with spu-gdb
 - controlling application execution, 56
 - overview, 54
 - reading processor information, 55–56
 - sample debug sessions, 56–58
 - setting breakpoints/watchpoints, 54–55
- declaring functions in assembly, 402**
- decode phase**
 - PPU pipeline, 112
 - SPU, 237
- decrementer, 242–244**
- #define directive, 37**
- Demand Data Load Request Queue (DLQ), 120**
- dependency, 45**
 - ALF task dependency, 645–646
 - dependency lines (makefiles), 45–46
- depth of color, 492**
- destroySceneManager function, 535**
- detachObject function, 550**
- /dev directory, 493**
 - /dev/fb0, 502
- development of Cell processor, 1**
- device files**
 - accessing, 493
 - block devices, 493–494
 - character devices, 493
- device tree compiler (dtc), 612**
- dfm instruction, 386**
- dfma instruction, 385**
- dfms instruction, 385**
- dfnma instruction, 385**
- dfnms instruction, 385**
- DFT (discrete Fourier transform)**
 - DFT matrix, 447
 - equation, 445
 - example, 445–447
 - fundamental frequencies, 443
 - overview, 442
 - single-frequency vectors, 444–445
- diagrams, butterfly, 450**
- dialogs, 59, 91**
- Digital Asset Exchange format. See DAE (Digital Asset Exchange) format**
- digital signal processor (DSP), 11**
- direct memory access, 285**
- direct SPE access**
 - in libsdpe, 151–152
 - overview, 149
 - SPU File System (SPUFS), 149–151
- direction numbers, 480**
- directives**
 - #define, 37
 - #include, 37
 - definition, 370
- directories**
 - /dev, 493
 - /opt/cell/sdk (SDK), 29
 - /opt/cell/systemsim-cell (SDK), 30
 - /opt/cell/toolchain (SDK), 30
 - /usr/bin (SDK), 28–29
- discrete Fourier transform, 442**

dispatch (PPU pipeline), 111

display monitors

- color depth, 492
- pixels, 492
- speed and frame rate, 492-493

distribution of number

sequences, transforming

- Box-Muller method, 484
- Moro inversion, 484
- overview, 483-484
- Polar method, 484
- spu_random2.c example, 486-487
- transformation functions, 485-486

divd2 function, 253

divf4 function, 174, 253

divf4_fast function, 171, 174, 253

divi4 function, 171, 174

division functions, 468-471

- PPU, 170-174
- SPU, 250-253

divu4 function, 171, 174

DLQ (Demand Data Load Request Queue), 120

DMA (direct memory access)

- atomic DMA, 306-308
- DMA between SPUs, 304-306
- double-buffered DMA, 295-296
- EIB (Element Interconnect Bus), 286-287
- MFC (Memory Flow Controller)
 - command queue, checking, 291
 - external access to, 288
 - scholar-butler analogy, 287-288
- overview, 285
- PPU-initiated DMA, 302-304
- request lists
 - list elements, 297-298
 - list functions, 298-301
 - overview, 296-297
- single DMA transfer on SPU, 288-290

single-buffered DMA, 293-294

Synchronization library (libsync)

- atomic functions, 308
- completion variables, 313
- concurrency example (spu_cashier.c), 313-315
- condition variables, 312
- mutexes, 308-310
- reader/writer locks, 310-311

tag groups

- advantages of, 290
- checking for DMA completion, 290-291
- ordering transfers in, 291-292

DOM (Document Object Model), COLLADA DOM, 577-580

dom_create.cpp file, 579-580

dot clock, 492

dot_product3 function, 412-413

dot_product3_v function, 412-413

dot_product4 function, 412-413

dot_product4_v function, 412

double-buffered DMA (direct memory access), 295-296

double-precision rounding mode, 283-284

downheap process, 354

downloading

- Gallium, 506-507
- Mesa, 506-507
- Ogre, 529
- SDK (Software Development Kit), 21

draw function, 499

drawing frame buffer, 498-502

DSP (digital signal processor), 11

dsync instruction, 394-395

dtc (device tree compiler), 612

DTLs (data transfer lists), 628-630

dual issue, improving chances of, 114-115

dual-pipeline architecture, 406-408

dual-threaded operation (PPU)
 intrinsic functions, 118
 multithreading example, 116-118
 thread resources, 115-116

DVDs, burning Fedora Core disk image to, 17-18

dynamic allocation
 functions, 233-234
 heap, 233
 spu_heapttest.c application, 234
dynamic linking, 596

E

Eclipse, 59, 84

editing source code, 87-89

EIB (Element Interconnect Bus), 286-287, 9

ELF (Executable and Linking Format) files

accessing in code, 604-607

CESOF format, 601-602

executable files

dynamic linking, 596

headers, 593-594

segments, 595-596

structure of, 594-595

viewing structure of, 593

libraries

goodbye.c source file, 597

hello.c source file, 596

libcall.c source file, 597

shared libraries, 598-599

static libraries, 597-598

object files

ELF headers, 586

HelloWorld.c example, 585-586

relocation tables, 592-593

section headers, 587-588, 591

symbol and string tables, 591-592

overview, 585

SPU-ELF files

complete PPU executable, 602-604

TOE section, 600-601

elf.h library, 102

elf.simple.c file, 604-605

Elf64_Ehdr structure, 606

Elf64_Shdr structure, 607

embedding SPE executable into PPU application, 133-135

compile-time embedding, 133-134

runtime embedding, 135

EMIT_DATA structure, 76-78

EMITTER_CONTROLS structure, 79

emitters (SystemSim)

emitter readers, 73, 76-81

emitter records, 73

event configuration, 73-76

event types, 73

Emitters button (SystemSim), 61

encryption, 361-362

ending tasks, 631

entities (Ogre), 549

Entity class, 549

eqv instruction, 386

eread.c file, 79-80

ereader_expect command, 74-75

ereader_start command, 74-75

erfcd2 function, 269

erfd2 function, 269

errors

detecting with FPSCR (floating-point status and control register), 281-283

error checking, 87

event handlers

creating, 136-137

definition, 322

Event Log button (SystemSim), 61

events

acknowledging, 327

ALF events, 646-647

- definition, 322
- emitter event configuration, 73-76
- emitter event types, 73
- MFC_COMMAND_QUEUE_AVAILABLE_EVENT, 322
- MFC_DECREMENTER_EVENT, 322
- MFC_IN_MBOX_AVAILABLE_EVENT, 322
- MFC_LIST_STALL_NOTIFY_EVENT, 322
- MFC_LLROST_EVENT, 322
- MFC_MULTI_SRC_SYNC_EVENT, 323
- MFC_OUT_INTR_MBOX_AVAILABLE_EVENT, 322
- MFC_OUT_MBOX_AVAILABLE_EVENT, 322
- MFC_PRIV_ATTN_EVENT, 323
- MFC_SIGNAL_NOTIFY_1_EVENT, 322
- MFC_SIGNAL_NOTIFY_2_EVENT, 322
- MFC_TAG_STATUS_UPDATE_EVENT, 322
- PPE event handling, 327-328
- recognizing
 - event-recognition functions, 323-324
 - interrupt handling, 325-327
 - polling, 324-325
 - waiting, 324
- registering, 136-137
- selecting, 323
- SPE events, 322-323
- trigger events
 - associating trigger actions with, 71-73
 - table of, 69-70
 - waiting for, 139-142
- example_handler function, 647**
- EXCLUDE_FILE statement, 347**

- Executable and Linking Format files, 585**
- executable loading (SPU), 232-233**
- executables**
 - debugging with Cell IDE, 94-96
 - ELF
 - dynamic linking, 596
 - headers, 593-594
 - segments, 595-596
 - structure of, 594-595
 - viewing structure of, 593
 - running with Cell IDE
 - on Cell simulators, 93-94
 - on remote Cell systems, 91-93
 - overview, 90
- execution of applications, controlling, 56**
- Exit button (SystemSim), 61**
- exp2d2 function, 268**
- exp2f4 function, 189, 268**
- expd2 function, 268**
- expf4 function, 189, 268**
- exponent functions**
 - PPU, 188
 - SPU, 268-270
- exponentiation functions, 471**
- eXtreme Data Rate Dynamic Random Access Memory (XDR DRAM), 6**

F

- fa/dfa instruction, 382**
- fabsd2 function, 263**
- fabsf4/d2 function, 263**
- fast Fourier transform, 439**
- FBIO_IOCTL_FSEL request code (ioctl), 496**
- FBIO_WAITFORVSYNC request code (ioctl), 496**
- fceq instruction, 389**
- fcgt instruction, 389**
- fcmeq instruction, 389**
- fcmgt instruction, 389**

- fdimd2 function, 247, 250**
- fdimf4 function, 167, 247, 250**
- FDRs (floating-point registers), 106**
- Fedora Core, installing on PlayStation 3**
 - burning Fedora Core disk image to DVD, 17-18
 - obtaining kboot bootloader, 18
 - overview, 16
 - step-by-step installation process, 18-20
- feedback shift registers (FSRs), 475-476**
- fesd instruction, 400**
- FFT (fast Fourier transform)**
 - addition property, 450
 - four-point Fourier transforms, 450-452
 - libfft library, 458-460
 - libfft_example library
 - one-dimensional FFT, 452-454
 - overview, 452
 - two-dimensional FFT, 454-457
 - libfft_spu library, 460-461
 - overview, 439, 448
 - shifting property, 449
 - stretching property, 448-449
 - two-point Fourier transforms, 450-452
- fft_1d_c2r_spu function, 461**
- fft_1d_initialize function, 459**
- fft_1d_r2 function, 452-454**
- fft_1d_r2c_spu function, 461**
- fft_2d function, 454-457**
- fft_2d_initialize function, 459**
- fft_2d_perform function, 460**
- fi instruction, 400-401**
- fields**
 - ALF system query fields, 622
 - ALF task descriptor fields, 624-625
- file systems, SPUFS (SPU File System), 149, 151**
- files, 79. See also individual files**
 - DAE (Digital Asset Exchange) format
 - asset element, 561-563
 - characteristics, 560-561
 - library_controllers element, 567-569
 - library_geometries element, 563-567
 - library_materials element, 570-571
 - dae_basic.cpp, 572-573
 - device files
 - accessing, 493
 - block devices, 493-494
 - character devices, 493
 - dom_create.cpp, 579-580
 - ELF (Executable and Linking Format) files
 - accessing in code, 604-607
 - CESOF format, 601-602
 - dynamic linking, 596
 - executable files, 593
 - headers, 593-594
 - libraries, 596-599
 - object files, 585-593
 - overview, 585
 - segments, 595-596
 - SPU-ELF files, 600-604
 - structure of, 594-595
 - makefiles
 - advanced example, 51-52
 - automatic variables, 50
 - comments, 47-48
 - dependency lines, 45-46
 - overview, 44
 - pattern rules, 50-51
 - phony targets, 49
 - shell lines, 46-47
 - simple example, 48
 - structure, 45
 - variables, 47-48

- *.material, 544-545
- Ogre resource configuration files, 547-548
- PPU register files, 106-108
- finish command, 56**
- first harmonics, 443**
- first-level interrupt handler (FLIH), 327**
- fixed-point exception register (XER), 107**
- Fixed-Point Unit (FXU), 105**
- FlexIO, 9**
- FLIH (first-level interrupt handler), 327**
- flipping frame buffer, 494**
- floating-point analysis functions**
 - PPU, 193-194
 - SPU, 273-276
- floating-point processing, 224-225**
- floating-point registers (FDRs), 106**
- floating-point status and control register.**
See **FPSCR**
- Floating-Point Unit (FPU), 156-158**
- flood2 function, 263**
- floorf4 function, 263**
- fm instruction, 386**
- fm/dfm instruction, 385**
- fma instruction, 385**
- fmaxd2 function, 259**
- fmaxd2 function, 259**
- fmaxf4 function, 185, 259**
- fmind2 function, 259**
- fminf4 function, 186, 259**
- fmodf4 function, 171, 174**
- fms instruction, 385**
- fnms instruction, 385**
- for statement (Tcl), 665-666**
- foreach command, 665**
- foreach statement (Tcl), 665-666**
- four-point Fourier transforms, 450-452**
- Fourier transforms, 448**
- FPSCR (floating-point status and control register), 107, 221-222**
 - configuring double-precision rounding mode with, 283-284
 - detecting errors with, 281-283
 - overview, 280
- FPU (Floating-Point Unit), 156-158**
- frame buffer**
 - accessing, 494
 - definition, 494
 - drawing, 498-502
 - flipping, 494
 - frames, 494
 - ioctl instructions
 - ioctl parameters, 494-495
 - Linux ioctl request codes, 495-496
 - PS3 ioctl request codes, 496-498
 - memory mapping, 494
 - overview, 491
- frame rate, 492-493**
- frameEnded function, 556**
- FrameListener class, 556-557**
- frameRenderingQueued function, 556**
- frames, 494**
- frameStarted function, 556**
- frds instruction, 400**
- FreeImage, installing, 530**
- frequency analysis, 440**
 - DFT (discrete Fourier transform)
 - DFT matrix, 447
 - equation, 445
 - example, 445-447
 - fundamental frequencies, 443
 - overview, 442
 - single-frequency vectors, 444-445
 - FFT (fast Fourier transform)
 - addition property, 450
 - four-point Fourier transforms, 450-452
 - libfft library, 458-460

