

Software Development Kit for Multicore Acceleration  
Version 3.1



# Programming Tutorial

*Version 3.1*



Software Development Kit for Multicore Acceleration  
Version 3.1



# Programming Tutorial

*Version 3.1*

**Note**

Before using this information and the product it supports, read the information in "Notices" on page 159.

**Edition notice**

This edition applies to the version 3.1, release 0 of the IBM Software Development Kit for Multicore Acceleration (Program number 5724-S84) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation 2005, 2008

---

## Preface

### About this book

This tutorial is written for programmers who are interested in developing applications or libraries for the Cell Broadband Engine™ (Cell/B.E.). It is not intended for programmers who want to develop device drivers, compilers, or operating systems for the Cell Broadband Engine.

The descriptions and examples in this tutorial are from the *Software Development Kit for Multicore Acceleration*, Version 3.0, and Version 3.1. The examples are chosen to highlight the general principals required for Cell Broadband Engine programming, so that an experienced programmer can apply this knowledge to other environments.

### Who should read this book

The document is intended for system and application programmers who wish to develop Cell Broadband Engine applications.

### Prerequisites

It is assumed that you are an experienced C/C++ programmer and are familiar with the basic concepts of single-instruction, multiple-data (SIMD) vector instruction sets, such as the PowerPC® Architecture™ Vector/SIMD Multimedia Extensions, Intel® MMX™, SSE, 3DNOW!, or x86-64 instruction sets.

It is also assumed that you have the Software Development Kit (SDK) for Multicore Acceleration, which includes a Cell/B.E. specific, 64-bit PowerPC Linux® operating system, SDK code examples, and the IBM® Full System Simulator for Cell/B.E.

### Related documentation

The following is a list of reference and supporting materials for the Cell Broadband Engine. Additional documentation is available in the IBM Systems Information Center. The information center provides you with a single, searchable site where you can access product documentation for IBM systems hardware, operating systems, and server software. Through a consistent framework, you can efficiently find information and personalize your access. The IBM Systems Information Center is at <http://publib.boulder.ibm.com/infocenter/systems>.

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine, Architecture*
- *Cell Broadband Engine Linux Reference Implementation, Application Binary Interface Specification*
- *Cell Broadband Engine Programming Handbook including the IBM® PowerXCell™ 8i processor*
- *Cell Broadband Engine Registers*
- *Accelerated Library Framework Programmer's Guide and API Reference*
- *Data Communication and Synchronization Programmer's Guide and API Reference*

- *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*
- *PowerPC Operating Environment Architecture, Book III*
- *PowerPC User Instruction Set Architecture, Book I*
- *PowerPC Virtual Environment Architecture, Book II*
- *SIMD Math Library Specification for Cell Broadband Engine*
- *Software Development Kit Programmer's Guide*
- *SPE Runtime Management Library (Version 2)*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*
- *Synergistic Processor Unit, Instruction Set Architecture*