- libfft_example library, 452-457
- libfft_spu library, 460-461
- overview, 448
- shifting property, 449
- stretching property, 448-449
- two-point Fourier transforms, 450-452
- frequency-domain representation, 440-441
- frequency vectors, 442
- fundamental frequencies, 443
- overview, 440
- signals and sampling, 441-442
- single-frequency vectors, 444-445
- time-domain representation, 440-441
- frequency vectors, 442**
- frequency-domain representation, 440-441**
- frest instruction, 385, 401**
- frexp2 function, 274-275**
- frexp4 function, 193-194, 274**
- frsqst instruction, 385, 401**
- frustum_matrix4x4 function, 420-422**
- fs/dfs instruction, 382**
- fscrrd instruction, 394-395**
- fscrwr instruction, 394-395**
- fsm instruction, 380**
- fsmb instruction, 380**
- fsmbi instruction, 380-381**
- fsmh instruction, 380-381**
- FSRs (feedback shift registers), 475-476**
- Full-System Simulator, 10, 59**
- functional units, 105, 220-221**
 - LSU (Load Store Unit)
 - DCache and memory synchronization, 119-120
 - overview, 118-119
 - PPSS (PowerPC Processor Storage Subsystem), 120

- MMU (Memory Management Unit)
 - overview, 121
 - TLB (Translation Lookaside Buffer), 123-124
 - virtual memory and segments, 122-123

functions. See individual functions

fundamental frequencies, 443

FXU (Fixed-Point Unit), 105

G

Gallium

- building libraries, 507-508
- downloading, 506-507
- overview, 505-506

game development, 529

GameOS, 16

gang contexts, 136

- affinity, 147-148
- creating, 147
- deallocating, 149

gb instruction, 400-401

gbb instruction, 400-401

gbh instruction, 400-401

gcd function, 470

GElf_Ehdr structure, 605

gelf_getehdr() function, 605

genb function, 250

genbx function, 250

general purpose registers (GPRs), 106, 221

generalized feedback shift registers (GFSRs), 476

Gerstner, Louis, 3

getActualHeight function, 538

getActualLeft function, 538

getActualTop function, 538

getActualWidth function, 538

getAsset function, 577

getAttributeValue, 578

- getAvailableRenderers function, 535
- getChild function, 550
- getChildIterator function, 550
- getDatabase function, 574
- getDocument function, 574-575
- getElement function, 574-575
- getElementCount function, 574-575
- getElementName, 578
- getExtra function, 577
- getID, 578
- getLibrary function, 577
- getScene function, 577
- getSceneManagerMetaData function, 535
- getType function, 552
- getTypeName, 578
- getXmlns function, 577
- GFSRs (generalized feedback shift registers), 476
- git utility, 506, 612
- GL_LINES constant, 519
- GL_LINE_LOOP constant, 519
- GL_LINE_STRIP constant, 519
- GL_POINTS constant, 519
- GL_POLYGON constant, 519
- GL_QUADS constant, 519
- GL_QUAD_STRIP constant, 519
- GL_TRIANGLES constant, 519
- GL_TRIANGLE_FAN constant, 519
- GL_TRIANGLE_STRIP constant, 519
- glBegin function, 519-520
- glBindBuffer function, 525
- GLbitfield datatype, 513
- GLboolean datatype, 512
- glBufferData function, 526
- glBufferSubData function, 526
- GLbyte datatype, 512
- GLclampd datatype, 513
- GLclampf datatype, 513
- glClear function, 516
- glClearColor function, 516
- glColor3f function, 521-522
- glColorPointer function, 526
- glDeleteBuffers function, 526
- GLdouble datatype, 513
- glDrawArrays function, 526
- glEnableClientState function, 525
- GLenum datatype, 513
- GLfloat datatype, 513
- glFrustum function, 515-516
- glGenBuffers function, 525
- glibc-kernheaders package, 610
- GLint datatype, 512
- glLoadIdentity function, 516
- glMatrixMode function, 516
- glNormal* function, 524
- glNormalPointer function, 526
- glOrtho function, 515-516
- glPerspective function, 515
- glPopMatrix function, 516
- glPushMatrix function, 516
- GLshort datatype, 512
- GLsizei datatype, 512
- GLubyte datatype, 512
- GLuint datatype, 513
- gluLookAt function, 516
- gluOrtho2Dd function, 516
- gluPerspective function, 516
- GLushort datatype, 512
- GLUT (OpenGL Utility Toolkit)
 - creating windows with, 508-512
 - functions, 508-510
 - overview, 508
- glutCreateWindow function, 509
- glutDestroyWindow function, 509
- glutDisplayFunc function, 509
- glutFullScreen function, 509
- glutInit function, 508
- glutInitDisplayMode function, 508
- glutInitWindowPosition function, 508

glutInitWindowSize function, 508
glutKeyboardFunc function, 509
glutMainLoop function, 509
glutMouseFunc function, 509
glutSolidSphere function, 509
glutSolidTeapot function, 509
glutSwapBuffers function, 509
glutWireSphere function, 509
glutWireTeapot function, 509
glVertexPointer function, 526
GNU General Public License (GPL), 27-28
Go button (SystemSim), 60
goodbye.c source file, 597
GPL (GNU General Public License), 27-28
GPRs (general purpose registers), 106, 221
graphics

- display monitors
 - color depth, 492
 - pixels, 492
 - speed and frame rate, 492-493
- frame buffer
 - accessing, 494
 - definition, 494
 - drawing, 498-502
 - flipping, 494
 - frames, 494
 - ioctl instructions, 494-498
 - memory mapping, 494
 - overview, 491
- OpenGL
 - Architecture Review Board (ARB), 504
 - Gallium, 505-508
 - Khronos Group, 504-505
 - Mesa, 505-508
 - overview, 503-504
- packaging with COLLADA
 - DAE (Digital Asset Exchange) format, 560, 567-571
 - DAE class, 573-574

- daeDatabase class, 574-576
- dae_basic.cpp sample application, 572-573
- DOM (Document Object Model), 577-580
- library installation, 572
- overview, 559-560
- graphics rounding mode, 156**
- groups, tag groups**
 - advantages of, 290
 - checking for DMA completion, 290-291
 - ordering transfers in, 291-292

H

- halt instructions (assembly instructions), 393-394**
- Hamilton, William, 423**
- handling events, 327-328**
- Hardware Implementation Register 1 (HID1), 157**
- Hardware tab (Target Environment Configuration dialog), 93**
- hasAttribute, 578**
- hbr instruction, 392**
- hbra instruction, 392-393**
- hbr instruction, 392**
- headers**
 - for ELF files
 - ELF headers, 586
 - program headers, 593-594
 - section headers, 587-591
 - PPU, 36-37
- heap, 233, 354**
- heapdown function, 356**
- heapdown process, 354**
- heapify process, 354**
- heapsort, 354-357**
- heapup process, 354**
- hello.c source file, 596**

HelloWorld.c file, 585-586

- ELF header, 586
- relocation tables, 592-593
- section header, 587-591
- symbol and string tables, 591-592

heq instruction, 393**heqi instruction, 393****hgt instruction, 393****hgti instruction, 393****HID1 (Hardware Implementation Register 1), 157****hint-for-branch instructions (assembly instructions), 392-393****history of Cell processor, 2-4****hlgt instruction, 393****hlti instruction, 393****Hofstee, H. Peter, 4****horizontal sync frequency, 493****hugepages, 124****Hyde, Randall, 5****hypotd2 function, 271**

|

I-ERAT (Instruction Effective-to-Real Address Translation), 110**IBM**

- code conventions, 144
- Full-System Simulator, 10, 59
- ILAR (International License Agreement for Early Release of Programs), 27
- Roadrunner supercomputer design, 3

IDE (Integrated Development Environment)

- CDT (C/C++ Development Tooling), installing, 85
- cell projects
 - adding/editing source code, 87-89
 - building, 90
 - configuring, 89-90
 - creating, 87

debugging executables with, 94-96

Eclipse, installing, 84

installing, 86

overview, 83

running executables with

on Cell simulators, 93-94

on remote Cell systems, 91-93

overview, 90

Idei, Nobuyuki, 2**identity_matrix4x4 function, 417****if...elseif...else statement, 662-663****il instruction, 375****ila instruction, 375-376****ILAR (International License Agreement for Early Release of Programs), 27****ilh instruction, 375-376****ilhu instruction, 376****ilogbd2 function, 274****ilogbf4 function, 193-194, 274****#include directive, 37****index_max_abs_col function, 427****index_max_abs_vec function, 427****indexed addressing (d-form), 373****indexed register indirect addressing (x-form), 373****Indigo, 504****info command, 55-56****init_completion function, 313****_init_fft_2d function, 454****initialise function, 535****initialisePlugins function, 534****initializing**

ALF (Accelerated Library Framework) environment

ALF system query fields, 622

alf_init function, 620-621

alf_num_instances_set function, 623

alf_query_system_info function, 621

- overview, 620
- ppu_alf_simple.c example, 622-623
- SPU (Synergistic Processor Unit), 230-232
- initResources function, 548**
- inout buffer, 620**
- input buffer, 620**
- input element, 566**
- input/output control (ioctl) instructions**
 - ioctl parameters, 494-495
 - Linux frame buffer ioctl request codes, 495-496
 - PS3 frame buffer ioctl request codes, 496-498
- Input/Output Interface (IOIF), 9-10**
- input, responding to in Ogre3D, 554-555**
- InputManager class, 554**
- insertDocument function, 574-575**
- insertElement function, 574-575**
- insertion sorts, vectorized**
 - intervector sort, 209-210
 - intravector sort, 210-211
 - overview, 208-209
 - vecsort.c example, 211-215
- install function (cellsdk), 22**
- installing**
 - add-on packages, 611
 - CDT (C/C++ Development Tooling), 85
 - Cell IDE (Integrated Development Environment), 86
 - COLLADA libraries, 572
 - dtc (device tree compiler), 612
 - Eclipse, 84
 - Fedora Core on PlayStation 3
 - burning Fedora Core disk image to DVD, 17-18
 - obtaining kboot bootloader, 18
 - overview, 16
 - step-by-step installation process, 18-20
 - FreeImage, 530
 - Ogre, 530
 - SDK (Software Development Kit)
 - optional packages, 23, 26-27
 - preparing for installation, 21-22
 - with cellsdk script, 22-23
- instruction buffers (PPU), 111**
- instruction cache (PPU pipeline)**
 - controlling, 112
 - overview, 110
- Instruction Effective-to-Real Address Translation (I-ERAT), 110**
- instruction pipeline (SPU)**
 - branch procession and prediction, 236-237
 - decode stage, 237
 - issue stage, 237-238
 - overview, 235-236
 - prefetch and buffering, 236
- instruction processing (PPU)**
 - branch prediction, 110-111
 - decode phase, 112
 - instruction buffers and dispatch, 111
 - instruction cache
 - controlling, 112
 - overview, 110
 - issue phase, 112
 - microcode translation, 111-112
 - overview, 108-110
 - pipeline bubbles, 108
- instruction set reference, 649-659**
- Instruction Unit (IU), 105**
- Integrated Development Environment, 59, 83**
- International License Agreement for Early Release of Programs (ILAR), 27**
- interrupt handling, 325-327**
- Interrupt Return (iret) instruction, 327**
- interrupt_service function, 325**
- intersect_ray1_triangle8_v function, 416**

intersect_ray8_triangle1_v function, 416
intersect_ray_triangle function, 415-416
intervector sort, 209-210
intravector sort, 210-211
intrinsic functions
 for dual-threaded applications, 118
 PPU, 103
intrinsics (SPU), 227
inv_length_vec3 function, 413-414
inv_length_vec3_v function, 413-414
inverse_matrix4x4 function, 417
ioctl instructions
 ioctl parameters, 494-495
 Linux frame buffer ioctl request codes, 495-496
 PS3 frame buffer ioctl request codes, 496-498
iohl instruction, 375-376
IOIF (Input/Output Interface), 9-10
iret (Interrupt Return) instruction, 327
is0denormd2 function, 274
is0denormf4 function, 193, 274
isamax_spu function, 432
isDocumentLoaded function, 574-575
isequald2 function, 257
isequalf4 function, 185, 257
isfinited2 function, 274
isfinitef4 function, 193, 274
isgreaterequald2 function, 185, 257
isgreaterqualf4 function, 185, 257
isgreaterd2 function, 257
isgreaterf4 function, 185, 257
isinf2 function, 274
isinf4 function, 193, 274
isInitialised function, 535
isKeyDown function, 554
isless equalf4/d2 function, 257
isless greaterf4/d2 function, 257
islessequalf4 function, 185
islessd2 function, 257

islessf4 function, 185, 257
islessgreaterf4 function, 185
isnand2 function, 274
isnanf4 function, 193, 274
isnormald2 function, 274
isnormalf4 function, 193, 274
isolated SPU, communicating with, 363-365
isolation library (SPU), 362-365
issue phase
 PPU pipeline, 112
 SPU, 237-238
IU (Instruction Unit), 105

J-K

Java mode, 157-158

Kahle, Jim, 4

kboot, 18

kernel

accessing kernel source code, 612-613
 API export definition section, 637
 boot configuration script, 614-615
 building, 613-614
 configuring, 613-614

Keyboard class, 554

keys

public key cryptography, 472-473
 SPU executables, 358-359

keywords element, 561

Khronos Group, 504-505

KS (Kirkpatrick-Stoll) method, 476

Kutaragi, Ken, 3

L

lappend command, 663

Large Matrix library (liblarge_matrix)

basic functions, 427-428
 linear equation solutions, 428-430
 overview, 411, 426-427

- Launch tab (Target Environment Configuration dialog), 93
- launching tasks, 631
- ldconfig command, 598
- length_vec3 function, 413-414
- length_vec3_v function, 413-414
- lerp_vec2_v function, 415
- lerp_vec3_v function, 415
- lerp_vec4 function, 415
- lerp_vec4_v function, 415
- Lesser GNU General Public License (LGPL), 28
- Levand, Geoff, 18
- LFSRs (linear-feedback shift registers), 475-476
- lgammad2 function, 269
- LGPL (Lesser GNU General Public License), 28
- libblas library, 411, 430-434
- libcall.c source file, 597
- libfft library, 458-460
- libfft_example library
 - one-dimensional FFT, 452-454
 - overview, 452
 - two-dimensional FFT, 454-457
- libfft_spu library, 460-461
- libfreevec library, 204-205
- liblarge_matrix library
 - basic functions, 427-428
 - linear equation solutions, 428-430
 - overview, 411, 426-427
- libmatrix library
 - basic functions, 417-418
 - matrix multiplication, 418-419
 - orthogonal projections, 420-421
 - overview, 411, 416-417
 - perspective projection, 421-422
 - projection functions, 419-420
 - spu_matmul.c example, 418-419
 - vector rotation
 - functions, 422
 - quaternions, 424
 - rotating coordinates in 2D and 3D, 423-424
 - spu_rotate.c example, 424-426
- libmc_rand library
 - distribution of number sequences, transforming
 - Box-Muller method, 484
 - Moro inversion, 484
 - overview, 483-484
 - Polar method, 484
 - spu_random2.c example, 486-487
 - transformation functions, 485-486
 - overview, 473-474
 - pseudo-random numbers, generating
 - FSRs (feedback shift registers), 475-476
 - functions, 477-478
 - Kirkpatrick-Stoll method (KS), 476
 - Mersenne Twister (MT) method, 477
 - overview, 474-475
 - spu_random1.c example, 478-480
 - quasi-random numbers, generating
 - functions, 481-483
 - overview, 474-475
 - Sobol generator, 480
- libmotovec library, 204-205
- libmpm (Multiprecision Mathematics Library)
 - division functions, 468-471
 - general arithmetic/logic functions, 464-468
 - modular exponentiation functions, 471
 - overview, 463-464
 - public key cryptography, 472-473
- libraries. *See individual libraries*

library_controllers element (DAE files)

- morph subelement, 569
- overview, 567
- skin subelement, 568-569

library_geometries element (DAE files)

- overview, 563
- shapes, 566-567
- source subelement, 564-565
- vertices subelement, 565

library_materials element (DAE files)

- overview, 570
- setparam subelement, 570-571
- technique_hint subelement, 570

libspe library, 132

- ALF (Accelerated Library Framework) and, 618-619
- direct SPE access
 - functions, 151-152
 - overview, 149
 - SPU File System (SPUFS), 149-151
- overview, 125, 127
- pthreads
 - compared to processes, 143
 - functions, 143-144
 - IBM code conventions, 144
 - overview, 142-143
 - ppu_threads.c example, 145-147
- secure contexts, 360-361
- spe_context_create function, 135-136
- spe_context_destroy function, 142
- spe_context_run function, 138
- spe_cpu_info_get function, 132-133
- spe_event_handler_create function, 136-137
- spe_event_handler_destroy function, 142
- spe_event_wait function, 139
- spe_image_close function, 142
- spe_image_open function, 135

- spe_stop_info_read function, 139-140
- spu_basic.c example
 - building from command line, 129
 - building with Cell SDK IDE, 129-131
 - code listing, 127
 - PPU code to launch, 128-129

libspe2.h library, 102**libsyntax library**

- atomic functions, 308
- completion variables, 313
- concurrency example (spu_cashier.c), 313-315
- condition variables, 312
- mutexes, 308-310
- reader/writer locks, 310-311

libvector library

- functions for vector products and lengths, 412-414
- graphic and miscellaneous functions, 414-416
- overview, 411-412

licensing

- GPL (GNU General Public License), 27-28
- ILAR (International License Agreement for Early Release of Programs), 27
- LGPL (Lesser GNU General Public License), 28

Light class, 552-554**lights, adding to scenes, 552-554****lindex command, 663****line numbering, 87****linear equation solutions, 428-430****linear-feedback shift registers (LFSRs), 475-476****lines element, 566****linestrips element, 566****link registers (LRs), 106**

- linker script, defining SPU overlays in, 346-347**
- linkers**
 - ppu-ld, 40-42
 - spu-ld, 43
- linking, dynamic, 596**
- Linux**
 - frame buffer**
 - accessing, 494
 - definition, 494
 - drawing, 498-502
 - flipping, 494
 - frames, 494
 - ioctl instructions, 494-498
 - memory mapping, 494
 - overview, 491
 - installing on PlayStation 3
 - burning Fedora Core disk image to DVD, 17-18
 - obtaining kboot bootloader, 18
 - overview, 16
 - step-by-step installation process, 18-20
 - kernel**
 - accessing kernel source code, 612-613
 - boot configuration script, 614-615
 - building, 613-614
 - configuring, 613-614
- list command, 663**
- list processing (Tcl), 663-664**
- lists**
 - data transfer lists, 628-630
 - DMA request lists
 - list elements, 297-298
 - list functions, 298-301
 - overview, 296-297
- llabsi2 function, 263**
- llength command, 663**
- llhu instruction, 375**
- llrintf4/d2 function, 263**
- llroundd2 function, 263**
- llroundf4 function, 263-265**
- lnop instruction, 394-395**
- Load and Store Unit (LSU), 105**
- load functions, 163, 166, 574**
- load immediate instructions (assembly instructions), 375-376**
- load/store instructions, 373-375, 649-659**
- load_vec_float4 function, 414-415**
- load_vec_int4 function, 414**
- loading**
 - program handle into context, 138
 - SPU executable, 232-233
 - SPU overlays, 346
- loadPlugins function, 534**
- local store (LS), 227-229**
- lock lines, 307**
- locks, reader/writer, 310-311**
- log10d2 function, 269**
- log10f4 function, 190, 269**
- log2d2 function, 269**
- log2f4 function, 190, 269**
- logarithm functions**
 - PPU, 188
 - SPU, 268-270
- logbf4 function, 193-194, 274**
- logd2 function, 269**
- logf4 function, 190, 269**
- logic functions, 183, 266-267, 464-468**
- logic instructions (assembly instructions), 386-387**
- loops**
 - Tcl loop iteration, 665-666
 - unrolling, 200
- lqa instruction, 374**
- lqd instruction, 374**
- lqr instruction, 374-375**
- lqx instruction, 374**
- lrange command, 663**

Ireplace command, 663

LRs (link registers), 106

LS (local store), 227-229

Iset command, 663

LSU (Load Store Unit)

DCache and memory
synchronization, 119-120

definition, 105

overview, 118-119

PPSS (PowerPC Processor Storage
Subsystem), 120

lu2_decomp function, 429

lu3_decomp_block function, 429

M

machine type, 59

macros

ALF_ACCEL_DTL_BEGIN, 635

ALF_ACCEL_DTL_END, 635

ALF_ACCEL_DTL_ENTRY_
ADD, 635

MEMCOPY_FWD_LOOP_
QUADWORD_ALTIVEC_
ALIGNED, 206

MEMCOPY_FWD_LOOP_
QUADWORD_ALTIVEC_
UNALIGNED, 206

MEMCOPY_FWD_UNTIL_
DEST_IS_ALTIVEC_
ALIGNED, 206

MEMCOPY_FWD_UNTIL_
DEST_IS_WORD_ALIGNED, 206

MEMCPY_FWD_NIBBLE, 207

MEMCPY_FWD_REST_
WORDS_UNALIGNED, 207

StageMISCmod, 437

STRCMP_LOOP_ALTIVEC_
WORD_UNALIGNED, 207

STRCMP_QUADWORD_
UNALIGNED, 208

STRCMP_SINGLE_ALTIVEC_
WORD_UNALIGNED, 207

STRCMP_UNTIL_SRC1_IS_
ALTIVEC_ALIGNED, 207

STRCMP_UNTIL_SRC1_WORD_
ALIGNED, 207

madd_matrix_matrix function, 428

madd_number_vector function, 427

madd_vector_vector function, 427

mailboxes

overview, 328

PPU mailbox communication
functions, 331-332

SPU mailbox communication

functions, 328-329

mailbox event processing, 330-331

mailbox read, 330

mailbox write, 329-330

SPU-SPU mailbox

communication, 333

make clean command, 52

make command, 44

makefiles

advanced example, 51-52

automatic variables, 50

comments, 47-48

dependency lines, 45-46

overview, 44

pattern rules, 50-51

phony targets, 49

shell lines, 46-47

simple example, 48

structure, 45

variables, 47-48

management process (SPE)