# Contents

<b>Preface</b> . . . . .	<b>iii</b>	Promoting scalar data types to vector data types	72
<b>Figures</b> . . . . .	<b>vii</b>	Differences between PPE and SPE SIMD support	72
<b>Tables</b> . . . . .	<b>ix</b>	Compiler directives . . . . .	75
<b>Chapter 1. Overview of the Cell Broadband Engine</b> . . . . .	<b>1</b>	MFC commands . . . . .	76
Introduction . . . . .	1	DMA-command tag groups . . . . .	79
Background and motivations . . . . .	1	Synchronizing DMA transfers . . . . .	80
Scaling the three performance-limiting walls . . . . .	3	MFC input and output macros . . . . .	80
Architecture overview . . . . .	4	Coding methods and examples . . . . .	83
The PowerPC Processor Element . . . . .	6	DMA transfers . . . . .	83
Synergistic Processor Elements . . . . .	7	Simple SPE-initiated DMA transfers example . . . . .	85
Programming Overview . . . . .	9	Transferring large buffer from main memory to SPE LS examples . . . . .	85
Byte ordering and bit numbering . . . . .	9	DMA-list transfers . . . . .	90
SIMD vectorization . . . . .	10	Transferring data from main memory to SPE LS using DMA List . . . . .	92
SIMD C-language intrinsics . . . . .	11	Destructive DMA list example . . . . .	94
Threads and tasks . . . . .	12	Vectorizing a loop . . . . .	94
The runtime environment . . . . .	13	Reducing the impact of branches . . . . .	95
Application partitioning . . . . .	13	Porting SIMD code from the PPE to the SPEs . . . . .	98
The software development kit . . . . .	16	Code-mapping considerations . . . . .	99
<b>Chapter 2. The PPE and the programming process</b> . . . . .	<b>19</b>	Simple macro translation . . . . .	100
PPE registers . . . . .	19	Example 1: Euler particle-system simulation . . . . .	102
PPE instruction sets . . . . .	21	Performance analysis . . . . .	112
PowerPC instructions . . . . .	22	Performance issues . . . . .	112
Vector/SIMD Multimedia Extension instructions . . . . .	24	Example 1: Tuning SPE performance with static and dynamic timing analysis . . . . .	112
C/C++ language extensions (intrinsics) . . . . .	25	General SPE programming tips . . . . .	121
Programming with Vector/SIMD Multimedia Extension intrinsics . . . . .	33	<b>Chapter 4. Programming models</b> . . . . .	<b>123</b>
The PPE and the SPEs . . . . .	35	Function-Offload Model . . . . .	123
Storage Domains . . . . .	35	Remote procedure call . . . . .	124
Issuing DMA commands from the PPE . . . . .	37	Device-Extension Model . . . . .	124
Creating threads for the SPEs . . . . .	38	Computation-Acceleration Model . . . . .	125
Communication between the PPE and SPEs . . . . .	40	Streaming model . . . . .	125
Developing code for the Cell Broadband Engine . . . . .	41	Shared-Memory Multiprocessor Model . . . . .	125
Producing a simple multi-threaded CBE program . . . . .	42	Asymmetric-Thread Runtime Model . . . . .	126
Running the program in the simulator . . . . .	44	User-mode thread model . . . . .	126
Debugging programs . . . . .	48	Cell application frameworks . . . . .	126
<b>Chapter 3. Programming the SPEs</b> . . . . .	<b>49</b>	SPE overlays . . . . .	127
SPE configuration . . . . .	49	<b>Chapter 5. The simulator</b> . . . . .	<b>129</b>
Synergistic Processor Unit . . . . .	50	Simulator basics . . . . .	130
Memory flow controller . . . . .	54	Operating-system modes . . . . .	130
Channels . . . . .	55	Interacting with the simulator . . . . .	130
SPU instruction set . . . . .	60	Command-line interface . . . . .	131
Data layout in registers . . . . .	60	Graphical User Interface . . . . .	132
Instruction types . . . . .	62	The simulation panel . . . . .	133
SPU C/C++ language extensions (intrinsics) . . . . .	64	GUI buttons . . . . .	141
Assembly language versus intrinsics comparison: an example . . . . .	65	Performance monitoring . . . . .	146
Intrinsic classes . . . . .	66	Displaying performance statistics . . . . .	147
		SPE performance profile checkpoints . . . . .	150
		Example program: tpa1 . . . . .	152
		Emitters . . . . .	153
		SPU performance and semantics . . . . .	155