creating contexts, 135-136

creating event handlers/registering
events, 136-137

deallocating data structures, 142

embedding SPE executable into PPU
application, 133-135

compile-time embedding, 133-134

runtime embedding, 135

- examining system configuration, 132-133
- loading program handle into context, 138
- overview, 132
- waiting for events and analyzing stop information, 139-142
- many-to-one communication, 336-337**
- MASSV (Mathematics Acceleration Subsystem for Vectors), 102, 161-162, 242**
- *.material files, 544-545**
- materials (Ogre), 544-547**
- math functions**
 - division functions, 468-471
 - general arithmetic functions, 464-468
 - modular exponentiation functions, 471
- Math library (SIMD), 160, 241-242**
- Mathematics Acceleration Subsystem for Vectors (MASSV), 102, 161-162, 242**
- Matlnit_MxM function, 436**
- MatMult_MxM function, 436**
- matrices**
 - addition and subtraction, 642-645
 - Large Matrix library (liblarge_matrix)
 - basic functions, 427-428
 - linear equation solutions, 428-430
 - overview, 411, 426-427
 - Matrix library (libmatrix)
 - basic functions, 417-418
 - matrix multiplication, 418-419
 - orthogonal projections, 420-421
 - overview, 411, 416-417
 - perspective projection, 421-422
 - projection functions, 419-420
 - spu_matmul.c example, 418-419
 - vector rotation, 422-426
- multiprocessor matrix multiplication, 434-438
- projection matrices
 - orthogonal projections, 420-421
 - perspective projection, 421-422
 - projection functions, 419-420
- Matrix library (libmatrix)**
 - basic functions, 417-418
 - matrix multiplication, 418-419
 - orthogonal projections, 420-421
 - overview, 411, 416-417
 - perspective projection, 421-422
 - projection functions, 419-420
 - spu_matmul.c example, 418-419
 - vector rotation
 - functions, 422
 - quaternions, 424
 - rotating coordinates in 2D and 3D, 423-424
 - spu_rotate.c example, 424-426
- matrix_mul file, 434-438**
- mc_rand_hw_array_u4 function, 478**
- mc_rand_hw_init function, 477**
- mc_rand_hw_u4 function, 478**
- mc_rand_ks_0_to_1_array_f4 function, 478**
- mc_rand_ks_0_to_1_f4 function, 478**
- mc_rand_ks_init function, 477**
- mc_rand_mt_init function, 477**
- mc_rand_mt_minus1_to_1_array_d2 function, 478**
- mc_rand_mt_minus1_to_1_d2 function, 478**
- mc_rand_sb_0_to_1_d2 function, 482**
- mc_rand_sb_0_to_1_f4 function, 482**
- mc_rand_sb_array_u4 function, 482**
- mc_rand_sb_init function, 481**
- mc_rand_sb_minus1_to_1_d2 function, 482**
- mc_rand_sb_minus1_to_1_f4 function, 482**
- mc_rand_sb_seed function, 482**
- mc_rand_sb_u4 function, 482**
- mc_rand_XX_0_to_1_array_d2 function, 477**
- mc_rand_XX_0_to_1_array_f4 function, 477**

mc_rand_XX_0_to_1_d2 function, 477
mc_rand_XX_0_to_1_f4 function, 477
mc_rand_XX_array_u4 function, 477
mc_rand_XX_minus1_to_1_array_d2 function, 477
mc_rand_XX_minus1_to_1_array_f4 function, 477
mc_rand_XX_minus1_to_1_d2 function, 477
mc_rand_XX_minus1_to_1_f4 function, 477
mc_rand_XX_u4 function, 477
mc_transform_bm2_array_d2 function, 485
mc_transform_bm2_array_f4 function, 485
mc_transform_mi_array_d2 function, 485
mc_transform_mi_f4 function, 485
mc_transform_po_array_d2 function, 485
mc_transform_po_array_f4 function, 485
mc_transform_po_d2 function, 485
mc_transform_po_f4 function, 485
memccpy function, 204
memchr function, 204
memcmp function, 204
MEMCOPY_FWD_LOOP_QUADWORD_ALTIVEC_ALIGNED macro, 206
MEMCOPY_FWD_LOOP_QUADWORD_ALTIVEC_UNALIGNED macro, 206
MEMCOPY_FWD_UNTIL_DEST_IS_ALTIVEC_ALIGNED macro, 206
MEMCOPY_FWD_UNTIL_DEST_IS_WORD_ALIGNED macro, 206
memcpy function, 205-207
MEMCPY_FWD_NIBBLE macro, 207
MEMCPY_FWD_REST_WORDS_UNALIGNED macro, 207
memfrob function, 205
memmove function, 205
memory
 accelerator memory buffers, 619-620
 dynamic allocation
 functions, 233-234
 heap, 233
 spu_heapttest.c application, 234

 loading SPU overlays into, 346
 SPU software cache
 accessing, 352-354
 cache statistics, 357
 configuring, 350-352
 heapsort, 354-357
 overview, 349-350
 synchronizing, 119-120
 unaligned memory, accessing, 195-197
 adding unaligned arrays, 198-200
 vec_lvsl function, 197
 vec_perm function, 197
 virtual, 122-123

Memory Flow Controller. See MFC

Memory Interface Controller (MIC), 6-7

Memory Management Unit. See MMU

memory mapping, 494

memory synchronization, 229

memcpy function, 205

memchr function, 205

memset function, 205

menuconfig command, 613

Mersenne Twister (MT) method, 477

Mesa

 building libraries, 507-508

 downloading, 506-507

 overview, 505

meshes (Ogre), 538, 540-541

MFC (Memory Flow Controller)

 command queue, checking, 291

 external access to, 288

 multisource synchronization, 341-342

 PPU access to MFC registers, 321

 scholar-butler analogy, 287-288

mfc_ceil128 function, 302

MFC_DECREMENTER_EVENT event, 322

mfc_ea2h function, 302

mfc_ea2l function, 302

mfc_eieio function, 341

- `mfc_get` function, 288
- `mfc_getb` function, 292
- `mfc_getf` function, 292
- `mfc_getl` function, 299
- `mfc_getlb` function, 299
- `mfc_getlf` function, 299
- `mfc_getllar` function, 306-307
- `mfc_hl2ea` function, 302
- `MFC_IN_MBOX_AVAILABLE_EVENT` event, 322
- `mfc_list_element` data structures, 298
- `MFC_LIST_STALL_NOTIFY_EVENT` event, 322
- `MFC_LLRLOST_EVENT` event, 322
- `MFC_MULTI_SRC_SYNC_EVENT` event, 323
- `MFC_OUT_INTR_MBOX_AVAILABLE_EVENT` event, 322
- `MFC_OUT_MBOX_AVAILABLE_EVENT` event, 322
- `MFC_PRIV_ATTENTION_EVENT` event, 323
- `mfc_put` function, 288
- `mfc_putb` function, 292
- `mfc_putf` function, 292
- `mfc_putl` function, 299
- `mfc_putlb` function, 299
- `mfc_putlf` function, 299
- `mfc_putllc` function, 306-307
- `mfc_putlluc` function, 306-307
- `mfc_putqlluc` function, 306-307
- `mfc_read_atomic_status` function, 306-307
- `mfc_read_tag_mask` function, 290
- `mfc_read_tag_status_all` function, 289, 291, 307, 335
- `mfc_read_tag_status_any` function, 291
- `mfc_read_tag_status_immediate` function, 291
- `MFC_SIGNAL_NOTIFY_1_EVENT` event, 322
- `MFC_SIGNAL_NOTIFY_2_EVENT` event, 322
- `mfc_sndsig` function, 335-336
- `mfc_stat_cmd_queue` function, 291
- `mfc_stat_multi_src_sync_request` function, 342
- `mfc_sync` function, 341
- `MFC_TAG_STATUS_UPDATE_EVENT` event, 322
- `mfc_write_multi_src_sync_request` function, 342
- `mfc_write_tag_mask` function, 335
- `mfspr` instruction, 394
- `__mftb()` function, 103
- `MIC` (Memory Interface Controller), 6-7
- microcode translation (PPU), 111-112
- microcoded instructions, removing, 113-114
- `mkdir` command, 611
- MMU** (Memory Management Unit)
 - definition, 105
 - overview, 121
 - TLB (Translation Lookaside Buffer), 123-124
 - virtual memory and segments, 122-123
- Mode button** (SystemSim), 61
- `modfd2` function, 274-275
- modified element, 562
- modular exponentiation functions, 471
- monitors**
 - color depth, 492
 - pixels, 492
 - speed and frame rate, 492-493
- Monte Carlo library** (`libmc_rand`)
 - distribution of number sequences, transforming
 - Box-Muller method, 484
 - Moro inversion, 484
 - overview, 483-484
 - Polar method, 484
 - `spu_random2.c` example, 486-487
 - transformation functions, 485-486
 - overview, 473-474
 - pseudo-random numbers, generating
 - FSRs (feedback shift registers), 475-476
 - functions, 477-478

- Kirkpatrick–Stoll method (KS), 476
- Mersenne Twister (MT)
 - method, 477
 - overview, 474–475
 - spu_random1.c example, 478–480
- quasi-random numbers, generating
 - functions, 481–483
 - overview, 474–475
 - Sobol generator, 480
- Moro inversion, 484**
- morph element, 569**
- mount command, 22, 611–612**
- mount points, creating, 611**
- move function, 537**
- moveable objects, 549**
- mpm_abs function, 464**
- mpm_add function, 464, 466**
- mpm_add2 function, 464, 466**
- mpm_add3 function, 464, 466**
- mpm_add_partial function, 465–466**
- mpm_cmpeq function, 465**
- mpm_cmpeq2 function, 465**
- mpm_cmpge function, 465**
- mpm_cmpge2 function, 465**
- mpm_cmpgeq function, 468**
- mpm_cmpgt function, 465**
- mpm_cmpgt2 function, 465**
- mpm_cmpgte function, 468**
- mpm_div function, 469–470**
- mpm_div2 function, 469–470**
- mpm_fixed_mod_reduction function, 469–470**
- mpm_gcd function, 469**
- mpm_madd function, 465, 468**
- mpm_mod function, 469–470**
- mpm_mod_exp function, 469, 471**
- mpm_mod_exp2 function, 469, 471**
- mpm_mod_exp3 function, 469, 471**
- mpm_mont_mod_exp function, 469–471**
- mpm_mont_mod_exp2 function, 471**
- mpm_mont_mod_exp3 function, 471**
- mpm_mont_mod_mul function, 469–470**
- mpm_mul function, 465, 468**
- mpm_mult_inv function, 470**
- mpm_mult_inv2 function, 470**
- mpm_mult_inv3 function, 470**
- mpm_mul_inv function, 469**
- mpm_mul_inv2 function, 469**
- mpm_mul_inv3 function, 469**
- mpm_neg function, 464**
- mpm_sizeof function, 465, 468**
- mpm_square function, 465, 468**
- mpm_sub function, 465**
- mpm_sub2 function, 465**
- mpm_swap_endian function, 465, 468**
- mpy instruction, 384**
- mpya instruction, 385**
- mpyh instruction, 384**
- mpyhh instruction, 384**
- mpyhha instruction, 385**
- mpyhha instruction, 385**
- mpyhhu instruction, 384**
- mpyi instruction, 384**
- mpys instruction, 384**
- mpyu instruction, 384**
- mpyui instruction, 384**
- MT (Mersenne Twister) method, 477**
- mtspr instruction, 394**
- mult_matrix4x4 function, 417**
- mult_quat function, 422, 424**
- multiplication**
 - matrix multiplication, 418–419
 - multiplication functions
 - PPU, 170–174
 - SPU, 250–253
 - multiplication instructions (assembly instructions), 384–386
 - multiprocessor matrix multiplication, 434–438

Multiprecision Mathematics Library (libmpm), 463
 division functions, 468-471
 general arithmetic/logic functions, 464-468
 modular exponentiation functions, 471
 overview, 463-464
 public key cryptography, 472-473

multiprocessor matrix multiplication, 434-438

multiprocessor synchronization
 MFC multisource synchronization, 341-342
 multiprocessor DMA ordering, 340-341

multithreading, 116

mutex_init function, 310

mutex_lock function, 310

mutex_trylock function, 310

mutex_unlock function, 310

mutexes, 308-310

N

nand instruction, 387

NCU (Noncacheable Unit), 120

nearbyintd2 function, 263, 283

nearbyintf4 function, 180-181, 263

negatef4 function, 181

next command, 56

next num command, 56

ninja.material file, 547

ninja_res.cfg file, 547

nmsub_matrix_matrix function, 428

nmsub_number_vector function, 428

nmsub_vector_vector function, 428

nodes (Ogre), 549-552

Noncacheable Unit (NCU), 120

nop instruction, 394-395

nor instruction, 387

normal vectors, 524

normalize3 function, 413

normalize3_v function, 413

normalize4 function, 413

normalized vectors, 414

notification channels (signals), 333-335

notification modes (signals), 336-337

numbering code lines, 87

numbers
 direction numbers, 480
 distribution of number sequences, transforming
 Box-Muller method, 484
 Moro inversion, 484
 overview, 483-484
 Polar method, 484
 spu_random2.c example, 486-487
 transformation functions, 485-486

prime numbers, finding, 56-57

pseudo-random numbers, generating
 FSRs (feedback shift registers), 475-476
 functions, 477-478
 Kirkpatrick-Stoll method (KS), 476
 Mersenne Twister (MT) method, 477
 overview, 474-475
 spu_random1.c example, 478-480

quasi-random numbers, generating
 functions, 481-483
 overview, 474-475
 Sobol generator, 480