**Notices . . . . . 159**  
Edition notices . . . . . 161

**Trademarks . . . . . 163**

**Glossary . . . . . 165**

**Index . . . . . 173**



---

## Figures

1. Overview of Cell Broadband Engine architecture . . . . .	5	23. SIMD floating-point Add instruction function	63
2. PowerPC Processor Element (PPE) block diagram . . . . .	6	24. Array-of-structures data organization for one triangle . . . . .	63
3. Synergistic Processor Element (SPE) block diagram . . . . .	8	25. Structure-of-arrays data organization for four triangles . . . . .	64
4. Big-endian byte and bit ordering . . . . .	10	26. DMA transfers using a double-buffering method . . . . .	86
5. Four concurrent Add operations. . . . .	11	27. Example of the Function-Offload (or RPC) Model . . . . .	124
6. Byte-shuffle operation . . . . .	11	28. Simulation stack . . . . .	129
7. Application partitioning model . . . . .	14	29. Simulator structures and screens . . . . .	131
8. PPE-centric multistage pipeline model and parallel stages model . . . . .	15	30. Main Graphical User Interface for the simulator . . . . .	133
9. PPE-centric services model . . . . .	15	31. Project and processor folders . . . . .	134
10. PPE user-register set . . . . .	20	32. PPE General-Purpose Registers window	135
11. Concurrent execution of integer, floating-point, and vector units . . . . .	24	33. PPE Floating-Point Registers window	135
12. Running the Vector/SIMD Multimedia Extension sample program . . . . .	34	34. PPE Core window . . . . .	136
13. Storage domains defined in the Cell Broadband Engine . . . . .	36	35. SPE MFC window . . . . .	137
14. Sample project directory structure and makefiles . . . . .	42	36. SPE MFC Address Translation window	138
15. Windows visible after starting the simulator GUI . . . . .	45	37. SPE Channels window . . . . .	139
16. Console window on completion of Linux boot	46	38. SPE statistics. . . . .	140
17. Loading the program into the simulation environment . . . . .	47	39. Debug Controls window. . . . .	143
18. Running the sample program . . . . .	48	40. SPE Visualization window . . . . .	144
19. SPE architectural block diagram. . . . .	50	41. Track All PCs window . . . . .	145
20. SPE user-register set . . . . .	51	42. SPU Modes window . . . . .	146
21. Big-endian ordering supported by the SPE	61	43. tpa1 statistics for SPE 0 . . . . .	149
22. Register layout of data types and preferred (scalar) slot. . . . .	61	44. tpa1 statistics for SPE 2 . . . . .	150
		45. Profile checkpoint output for SPE 2 . . . . .	152
		46. Emitters . . . . .	154
		47. Emitter architecture . . . . .	154



---

## Tables

1. PPE and SPE intrinsic classes . . . . .	12	21. PPE and SPE Architectural Comparison	72
2. Definition of threads and tasks . . . . .	12	22. PPE versus SPU Vector Data Types. . . . .	73
3. PPE-specific scalar intrinsics . . . . .	26	23. Single-Token Vector Keyword Data Types	75
4. Vector/SIMD Multimedia Extension data types	29	24. MFC DMA Command . . . . .	76
5. Vector/SIMD Multimedia Extension specific and generic intrinsics . . . . .	29	25. MFC Command Suffixes . . . . .	78
6. Vector/SIMD Multimedia Extension predicate intrinsics . . . . .	32	26. MFC Synchronization Commands . . . . .	79
7. MFC command-parameter registers for PPE-initiated DMA transfers . . . . .	37	27. MFC Atomic Commands . . . . .	79
8. Mailbox channels and MMIO registers . . . . .	40	28. MFC Input and Output Macros . . . . .	80
9. Signal notification channels and MMIO registers. . . . .	41	29. MFC Tag Manager Functions. . . . .	83
10. LS-Access Arbitration Priority and Transfer Size . . . . .	53	30. Interrupt Safe Critical Section Functions	83
11. SPU Instruction Latency and Pipeline, by Instruction Class . . . . .	53	31. Time step description . . . . .	87
12. SPE Channels . . . . .	55	32. Time step description . . . . .	88
13. SPE Channel Instructions . . . . .	57	33. Time step description . . . . .	89
14. Vector Data Types . . . . .	61	34. Time step description . . . . .	90
15. SPU Instruction Types . . . . .	62	35. Branch-Hint Instructions . . . . .	97
16. Specific intrinsics not available as generic intrinsics . . . . .	66	36. Vector/SIMD Multimedia Extension Single-Token Data Types . . . . .	100
17. Specific Casting Intrinsics . . . . .	67	37. SPU Intrinsics with One-to-One Vector/SIMD Multimedia Extension Mapping . . . . .	101
18. Generic SPU Intrinsics . . . . .	69	38. Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping . . . . .	101
19. Composite SPU intrinsics . . . . .	71	39. Important Commands for the IBM Full System Simulator for the Cell Broadband Engine . . . . .	132
20. Intrinsics for Changing Scalar and Vector Data Types . . . . .	72	40. Simulator Performance Statistics for the SPU	155