O

Object-Oriented Graphics Rendering Engine, 529

objects

moveable objects, 549

object files (ELF)
 ELF headers, 586
 HelloWorld.c example, 585-586

- relocation tables, 592-593
- section headers, 587-591
- symbol and string tables, 591-592
- vertex buffer objects, 525-527

Ogre3D

- animation, 555
- building applications, 531-532
- Camera class, 536-537
- downloading, 529
- entities, 549
- FrameListener class, 556-557
- FreeImage, 530
- InputManager class, 554
- installing, 530
- Light class, 552-554
- materials, 544-547
- meshes, 538-541
- OgreXMLConverter, 540
- ogre_basic application, 532-533
- overview, 529
- plug-ins, 533-534
- resource configuration files, 547-548
- Root class, 533-536
- SceneManager class, 536
- SceneNode class, 549-552
- Setup dialog, 531-532
- skeletons, 541-544
- user input, responding to, 554-555
- Viewport class, 538

ogre_basic application, 532-533**OgreXMLConverter, 540****one-dimensional FFT (fast Fourier transform), 452-454****OpenGL**

- Architecture Review Board (ARB), 504
- datatypes, 512-513

Gallium

- building libraries, 507-508
- downloading, 506-507
- overview, 505-506

GLUT (OpenGL Utility Toolkit)

- creating windows with, 508-512
- functions, 508-510
- overview, 508

Khronos Group, 504-505**Mesa**

- building libraries, 507-508
- downloading, 506-507
- overview, 505

normal vectors, 524

overview, 503-504

shapes, 518-520

vertices

color, 521-523

defining, 518-520

vertex buffer objects, 525-527

viewing regions, defining, 513-514

orthographic projections, 514-515

perspective projections, 515-518

/opt/cell/sdk directory (SDK), 29**/opt/cell/systemsim-cell directory (SDK), 30****/opt/cell/toolchain directory (SDK), 30****optional SDK packages, installing, 23, 26-27****Options button (SystemSim), 61****or instruction, 386****orbi instruction, 386****orc instruction, 386****orhi instruction, 386****ori instruction, 386****orthogonal projections, 420-421****orthographic projections, 514-515****ortho_matrix4x4 function, 420****orx instruction, 387, 391**

Outline view (Eclipse), 88, 95

output buffer, 620

OVERLAY statement, 347

overlay1.c file, 348-349

overlays (SPU)

- defining in linker script, 346-347
- loading into memory, 346
- overlay1y.c example, 348-349
- overview, 345-346
- spu_overlay.c example, 348

P

p element, 566-567

packages

- add-on packages
 - installing, 611
 - overview, 609
 - ps3-utils, 610-611
 - table of available packages, 610
- optional SDK packages, installing, 23, 26-27

packaging graphics, 559

packing functions (PPU), 174-177

parameter contexts

- adding to work blocks, 628
- parameter context buffer, 620

pass attributes (materials), 545-547

pattern rules, 50-51

Paul, Brian, 505

permutation functions (PPU), 177-180

perspective projections, 421-422, 515-518

perspective_matrix4x4 function, 420, 422

petaflops, 1

ph element, 567

phony targets, 49

pipeline, PPU

- branch prediction, 110-111
- configuring
 - controlling instruction cache, 112-113
 - improving chances of dual issue, 114-115
 - preventing branch misses, 113
 - removing microcoded instructions, 113-114
- decode phase, 112
- dual-pipeline architecture, 406-408
- instruction buffers and dispatch, 111
- instruction cache
 - controlling, 112
 - overview, 110
- issue phase, 112
- microcode translation, 111-112
- overview, 108-110
- pipeline bubbles, 108

pipeline, SPU

- branch procession and prediction, 236-237
- decode stage, 237
- dual-pipeline architecture, 406-408
- issue stage, 237-238
- overview, 235-236
- prefetch and buffering, 236

pitch function, 537, 550

pixel clock, 492

pixels, 492

placeElement function, 577

placeElementAfter function, 578

placeElementAt function, 577

placeElementBefore function, 578

plain text (PT), 472

PlayStation 3

- installing Fedora Core on
 - burning Fedora Core disk image to DVD, 17-18
 - obtaining kboot bootloader, 18
 - overview, 16
 - step-by-step installation process, 18-20
- sales of, 581

Plugin_BSPSceneManager.so, 534

Plugin_OctreeSceneManager.so, 534

Plugin_ParticleFX.so, 534

plug-ins

- definition, 84
- Ogre plug-ins, 533-534

plugins.cfg file, 533-534

polar coordinates, converting rectangular coordinates to, 272

Polar method, 484

polling, 324-325

polygons element, 566

polylist element, 566

The Potential of the Cell Processor for Scientific Computing, 4-5

pound sign (#), 37

PowerPC Processor Element. See PPE

PowerPC Processor Storage Subsystem (PPSS), 8, 120

PowerPC standard, 104-105

pwd2 function, 268

powf4 function, 189, 268

PPE (PowerPC Processor Element), 2, 7-8, 36, 59, 99

- event handling, 327-328
- overview, 7-8
- PPSS (PowerPC Processor Storage Subsystem), 8
- PPU (PowerPC Processing Unit), 7-8

PPSS (PowerPC Processor Storage Subsystem), 8, 120

PPU (PowerPC Processing Unit), 7-8

- applications, embedding SPE executables into, 133-135
- bit ordering, 101-102
- building applications for
 - build process, 37
 - PPU header files, 36-37
 - PPU source files, 36
 - ppu-as assembler, 40
 - ppu-cpp preprocessor, 37-38
 - ppu-gcc compiler, 38-43
 - ppu-ld linker, 40-42
- datatypes, 100-101
- dual-threaded operation
 - intrinsic functions, 118
 - multithreading example, 116-118
 - thread resources, 115-116
- FPU (Floating-Point Unit), 156-158
- functional units, 105
- intrinsic functions, 103
- libfft PPU functions, 458-460
- libraries, 102
- LSU (Load Store Unit)
 - DCache and memory synchronization, 119-120
 - overview, 118-119
 - PPSS (PowerPC Processor Storage Subsystem), 120
- mailbox communication functions, 331-332
- MMU (Memory Management Unit)
 - overview, 121
 - TLB (Translation Lookaside Buffer), 123-124
 - virtual memory and segments, 122-123
- overview, 99
- pipeline
 - branch prediction, 110-111
 - configuring, 112-115
 - decode phase, 112

- instruction buffers and dispatch, 111
- instruction cache, 110, 112
- issue phase, 112
- microcode translation, 111-112
- overview, 108-110
- pipeline bubbles, 108
- PowerPC standard, 104-105
- PPU access to MFC registers, 321
- PPU-initiated DMA (direct memory access), 302-304
- register files, 106-108
- signaling, 338
- time base register, 103-104
- timing, 103-104
- vector functions
 - addition/subtraction functions, 166-170
 - basic unary functions, 180-183
 - conversion, packing, and splatting functions, 174-177
 - exponent/logarithm functions, 188
 - floating-point analysis functions, 193-194
 - load and store functions, 163, 166
 - logic functions, 183
 - multiplication/division functions, 170-174
 - overview, 162-163
 - permutation and shifting functions, 177-180
 - trigonometric functions, 190-193
 - vector comparison functions with scalar return values, 186-188
 - vector comparison functions with vector return values, 184
- vector processing
 - accessing vectors with unions, 155-156
 - floating-point values, 156-158
 - graphics rounding mode, 156
 - Java mode, 157-158
 - overview, 154
 - vector datatypes, 154
 - vector registers, 158-159
- vectors, 239-241
- ppu_add.c file, 169-170**
- ppu_alf_simple.c file, 622-623**
- ppu_alf_text.c file, 638-640**
- ppu-as assembler, 40**
- ppu-cpp preprocessor, 37-38**
- ppu_denormal.c file, 157**
- ppu_divcomp.c file, 172-173**
- ppu-embedspu command, 129, 601**
- ppu_events.c file, 140-142**
- ppu_fbcheck.c file, 497**
- ppu_fbmulti.c file, 501**
- ppu_fbsingle.c file, 499-501**
- ppu-gcc compiler, 10, 38-43**
- ppu_glut_basic.c file, 511-512**
- PPU-initiated DMA (direct memory access), 302-304**
- ppu-ld linker, 40-42**
- ppu_libfft.c file, 459-460**
- ppu_load.c file, 165-166**
- ppu_massv.c file, 161-162**
- ppu_permtest.c file, 178-179**
- ppu_polar.c file, 192-193**
- ppu_prime.c file, 602-604**
- ppu_rounding.c file, 182-183**
- ppu_shiftadd.c file, 198-200**
- ppu_swizzle.c file, 202-203**
- ppu_threads.c file, 145-147**
- ppu_thread_data function, 144**
- ppu_thread_function function, 144**
- ppu_union.c file, 155**
- prefetch (SPU), 236**
- preprocessors**
 - ppu-cpp, 37-38
 - spu-cpp, 43

prime numbers, finding, 56-57
print command, 55-56
printf() function, 227
private keys, 359, 472
proc command, 666
procedures (Tcl), 666-668
processes, compared to pthreads, 143
profiling commands in SystemSim, 66-68
prof_cpN() function, 68
program handles
 definition, 125-126
 loading into contexts, 138
program headers (ELF), 593-594
Project menu commands, Build Project, 90
projection matrices
 orthogonal projections, 420-421
 perspective projection, 421-422
 projection functions, 419-420
projects. See also applications
 adding/editing source code, 87-89
 building, 90
 configuring, 89-90
 creating, 87
 definition, 86
 orthographic projections, 514-515
 perspective projections, 515-518
ps3-flash-util utility, 611
ps3-utils package, 610-611
ps3-video-mode utility, 611
PS3FB_IOCTL_GETMODE request code (ioctl), 498
PS3FB_IOCTL_OFF request code (ioctl), 498
PS3FB_IOCTL_ON request code (ioctl), 498
PS3FB_IOCTL_SCREENINFO request code (ioctl), 496-497
PS3FB_IOCTL_SETMODE request code (ioctl), 498
PS3s (PCs and PlayStation 3 consoles), 1

pseudo-random numbers, generating
 FSRs (feedback shift registers), 475-476
 functions, 477-478
 Kirkpatrick-Stoll method (KS), 476
 Mersenne Twister (MT) method, 477
 overview, 474-475
 spu_random1.c example, 478-480
PT (plain text), 472
pthread_create function, 143
pthread_exit function, 143
pthread_join function, 143
pthread.h library, 102
pthreads
 compared to processes, 143
 functions, 143-147
 overview, 142-143
public key cryptography, 472-473
public keys, 359, 472
PuTTY, 31

Q

QMC (Quasi-Monte Carlo) methods, 475
quasi-random numbers, generating
 functions, 481-483
 overview, 474-475
 Sobol generator, 480
quaternions, 424
quat_to_rot_matrix4x4 function, 422, 424

R

random numbers, 474
RAs (real addresses), 110
RAW (Read-After-Write) hazard, 112
rhcmt instruction, 394
rdch instruction, 394
Read-After-Write (RAW) hazard, 112
read_lock function, 311
read_trylock function, 311

- read_unlock** function, 311
- readelf** command, 586, 598
- reader/writer** locks, 310-311
- readers**, emitter readers, 73, 76-81
- reading** processor information, 55-56
- real** addresses (RAs), 110
- recipd2** function, 263
- recip4f** function, 181, 263
- recognizing** events
 - event-recognition functions, 323-324
 - interrupt handling, 325-327
 - polling, 324-325
 - waiting, 324
- records**, emitter records, 73
- rectangular** coordinates, converting to polar coordinates, 272
- Red Hat Package Manager (rpm)**, 21, 611
- refactoring**, 88
- reflect_vec3** function, 414-415
- reflect_vec3_v** function, 414-415
- register** files (PPU), 106-108
- registering** events, 136-137
- registers**, SPU user registers, 221-222
- relocation** tables (ELF files), 592-593
- remainderf4** function, 171, 174
- remote** systems
 - connecting to, 30
 - with PuTTY, 31
 - with WinSCP, 31-33
 - running executables on, 91-93
- removeAndDestroyChild** function, 550
- removeChild** function, 550
- removeChildElement**, 578
- removeDocument** function, 574-575
- removeElement** function, 574-575
- removeUpdate** function (cellsdk), 22
- remquof4** function, 171, 174
- rendering** graphics with Ogre3D, 534-536
- renderOneFrame** function, 535
- RenderSystem_GL.so**, 534
- request** codes (ioctl)
 - Linux frame buffer, 495-496
 - PS3 frame buffer, 496-498
- request** lists (DMA)
 - list elements, 297-298
 - list functions, 298-301
 - overview, 296-297
- resource** configuration files (Ogre), 547-548
- responding** to user input in Ogre3D, 554-555
- restoreConfig** function, 535
- return** statement, 667
- revision** element, 561
- Rivest**, Ron, 472
- Roadrunner** supercomputer design, 3
- roll** function, 537, 550
- Root** class, 533-536
- rot** instruction, 397
- rot_matrix4x4** function, 422
- rot_matrix4x4_to_quat** function, 424
- rotate** functions (SPU), 260-263
- rotate** instructions (assembly instructions), 397-399
- rotating** vectors
 - functions, 422
 - quaternions, 424
 - rotating coordinates in 2D and 3D, 423-424
 - spu_rotate.c example, 424-426
- roth** instruction, 397
- rothi** instruction, 397
- rothm** instruction, 398
- rothmi** instruction, 398
- roti** instruction, 397
- rotm** instruction, 398
- rotma** instruction, 398
- rotmah** instruction, 398
- rotmahi** instruction, 398
- rotmai** instruction, 398
- rotmi** instruction, 398

- security (SPU)**
 - application encryption, 361-362
 - building secure applications, 359-360
 - keys, 358-359
 - libspe and secure contexts, 360-361
 - overview, 357-358
 - signatures, 358-359
 - SPU isolation library, 362-365
 - tools, 358
- segments, 122-123**
- selb instruction, 380**
- select functions (SPU), 253-256**
- selecting events, 323**
- selection mask creation instructions (assembly instructions), 380-381**
- sending signals from SPE, 335-336**
- Service GDB button (SystemSim), 60**
- setAspectRatio function, 537**
- setAttenuation function, 552-553**
- setAttribute, 578**
- setCastShadows function, 553**
- setDatabase function, 574**
- setDiffuseColour function, 553**
- setDirection function, 537, 552**
- setElementName, 578**
- setEnabled function, 555**
- setFarClipDistance function, 537**
- setFOVy function, 537**
- setLoop function, 555**
- setNearClipDistance function, 537**
- setOrientation function, 550**
- setparam element, 570-571**
- setPosition function, 537, 550, 552**
- setRenderSystem function, 535**
- setSpecularColour function, 553**
- setPosition function, 555**
- setType function, 552**
- Setup dialog (Ogre), 531-532**
- setXmlns function, 577**
- sf instruction, 382**
- sfh instruction, 382**
- sfhi instruction, 382**
- sfi instruction, 382**
- SFP (SPU Floating-Point Unit), 221**
- SFS (SPU Odd Fixed-Point Unit), 221**
- SFX (SPU Even Fixed-Point Unit), 221**
- sfx instruction, 382**
- sgemm_spu function, 432**
- sgemv_spu function, 432**
- Shamir, Adi, 472**
- shapes, 518-520, 566-567**
- shared libraries, 598-599**
- shell lines (makefiles), 46-47**
- shift functions**
 - PPU, 177-180
 - SPU, 260-263
- shift instructions (assembly instructions), 395-396**
- shifting property (FFT), 449**
- shl instruction, 396**
- shlh instruction, 396**
- shlhi instruction, 396**
- shli instruction, 396**
- shlq instruction, 396**
- shlqbi instruction, 396**
- shlqbii instruction, 396**
- shlqby instruction, 396**
- shlqbybi instruction, 396**
- shlqbyi instruction, 396**
- showConfigDialog function, 535**
- shufb instruction, 377-378, 396**
- shuffle functions (SPU), 253-256**
- shuffle mask creation instructions (assembly instructions), 377-380**
- shuffling instructions (assembly instructions), 377-380**
- shutdownPlugins function, 534**
- si_ilh function, 369**
- si_prefix, 369**

Sieve of Eratosthenes, 56-57**sieve.tcl file, 661, 667-668****signals, 441-442**

- many-to-one communication, 336-337

- notification channels, 333-335

- notification modes, 336-337

- overview, 333

- PPU signaling, 338

- read operations, 333-334

- sending from SPE, 335-336

- SPE synchronization, 338-340

- time-domain signals, 440

signatures, SPU executables, 358-359**signbitd2 function, 274****signbitf4 function, 193-194, 274****SIMD**

- Math library, 241-242

- vector functions for PPU

- addition/subtraction functions, 166-170

- basic unary functions, 180-183

- conversion, packing, and splatting functions, 174-177

- exponent/logarithm functions, 188

- floating-point analysis functions, 193-194

- load and store functions, 163, 166

- logic functions, 183

- multiplication/division functions, 170-174

- overview, 162-163

- permutation and shifting functions, 177-180

- trigonometric functions, 190-193

- vector comparison functions with scalar return values, 186-188

- vector comparison functions with vector return values, 184

- vectorization of data transfer and string manipulation

- libfreevec library, 204-205

- libmtovec library, 204-205

- memcpy function, 206-207

- strcmp function, 207-208

- vectorization of scalar algorithms, 195-197

- loop unrolling, 200

- swizzling, 200-203

- vectorized addition, 195-200

- vectorized addition, 196-197

- vectorized insertion sort

- intervector sort, 209-210

- intravector sort, 210-211

- overview, 208-209

- vecsort.c example, 211-215

simdmath.h library, 102**simemit_set command, 74****simple assembly file example, 370-371****simulated applications**

- building, 64

- compiling, 64

- running, 65

- transferring to simulated Cell, 64-65

Simulator tab (Target Environment Configuration dialog), 93**simulators**

- running executables on, 93-94

- SystemSim

- checkpoints, 68-69

- command window, 61-63

- configuring, 59-60

- console window, 63

- control buttons, 60-61

- emitters, 73-81

- overview, 59

- running, 60-61

- simulated applications, compiling and running, 64-65

- starting, 60-61
- statistics and profiling commands, 66-68
- trigger actions, 70-73
- trigger events, 69-73
- sind2 function, 270**
- sinf4 function, 190, 270**
- single-buffered DMA (direct memory access), 293-294**
- single DMA transfer on SPU, 288-290**
- single-frequency vectors, 444-445**
- sinhd2 function, 270**
- sinhf4 function, 190, 270**
- skeletons (Ogre), 541-544**
- skin element, 568-569**
- slerp_quat function, 422**
- SLIH (second-level interrupt handler), 327**
- SLS (SPU Load/Store Unit), 221**
- SMT (symmetric multithreading), 8**
- SoA, converting AoS (array of structures) to, 200-203, 278-280**
- Sobol generator, 480**
- software cache (SPU)**
 - accessing
 - safe software cache functions, 352-353
 - unsafe software cache functions, 353-354
 - cache statistics, 357
 - configuring, 350-352
 - heapsort, 354-357
 - overview, 349-350
- Software Development Kit (SDK), 10, 21, 59**
- software licensing**
 - GPL (GNU General Public License), 27-28
 - ILAR (International License Agreement for Early Release of Programs), 27
 - LGPL (Lesser GNU General Public License), 28
- solve_linear_system_1 function, 429**
- solve_unit_lower function, 429**
- solve_unit_lower_1 function, 429**
- solve_upper_1 function, 429**
- sorts, vectorized insertion sort**
 - intervector sort, 209-210
 - intravector sort, 210-211
 - overview, 208-209
 - vecsort.c example, 211-215
- source code, editing, 87-89**
- source_data element, 561-562**
- source element, 564-565**
- SPE (Synergistic Processor Element), 2, 127.**
See also SPE Runtime Management Library (libspe)
 - channels, 318-320
 - direct access
 - in libspe, 151-152
 - overview, 149
 - SPU File System (SPUFS), 149-151
 - events, 322-323
 - management process
 - creating contexts, 135-136
 - creating event handlers/registering events, 136-137
 - deallocating data structures, 142
 - embedding SPE executable into PPU application, 133-135
 - examining system configuration, 132-133
 - loading program handle into context, 138
 - overview, 132
 - waiting for events and analyzing stop information, 139-142
 - overview, 8-9
 - sending signals from, 335-336
 - synchronization, 338-340
- spe_context_create function, 135-136, 337**
- spe_context_destroy function, 142, 149**

- spe_context_load() function, 361
- spe_context_run function, 138
- spe_cpu_info_get function, 132-133
- SPE_EVENT_ALL_EVENTS event type, 327
- spe_event_handler_create function, 136-137
- spe_event_handler_destroy function, 142
- SPE_EVENT_IN_MBOX event type, 327
- SPE_EVENT_OUT_INTR_MBOX event type, 327
- SPE_EVENT_SPE_STOPPED event type, 327
- SPE_EVENT_TAG_GROUP event type, 327
- spe_event_wait function, 139
- spe_gang_context_create function, 147
- spe_gang_context_destroy function, 149
- spe_image_close function, 142
- spe_image_open function, 135
- spe_in_mbox_status function, 332
- spe_in_mbox_write function, 332
- spe_ls_area_get function, 151
- spe_mfcio_get function, 303-304
- spe_mfcio_getb function, 303
- spe_mfcio_getf function, 303
- spe_mfcio_put function, 303
- spe_mfcio_putb function, 303
- spe_mfcio_putf function, 303
- spe_mfcio_tag_status_read function, 303-304
- spe_out_intr_mbox_read function, 332
- spe_out_intr_mbox_status function, 332
- spe_out_mbox_read function, 331
- spe_out_mbox_status function, 331
- spe_ps_area_get function, 151
- SPE Runtime Management Library (libspe), 132
 - ALF (Accelerated Library Framework) and, 618-619
 - overview, 125-127
 - pthreads
 - compared to processes, 143
 - functions, 143-144
 - IBM code conventions, 144
 - overview, 142-143
 - ppu_threads.c example, 145-147
- spe_context_create function, 135-136
- spe_context_destroy function, 142
- spe_cpu_info_get function, 132-133
- spe_event_handler_create function, 136-137
- spe_event_handler_destroy function, 142
- spe_event_wait function, 139
- spe_image_close function, 142
- spe_image_open function, 135
- spe_stop_info_read function, 139-140
- spu_basic.c example
 - building from command line, 129
 - building with Cell SDK IDE, 129-131
 - code listing, 127
 - PPU code to launch, 128-129
- spu_program_load function, 138
- spe_signal_write function, 338
- spe_stop_info_read function, 139-140
- spe_stop_info_read() function, 361
- SPE Visualization button (SystemSim), 61
- speed of displays, 492-493
- spinlock, 310
- spinning, 310
- splatting functions (PPU), 174-177
- splat_matrix4x4 function, 418
- SPU (Synergistic Processor Unit), 219
 - addressing modes, 373
 - assembly language, 367
 - building applications for, 43
 - channels
 - definition, 317
 - functions, 318-321
 - PPU access to MFC registers, 321
 - scholar-butler analogy, 317-318
 - SPE channels, 318-320

- communication, 317, 285
- converting AOS to SOA, 278-280
- datatypes
 - scalar datatypes, 223-224
 - vector datatypes, 224
- decremeter, 242-244
- DMA between SPUs, 304-306
- dynamic allocation
 - functions, 233-234
 - heap, 233
 - spu_heapttest.c application, 234
- executable loading, 232-233
- floating-point processing, 224-225
- floating-point status and control register (FPSCR)
 - configuring double-precision rounding mode with, 283-284
 - detecting errors with, 281-283
 - overview, 280
- functional units, 220-221
- initializing, 230-232
- instruction pipeline
 - branch procession and prediction, 236-237
 - decode stage, 237
 - issue stage, 237-238
 - overview, 235-236
 - prefetch and buffering, 236
- instruction set reference, 649-659
- libfft_spu SPU functions, 460-461
- libraries
 - C/C++ standard libraries, 226-227
 - intrinsics, 227
- LS (local store)
 - memory synchronization, 229
 - overview, 227-229
- mailbox communication
 - functions, 328-329
 - mailbox event processing, 330-331
 - mailbox read, 330
 - mailbox write, 329-330
- overlays
 - defining in linker script, 346-347
 - loading into memory, 346
 - overlay1.c example, 348-349
 - overview, 345-346
 - spu_overlay.c example, 348
- overview, 219
- security
 - application encryption, 361-362
 - building secure applications, 359-360
 - keys, 358-359
 - libspe and secure contexts, 360-361
 - overview, 357-358
 - signatures, 358-359
 - SPU isolation library, 362-365
 - tools, 358
- software cache
 - accessing, 352-354
 - cache statistics, 357
 - configuring, 350-352
 - heapsort, 354-357
 - overview, 349-350
- SPU Channel and DMA Unit (SSC), 221
- SPU Control Unit (SCN), 221
- SPU Even Fixed-Point Unit (SFX), 221
- SPU File System (SPUFS), 149-151
- SPU Floating-Point Unit (SFP), 221
- SPU Load/Store Unit (SLS), 221
- SPU Modes button (SystemSim), 61
- SPU Odd Fixed-Point Unit (SFS), 221
- SPU-SPU mailbox communication, 333
- spulets, 232-233
- stack operation, 230-232