---

# Chapter 1. Overview of the Cell Broadband Engine

---

## Introduction

The first generation Cell Broadband Engine is the first incarnation of a new family of microprocessors conforming to the *Cell Broadband Processor Architecture* (CBEA). The CBEA is a new architecture that extends the 64-bit PowerPC Architecture.

The CBEA and the Cell Broadband Engine are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

## Background and motivations

Although the Cell Broadband Engine is initially intended for application in game consoles and media-rich consumer-electronics devices such as high-definition televisions, the architecture and the Cell Broadband Engine implementation have been designed to enable fundamental advances in processor performance. A much broader use of the architecture is envisioned.

The Cell Broadband Engine is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. In this respect, it extends current trends in PC and server processors. The most distinguishing feature of the Cell Broadband Engine is that, although all processors share main storage (the effective-address space that includes main memory), their function is specialized into two types:

- the PowerPC Processor Element (PPE),
- the Synergistic Processor Element (SPE).

The Cell Broadband Engine has:

- one PPE,
- eight SPEs.

*The PPE* (the first type of processor element) is a 64-bit PowerPC Architecture core. It is fully compliant with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications.

*The SPE* (the second type of processor element) is optimized for running compute-intensive applications, and it is not optimized for running an operating system. The SPEs are independent processors, each running its own individual application programs. Each SPE has full access to coherent shared memory, including the memory-mapped I/O space.

The designation *synergistic* for this processor was chosen carefully because there is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level control thread of an application. The PPE depends on the SPEs to provide the bulk of the application performance.

The SPEs are designed to be programmed in high-level languages and support a rich instruction set that includes extensive single-instruction, multiple-data (SIMD) functionality. However, just like conventional processors with SIMD extensions, use of SIMD data types is preferred, not mandatory. For programming convenience, the PPE also supports the PowerPC Architecture Vector/SIMD Multimedia Extension.

To an application programmer, the Cell Broadband Engine looks like a 9-way coherent multiprocessor. The PPE is more adept at control-intensive tasks and quicker at task switching. The SPEs are more adept at compute-intensive tasks and slower at task switching. However, either processor is capable of both types of functions. This specialization has allowed increased efficiency in the implementation of both the PPE and especially the SPEs. It is a significant factor in the approximate order-of-magnitude improvement in peak computational performance and area-and-power efficiency that the Cell Broadband Engine achieves over conventional PC processors.

A significant difference between the PPE and SPEs is how they access memory:

- The PPE accesses main storage (the effective-address space that includes main memory) with load and store instructions that go between a private register file and main storage (which may be cached).
- The SPEs access main storage with direct memory access (DMA) commands that go between main storage and a private local memory used to store both instructions and data. SPE instruction-fetches and load and store instructions access this private local store, rather than shared main storage. This 3-level organization of storage (register file, local store, main storage), with asynchronous DMA transfers between local store and main storage, is a radical break with conventional architecture and programming models, because it explicitly parallelizes computation and the transfers of data and instructions.

The reason for this radical change is that memory latency, measured in processor cycles, has gone up several hundredfold in the last 20 years. The result is that application performance is, in most cases, limited by memory latency rather than by peak compute capability or peak bandwidth. When a sequential program on a conventional architecture performs a load instruction that misses in the caches, program execution now comes to a halt for several hundred cycles. Compared to this penalty, the few cycles it takes to set up a DMA transfer for an SPE is quite small. Conventional processors, even with deep and costly speculation, manage to get, at best, a handful of independent memory accesses in flight. The result can be compared to a bucket brigade in which a hundred people are required to cover the distance to the water needed to put the fire out, but only a few buckets are available. In contrast, the explicit DMA model allows each SPE to have many concurrent memory accesses in flight, without the need for speculation.