- unaligned data, processing, 276–277
- user registers, 221–222
- vector functions, 239–241
 - addition/subtraction functions, 246–250
 - basic unary functions, 263–266
 - compare/count functions, 256–260
 - exponent/logarithm functions, 268–270
 - floating-point analysis functions, 273–276
 - logical functions, 266–267
 - multiplication/division functions, 250–253
 - overview, 244
 - shift/rotate functions, 260–263
 - shuffle/select functions, 253–256
 - trigonometric functions, 270–273
 - vector/scalar functions, 245–246
- spulets, 232–233**
- spu_absd function, 247, 250**
- spu_add function, 246–247**
- spu_addcx function, 248**
- spu_addlarge.c file, 248**
- spu_addreg.s file, 370–371**
- spu_addx function, 240, 246–247**
- spu_alf_text.c file, 640–642**
- spu_and function, 266**
- spu_andc function, 266–267**
- spu-as assembler, 43, 371**
- spu_avg function, 247, 250**
- spu_basic.c application**
 - building from command line, 129
 - building with Cell SDK IDE, 129–131
 - code listing, 127
 - PPU code to launch, 128–129
- spu_bisled function, 324–325**
- spu_blas.c file, 433–434**
- spu_caller.c application, 404**
- spu_caller.s application, 405**
- spu_cashier.c file, 313–315**
- spu_cmpabseq function, 258**
- spu_cmpabsgt function, 258**
- spu_cmpeq function, 256**
- spu_cmpgt function, 256**
- spu_cntb function, 259**
- spu_cntlz function, 259**
- spu_complit.c file, 259–260**
- spu_context_create function, 360**
- spu_convtf function, 265**
- spu_convts function, 265**
- spu_convtu function, 265**
- spu-cpp preprocessor, 43**
- spu_dectest.c file, 243**
- spu_dmabasic.c file, 289–290**
- spu_dmalist.c file, 300–301**
- spu_double.c file, 295–296**
- spu_dsync() function, 229**
- SPU-ELF files**
 - complete PPU executable, 602–604
 - TOE section, 600–601
- spu_eqv function, 266**
- spu_extract function, 245**
- spu_fpscr.c file, 282–283**
- spu_func.c application, 404–405**
- spu_gather function, 259**
- spu-gcc compiler, 10, 43**
- spu-gcc debugger, 371**
- spu-gdb debugger**
 - controlling application execution, 56
 - overview, 54
 - reading processor information, 55–56
 - sample debug sessions, 56–58
 - setting breakpoints/watchpoints, 54–55
- spu_genb function, 247**
- spu_genbx function, 247**
- spu_genc function, 246, 248**
- spu_gencx function, 247–248**

spu_heapstest.c file, 234
spu_idisable function, 324-325
spu_ienable function, 324-325
spu_insert function, 245
spu_interrupt.c file, 326-327
spu_intfract.c file, 275
spu-ld linker, 43
spu_libfft.c file, 461
spu_madd function, 252
spu_mask.s file, 378-379
spu_maskb function, 255
spu_maskh function, 255
spu_maskvec.c file, 255-256
spu_maskw function, 255
spu_matmul.c file, 418-419
spu_mbox_interrupt.c file, 330-331
spu_mffpscr function, 282
spu_mhhadd function, 252
spu_mpm_add.c file, 467-468
spu_msub function, 253
spu_mtfpscr function, 283
spu_mul function, 252
spu_mule function, 251
spu_mulh function, 252
spu_mulo function, 251
spu_mulsr function, 251
spu_nand function, 267
spu_nmadd function, 252
spu_nmsub function, 253
spu_nor function, 267
spu_onedfft.c file, 453
spu_or function, 266
spu_orc function, 267
spu_orx function, 267
spu_overlay.c file, 348
spu_prime.c file, 600-601
spu_program_load function, 138
spu_promote function, 245
spu_random1.c file, 478, 480

spu_random2.c file, 486-487
spu_re function, 263
spu_readch function, 318
spu_readchcnt function, 318, 324
spu_read_decrementer function, 242
spu_read_event_stat function, 324
spu_read_in_mbox function, 329-330
spu_read_signal1 function, 334
spu_read_signal2 function, 334
spu_rect2polar.c file, 272
spu_rl function, 262
spu_rlmask function, 263
spu_rlmaska function, 263
spu_rlmaskqw function, 263
spu_rlqw function, 262
spu_rlqwbyte function, 262
spu_rlqwbytebc function, 262
spu_rndmode.c file, 283-284
spu_rotate.c file, 424-426
spu_roundfloat.c file, 265
spu_rsqrte function, 268
spu_secure_comm.c file, 364
spu_sel function, 254
spu_shright.s application, 399
spu_shuffle function, 253-254
spu_sieve.c file, 56-57
spu_signal_or.c file, 337
spu_sigsync.c file, 339-340
spu_single.c file, 294
spu_sl function, 262
spu_slqw function, 262
spu_slqwbyte function, 262
spu_slqwbytebc function, 262
spu_sndsig function, 335
spu_sndsigb function, 335
spu_sndsigf function, 335
spu_splats function, 245
spu_stacktest.c file, 231-232
spu_stat_event_status function, 324

spu_stat_in_mbox function, 329
spu_stat_out_intr_mbox function, 329
spu_stat_out_mbox function, 329-330
spu_stat_signal1 function, 334
spu_stat_signal2 function, 334
spu_sub function, 247, 250
spu_subtract.s application, 383
spu_subx function, 240, 247, 250
spu_sumb function, 247, 250
spu_sync function, 229
spu_sync_c function, 229
spu_testsv function, 258
spu_twodfft.c file, 456-457
spu_vecscal.c file, 246
spu_writtech function, 318, 323
spu_write_decrementer function, 242
spu_write_event_ack function, 327
spu_write_event_mask function, 323
spu_write_out_intr_mbox function, 329
spu_write_out_mbox function, 329
spu_xor function, 267
SPUFS (SPU File System), 149-151
sqrtd2 function, 268
sqrtf4 function, 189, 268
sqrtf4_fast function, 268
square.tcl file, 661
SSC (SPU Channel and DMA Unit), 221
sscal_spu function, 432
ssyrk64x64 function, 432
stack
 managing, 402-403
 operation, 230-232
 stack buffer, 620
 stack pointer, displaying, 231-232
StageMISCmod macro, 437
standalone mode (SystemSim), 59
starting SystemSim, 60-61
startRendering function, 535

statements
 EXCLUDE_FILE, 347
 for, 665-666
 foreach, 665-666
 if...elseif...else, 662-663
 OVERLAY, 347
static libraries, 597-598
statistics
 cache statistics, 357
 in SystemSim, 66-68
step command, 56
step num command, 56
stop information, analyzing, 139-142
stop instruction, 393
stopd instruction, 393
store functions (PPU), 163, 166
stqa instruction, 374
stqd instruction, 374
stqr instruction, 374-375
stqx instruction, 374
strcmp function, 205-208
STRCMP_LOOP_ALTIVEC_WORD_UNALIGNED macro, 207
STRCMP_QUADWORD_UNALIGNED macro, 208
STRCMP_SINGLE_ALTIVEC_WORD_UNALIGNED macro, 207
STRCMP_UNTIL_SRC1_IS_ALTIVEC_ALIGNED macro, 207
STRCMP_UNTIL_SRC1_WORD_ALIGNED macro, 207
strcpy function, 205
stretching property (FFT), 448-449
strings
 string tables (ELF files), 591-592
 vectorization of string manipulation
 libfreevec library, 204-205
 libmotovec library, 204-205
 memcpy function, 206-207
 strcmp function, 207-208
strip function, 588

- strlen function, 205**
- strncpy function, 205**
- strnlen function, 205**
- strsm_64x64 function, 432**
- strsm_spu function, 432**
- structures, 77**
 - deallocating, 142
 - Elf64_Ehdr, 606
 - Elf64_Shdr, 607
 - GElf_Ehdr, 605
 - mfc_list_element, 298
- Stop button (SystemSim), 60**
- subject element, 561**
- subtraction**
 - matrix subtraction, 642-645
 - subtraction functions
 - PPU, 166-170
 - SPU, 246-250
 - subtraction instructions (assembly instructions), 381-384
- sum_across_float function, 415**
- sum_across_float3 function, 414-415**
- sum_across_float4 function, 414**
- sumb instruction, 382-384**
- swab function, 205**
- swap_matrix_rows function, 427-428**
- swap_vectors function, 427-428**
- swizzling, 200-203, 278-280**
- symbol tables (ELF files), 591-592**
- symmetric multithreading (SMT), 8**
- sync instruction, 394-395**
- syncc instruction, 394-395**
- synchronization**
 - memory synchronization, 119-120, 229
 - multiprocessor synchronization
 - MFC multisource synchronization, 341-342
 - multiprocessor DMA ordering, 340-341
 - SPE synchronization, 338-340
 - Synchronization library (libsync), 308
 - atomic functions, 308
 - completion variables, 313
 - concurrency example (spu_cashier.c), 313-315
 - condition variables, 312
 - mutexes, 308-310
 - reader/writer locks, 310-311
- Synergistic Processor Element. See SPE**
- Synergistic Processor Unit. See SPU**
- system configuration, examining, 132-133**
- SystemSim**
 - checkpoints, 68-69
 - command window, 61-63
 - configuring, 59-60
 - console window, 63
 - control buttons, 60-61
 - emitters
 - emitter readers, 73, 76-81
 - emitter records, 73
 - event configuration, 73-76
 - event types, 73
 - overview, 59
 - running, 60-61
 - simulated applications
 - building, 64
 - compiling, 64
 - running, 65
 - transferring to simulated Cell, 64-65
 - starting, 60-61
 - statistics and profiling commands, 66-68
 - trigger actions
 - associating with trigger events, 71-73
 - definition, 70
 - implementing, 70-71

- trigger events
 - associating trigger actions with, 71-73
 - table of, 69-70

systemsim command, 60

T

tag groups

- advantages of, 290
- checking for DMA completion, 290-291
- ordering transfers in, 291-292

tand2 function, 271

tanf4 function, 191, 271

tar command, 612

Target Environment Configuration dialog, 91-93

targets

- definition, 45
- phony targets, 49

task context buffer, 620

task descriptors

- adding task contexts to, 625-626
- creating, 623
- customizing, 624-625
- fields, 624-625

tasks

- adding work blocks to, 631
- creating, 626-627
- definition, 618
- ending, 631
- launching, 631
- task dependency, 645-646
- task descriptors
 - adding task contexts to, 625-626
 - creating, 623
 - customizing, 624-625
 - fields, 624-625
- task instances, 618

- work blocks, 618
 - adding parameter contexts to, 628
 - adding to tasks, 631
 - buffer storage, 629
 - creating, 627-628
 - data transfer lists, 628-630