The most productive SPE memory-access model appears to be the one in which a list (such as a scatter-gather list) of DMA transfers is constructed in an SPE's local store, so that the SPE's DMA controller can process the list asynchronously while the SPE operates on previously transferred data. In several cases, this new approach to accessing memory has led to application performance exceeding that of conventional processors by almost two orders of magnitude, significantly more than one would expect from the peak performance ratio (about 10x) between the Cell Broadband Engine and conventional PC processors.

It is also possible to write compilers that manage an SPE's local Store as a very large second-level register file or to automatically bring in code when needed and present a conventional symmetric multiprocessing (SMP) model. Although such a compiler exists, at least in prototype form, it does not today result in the most optimal application performance. Hence, this tutorial focuses on approaches to programming the Cell Broadband Engine that expose the local store and the asynchronous DMA-transfer commands.

## Scaling the three performance-limiting walls

The Cell Broadband Engine overcomes three important limiters of contemporary microprocessor performance: power use, memory use, and processor frequency.

### Scaling the power-limitation wall

Increasingly, microprocessor performance is limited by achievable power dissipation rather than by the number of available integrated-circuit resources (transistors and wires).

Therefore, the only way to significantly increase the performance of microprocessors is to improve power efficiency at about the same rate as the performance increase.

One way to increase power efficiency is to differentiate between:

- processors optimized to run an operating system and control-intensive code, and
- processors optimized to run compute-intensive applications.

The Cell Broadband Engine does this by providing a general-purpose PPE to run the operating system and other control-plane code, and eight SPEs specialized for computing data-rich (*data-plane*) applications.

### Scaling the memory-limitation wall

On multi-gigahertz symmetric multiprocessors (even those with integrated memory controllers) latency to DRAM memory is currently approaching 1,000 cycles.

As a result, program performance is dominated by the activity of moving data between main storage (the effective-address space that includes main memory) and the processor. Increasingly, compilers and even application writers must manage this movement of data explicitly, even though the hardware cache mechanisms are supposed to relieve them of this task.

The Cell Broadband Engine's SPEs use *two* mechanisms to deal with long main-memory latencies:

- a 3-level memory structure (main storage, local stores in each SPE, and large register files in each SPE),
- asynchronous DMA transfers between main storage and local stores.

These features allow programmers to schedule simultaneous data and code transfers to cover long latencies effectively. Because of this organization, the Cell Broadband Engine can usefully support 128 simultaneous transfers between the eight SPE local stores and main storage. This surpasses the number of simultaneous transfers on conventional processors by a factor of almost twenty.

### Scaling the frequency-limitation wall

Conventional processors require increasingly deeper instruction pipelines to achieve higher operating frequencies. This technique has reached a point of diminishing returns – and even negative returns if power is taken into account.

By specializing the PPE and the SPEs for control and compute-intensive tasks, respectively, the Cell Broadband Engine Architecture, on which the Cell Broadband Engine is based, allows both the PPE and the SPEs to be designed for high frequency without excessive overhead. The PPE achieves efficiency primarily by executing two threads simultaneously rather than by optimizing single-thread performance.

Each SPE achieves efficiency by using a large register file, which supports many simultaneous in-process instructions without the overhead of register-renaming or out-of-order processing. Each SPE also achieves efficiency by using asynchronous DMA transfers, which support many concurrent memory operations without the overhead of speculation.

### **How the Cell Broadband Engine overcomes performance limitations**

By optimizing control-plane and data-plane processors *individually*, the Cell Broadband Engine alleviates the problems posed by the power, memory, and frequency limitations.

The net result is a processor that, at the power budget of a conventional PC processor, can provide approximately ten-fold the peak performance of a conventional processor. Of course, actual application performance varies. Some applications may benefit little from the SPEs, whereas others show a performance increase well in excess of ten-fold. In general, compute-intensive applications that use 32-bit or smaller data formats (such as single-precision floating-point and integer) are excellent candidates for the Cell Broadband Engine.