Tcl (Tool Command Language), 59

- arrays, 664-665
- if...elseif...else statement, 662-663
- list processing, 663-664
- loop iteration, 665-666
- overview, 661-662
- procedure declarations, 666-668

technique_common element, 564

technique element, 564

technique_hint element, 570

text, transferring and displaying with ALF, 638-642

threads

- PPU dual-threaded operation
 - intrinsic functions, 118
 - multithreading example, 116-118
 - thread resources, 115-116
- pthreads
 - compared to processes, 143
 - functions, 143-144
 - IBM code conventions, 144
 - overview, 142-143
 - ppu_threads.c example, 145-147

time base register (PPU), 103-104

time-domain representation, 440-441

time-domain signals, 440

time vectors, 442

timeline of Cell processor development, 3

timing (PPU), 103-104

title element, 561

TLB (Translation Lookaside Buffer), 123-124

TOE section (SPU-ELF files), 600-601

toe.s file, 600

Tool Command Language. See Tcl

Track All PCs button (SystemSim), 61

transferring

data, 204

text with ALE, 638-642

transforming distribution of number sequences

Box-Muller method, 484

Moro inversion, 484

overview, 483-484

Polar method, 484

spu_random2.c example, 486-487

transformation functions, 485-486

translate function, 550

Translation Lookaside Buffer (TLB), 123-124

transpose_matrix function, 427-428

transpose_matrix4x4 function, 417

triangles element, 566

trifans element, 566

trigger actions (SystemSim)

associating with trigger events, 71-73

definition, 70

implementing, 70-71

trigger events (SystemSim)

associating trigger actions with, 71-73

table of, 69-70

Triggers/Breakpoints button (SystemSim), 61

trigonometric functions

PPU, 190-193

SPU, 270-273

tristrips element, 566

Tukey, John, 439

Tungsten Graphics, 505

two-dimensional FFT (fast Fourier transform), 454-457

two-point Fourier transforms, 450-452

U

unaligned arrays, adding, 198-200

unaligned data, processing, 276-277

unaligned memory, accessing, 195-197

adding unaligned arrays, 198-200

vec_lvsl function, 197

vec_perm function, 197

uname command, 615

unary functions

PPU, 180-183

SPU, 263-266

uninstall function (cellsdk), 22

unions, accessing vectors with, 155-156

unit element, 561-562

unload function, 574

unloadPlugins function, 534

unmount function (cellsdk), 22

unrolling loops, 200

up_axis element, 561

update function (cellsdk), 22

UpdateGUI button (SystemSim), 61

**user input, responding to in
Ogre3D, 554-555**

user registers (SPU), 221-222

/usr/bin directory (SDK), 28-29

V

v event, 322

variables

automatic variables, 50

completion variables, 313

condition variables, 312

in makefiles, 47-48

Tcl variables, 662

VBOs (vertex buffer objects), 525-527

vec_abs function, 180-181

vec_abss function, 180-181

vec_add function, 167, 196

vec_addc function, 167

vec_adds function, 167

vec_all_eq function, 157, 186

vec_all_ge function, 187

vec_all_gt function, 186

vec_all_in function, 188
vec_all_le function, 187
vec_all_lt function, 187
vec_all_nan function, 193
vec_all_ne function, 186
vec_all_nge function, 187
vec_all_ngt function, 186
vec_all_nle function, 187
vec_all_nlt function, 187
vec_all_numeric function, 194
vec_and function, 184
vec_andc function, 184
vec_any_eq function, 186
vec_any_ge function, 187
vec_any_gt function, 186
vec_any_le function, 187
vec_any_lt function, 187
vec_any_nan function, 194
vec_any_ne function, 186
vec_any_nge function, 187
vec_any_ngt function, 187
vec_any_nle function, 188
vec_any_nlt function, 187
vec_any_numeric function, 194
vec_any_out function, 188
vec_avg function, 168
vec_ceil function, 180
vec_ceil/ceilf4 function, 181
vec_cmpb function, 185
vec_cmpeq function, 185
vec_cmpeqge function, 185
vec_cmpeqgt function, 185
vec_cmpeqle function, 185
vec_cmpeqlt function, 185
vec_cmpgt function, 210
vec_ctf function, 174-176
vec_cts function, 174
vec_ctu function, 174
vec_expte function, 189
vec_floor function, 180, 181
vec_floorf4 function, 181
vec_fmaf4 function, 170
vec_ld function, 163-164
vec_lde function, 163-164
vec_ldl function, 163-164
vec_logc function, 189
vec_lvlx function, 163-164
vec_lvxl function, 163-164
vec_lvr function, 163-164
vec_lvrx function, 164
vec_lvsl function, 163-164, 197
vec_lvsr function, 163-164
vec_madd function, 170-171
vec_madds function, 171
vec_max function, 185, 209-210
vec_mergeh function, 176-177
vec_mergel function, 175-177
vec_min function, 186, 209-210
vec_mladd function, 171-172
vec_mradds function, 171-172
vec_mule function, 170, 172
vec_mulo function, 170, 172
vec_nmsub function, 170-171
vec_nor function, 184
vec_or function, 184
vec_pack function, 175-176
vec_packpx function, 175-176
vec_packs function, 175-176
vec_packsu function, 175-176
vec_perm function, 177-178, 197
vec_re function, 181, 183
vec_rl function, 177, 180
vec_round function, 180-181
vec_rsqrte function, 183, 189
vec_rvxl function, 163
vec_sel function, 177, 180
vec_sl function, 177, 180
vec_sld function, 178, 180
vec_sll function, 177, 180
vec_slo function, 177, 180

- `vec_splat` function, 174, 176
- `vec_splat_s3` function, 176
- `vec_splat_s8` function, 175-176
- `vec_splat_s16` function, 175-176
- `vec_splat_s32` function, 175
- `vec_splat_u8` function, 175-176
- `vec_splat_u16` function, 175-176
- `vec_splat_u32` function, 175-176
- `vec_sr` function, 178
- `vec_sra` function, 178, 180
- `vec_srl` function, 178
- `vec_sro` function, 178
- `vec_st` function, 164, 166
- `vec_ste` function, 165-166
- `vec_stl` function, 164
- `vec_stvlx` function, 165
- `vec_stvxl` function, 165-166
- `vec_stvrx` function, 165
- `vec_stvrxl` function, 165-166
- `vec_sub` function, 167
- `vec_subc` function, 167
- `vec_subs` function, 167
- `vec_sum2s` function, 168-169
- `vec_sum4s` function, 168, 170
- `vec_sums` function, 168-169
- `vec_trunc` function, 180, 181
- `vec_truncf4` function, 181
- `vec_unpackh` function, 175, 177
- `vec_unpackl` function, 175-176
- `vec_xor` function, 184
- `vecsort.c` file, 211-215
- vector comparison functions (PPU)**
 - functions with scalar return values, 186-188
 - functions with vector return values, 184
- vector comparison instructions (assembly instructions), 388-389**
- vector datatypes, 224**
- Vector library (libvector)**
 - functions for vector products and lengths, 412-414
 - graphic and miscellaneous functions, 414-416
 - overview, 411-412
- vector register save and restore register (VRSAVE), 108, 158-159**
- vector registers (VRs), 108, 158-159**
- vector status and control register (VSCR), 108, 158**
- Vector/Scalar Unit (VSU), 105**
- vectorization, 195. See also vectors**
 - of data transfer and string manipulation
 - libfreevec library, 204-205
 - libmtovec library, 204-205
 - memcpy function, 206-207
 - strcmp function, 207-208
 - loop unrolling, 200
 - swizzling, 200-203
 - vectorized addition, 195-197
 - adding unaligned arrays, 198-200
 - `vec_lvsl` function, 197
 - `vec_perm` function, 197
 - vectorized insertion sort
 - intervector sort, 209-210
 - intravector sort, 210-211
 - overview, 208-209
 - `vecsort.c` example, 211-215
- vectors. See also vectorization**
 - accessing with unions, 155-156
 - AltiVec library, 160
 - datatypes, 154
 - frequency vectors, 442
 - MASSV library, 161-162, 242
 - normal vectors, 524
 - normalized vectors, 414
 - PPU vector functions
 - addition/subtraction, 166-170
 - basic unary functions, 180-183

- conversion, packing, and splatting functions, 174-177
- exponent/logarithm functions, 188
- floating-point analysis functions, 193-194
- load and store functions, 163, 166
- logic functions, 183
- multiplication/division functions, 170-174
- overview, 162-163
- permutation and shifting functions, 177-180
- trigonometric functions, 190-193
- vector comparison functions with scalar return values, 186-188
- vector comparison functions with vector return values, 184
- PPU vector processing
 - accessing vectors with unions, 155-156
 - floating-point values, 156-158
 - graphics rounding mode, 156
 - Java mode, 157-158
 - overview, 154
 - vector datatypes, 154
 - vector registers, 158-159
- rotating
 - functions, 422
 - quaternions, 424
 - rotating coordinates in 2D and 3D, 423-424
 - spu_rotate.c example, 424-426
- SIMD Math library, 160
- single-frequency vectors, 444-445
- SPU vector functions
 - addition/subtraction functions, 246-250
 - basic unary functions, 263-266
 - compare/count functions, 256-260
 - exponent/logarithm functions, 268-270
 - floating-point analysis functions, 273-276
 - logical functions, 266-267
 - multiplication/division functions, 250-253
 - overview, 244
 - shift/rotate functions, 260-263
 - shuffle/select functions, 253-256
 - trigonometric functions, 270-273
 - vector/scalar functions, 245-246
- SPU versus PPU, 239-241
- time vectors, 442
- vector datatypes, 224
- Vector library (libvector)
 - functions for vector products and lengths, 412-414
 - graphic and miscellaneous functions, 414-416
 - overview, 411-412
- VRs (vector registers), 158-159
- verify function (cellsdk), 22**
- vertex-oriented graphics, 518**
- vertical refresh rate, 493**
- vertical sync frequency, 493**
- vertices**
 - color, 521-523
 - defining, 518-520
 - vertex buffer objects, 525-527
- vertices element, 565**
- viewing ELF executable files, 593**
- viewing regions, defining, 513-514**
 - orthographic projections, 514-515
 - perspective projections, 515-518
- Viewport class, 538**
- virtual memory, 122-123**
- VR Save/Restore Register (VRSARE), 158-159**
- VRs (vector registers), 108, 158-159**
- VRSARE (vector register save and restore register), 108, 158-159**
- vsacos function, 191, 271**
- vsacosh function, 191, 271**
- vsasin function, 191, 270**

vsasinh function, 191, 270
 vsatan2 function, 191, 271
 vsatanh function, 191, 271
 vsbrt function, 161, 189, 268
 vscol function, 191, 270
 vscolsh function, 191, 270
 vscolsin function, 191, 271
 VSCR (vector status and control register),
 108, 158
 vsdiv function, 171, 253
 vsexp function, 189, 268
 vsxpm1 function, 189, 268
 vslog function, 190, 269
 vslog10 function, 190, 242, 269
 vslog1p function, 190, 269
 vspow function, 189, 268
 vsqdrf function, 189, 268
 vsrbrt function, 189, 269
 vsrbrf function, 189, 269
 vsrsqrt function, 189, 269
 vssin function, 162, 190, 270
 vssincos function, 191, 271
 vssinh function, 190, 270
 vssqrt function, 189, 268
 vstan function, 191, 271
 vstanh function, 191, 271
 VSU (Vector/Scalar Unit), 105

W

wait_for_completion function, 313
 waiting for events, 139-142, 324
 watch command, 54-55
 watchpoints, 54-55
 WAW (Write-After-Write) hazard, 112
 Welschenbach, Michael, 463
 Wessel, Caspar, 423
 where command, 55-56
 windows, creating with GLUT (OpenGL Utility
 Toolkit), 508, 510-512
 WinSCP, 31-33

work blocks, 618
 adding parameter contexts to, 628
 adding to tasks, 631
 buffer storage, 629
 creating, 627-628
 data transfer lists, 628-630
wrch instruction, 394
Write Great Code, 5
 Write-After-Write (WAW) hazard, 112
 write_lock function, 311
 write_trylock function, 311
 write_unlock function, 311
writing assembly-coded functions
 example, 404-405
 function declaration, 402
 stack management, 402-403

X-Y-Z

x-form (indexed register indirect
 addressing), 373
 x86 systems, 29-30
 XDR DRAM (eXtreme Data Rate Dynamic
 Random Access Memory), 6
 XER (fixed-point exception register), 107
 xform_vec3 function, 415-416
 xform_vec3_v function, 415-416
 xform_vec4 function, 415-416
 xform_vec4_v function, 415-416
 xor instruction, 387
 xorbi instruction, 387
 xorhi instruction, 387
 xori instruction, 387
 xsbh instruction, 400
 xshw instruction, 400
 xswd instruction, 400-401
 yaw function, 537, 550
 YUM (Yellow dog Updater, Modified), 21