The remainder of this chapter describes the Cell Broadband Engine hardware, some basic programming conventions, a typical software-development sequence, and the major support tools available in the software development kit (SDK).

- Programming the PPE is described in Chapter 2, “The PPE and the programming process,” on page 19.
- Programming the SPEs is described in Chapter 3, “Programming the SPEs,” on page 49.
- Programming models are described in Chapter 4, “Programming models,” on page 123.
- The *IBM Full System Simulator* for the Cell Broadband Engine is described in Chapter 5, “The simulator,” on page 129.
- A glossary is provided in “Glossary” on page 165.

---

## **Architecture overview**

The Cell Broadband Engine consists of *nine* processors on a single chip, all connected to each other and to external devices by a high-bandwidth, memory-coherent bus.

Figure 1 on page 5 shows a block diagram of the Cell Broadband Engine. The main blocks include the:

- *PowerPC Processor Element (PPE)*. The PPE is the main processor. It contains a 64-bit PowerPC Architecture reduced instruction set computer (RISC) core with a traditional virtual-memory subsystem. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It can run legacy PowerPC Architecture software and performs well executing system-control code. It supports both the PowerPC instruction set and the Vector/SIMD Multimedia Extension instruction set.
- *Synergistic Processor Elements (SPEs)*. The eight SPEs are SIMD processors optimized for data-rich operations allocated to them by the PPE. Each of these identical elements contains a RISC core, 256-KB, software-controlled local store for instructions and data, and a large (128-bit, 128-entry) unified register file. The SPEs support a special SIMD instruction set, and they rely on asynchronous



DMA transfers to move data and instructions between main storage (the effective-address space that includes main memory) and their local stores. SPE DMA transfers access main storage using PowerPC effective addresses. As on the PPE, address translation is governed by PowerPC Architecture segment and page tables. The SPEs are not intended to run an operating system.

- *Element Interconnect Bus (EIB)*. The PPE and SPEs communicate coherently with each other and with main storage and I/O through the EIB. The EIB is a 4-ring structure (two clockwise and two counterclockwise) for data, and a tree structure for commands. The EIB's internal bandwidth is 96 bytes per cycle, and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs.

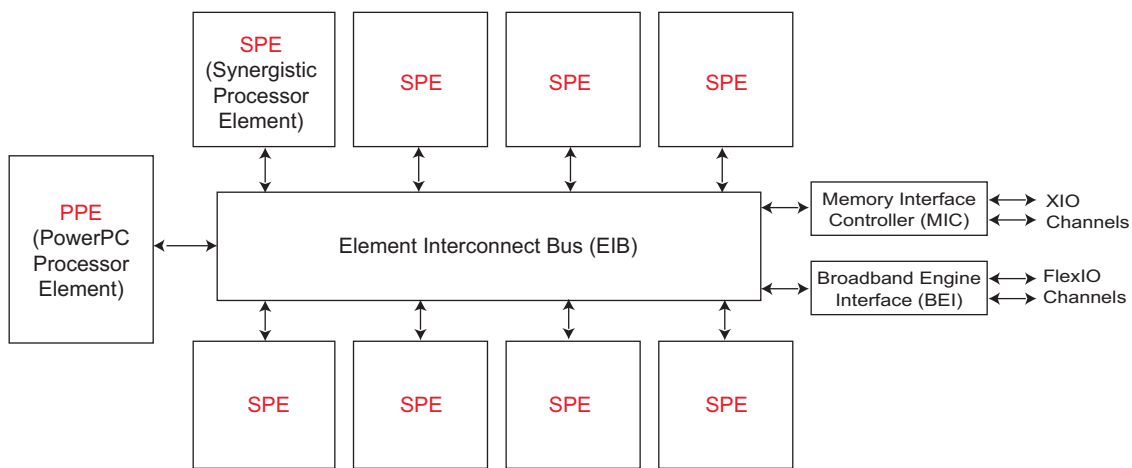


Figure 1. Overview of Cell Broadband Engine architecture

The memory-coherent EIB has *two* external interfaces, as shown in Figure 1:

- The *Memory Interface Controller (MIC)* provides the interface between the EIB and main storage. It supports two Rambus Extreme Data Rate (XDR) I/O (XIO) memory channels and memory accesses on each channel of 1-8, 16, 32, 64, or 128 bytes.
- The *Cell Broadband Engine Interface (BEI)* manages data transfers between the EIB and I/O devices. It provides address translation, command processing, an internal interrupt controller, and bus interfacing. It supports two Rambus FlexIO external I/O channels. One channel supports only non-coherent I/O devices. The other channel can be configured to support either non-coherent transfers or coherent transfers that extend the logical EIB to another compatible external device, such as another Cell Broadband Engine.

The Cell Broadband Engine supports concurrent real-time and non-real-time operating systems and resource management. Software development in the C/C++ language is supported by a rich set of language extensions that define C/C++ data types for SIMD operations and map C/C++ intrinsics (commands, in the form of function calls) to one or more assembly instructions.

These language extensions give C/C++ programmers much greater control over code performance, without the need for assembly-language programming. Software development is further supported by:

- a complete Linux-based SDK,
- a full-system simulator, and
- a rich set of application libraries, performance tools and debug tools.

## The PowerPC Processor Element

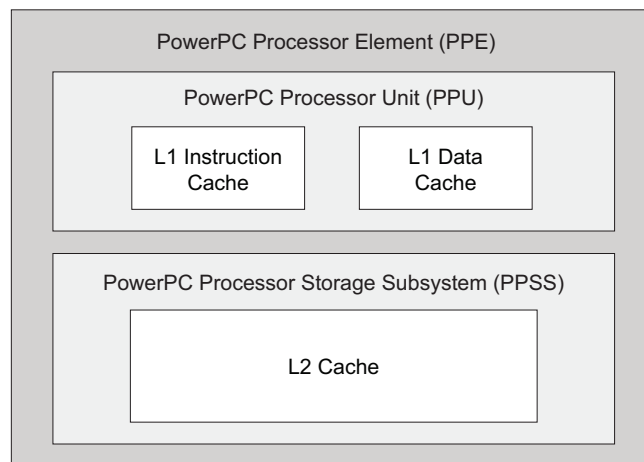
The *PowerPC Processor Element (PPE)* is a general-purpose, dual-threaded, 64-bit RISC processor that conforms to the PowerPC Architecture, version 2.02, with the Vector/SIMD Multimedia Extension.

Programs written for the PowerPC 970 processor, for example, should run on the Cell Broadband Engine without modification.

As shown in Figure 2, the PPE consists of *two* main units:

- The Power Processor Unit (PPU).
- The Power Processor Storage Subsystem (PPSS).

The PPE is responsible for overall control of the system. It runs the operating systems for all applications running on the Cell Broadband Engine.



---

Figure 2. PowerPC Processor Element (PPE) block diagram

The PPU deals with instruction control and execution. It includes:

- the full set of 64-bit PowerPC registers,
- 32 128-bit vector registers,
- a 32-KB level 1 (L1) instruction cache,
- a 32-KB level 1 (L1) data cache,
- an instruction-control unit,
- a load and store unit,
- a fixed-point integer unit,

- a floating-point unit,
- a vector unit,
- a branch unit,
- a virtual-memory management unit.

The PPU supports two simultaneous threads of execution and can be viewed as a 2-way multiprocessor with shared dataflow. This appears to software as two independent processing units. The state for each thread is duplicated, including all architected and special-purpose registers except those that deal with system-level resources, such as logical partitions, memory, and thread-control. Most non-architected resources, such as caches and queues, are shared by both threads, except in cases where the resource is small or offers a critical performance improvement to multithreaded applications.

The PPSS handles memory requests from the PPE and external requests to the PPE from other processors or I/O devices. It includes:

- a unified 512-KB level 2 (L2) instruction and data cache,
- various queues,
- a bus interface unit that handles bus arbitration and pacing on the EIB.

Memory is seen as a linear array of bytes indexed from 0 to  $2^{64} - 1$ . Each byte is identified by its index, called an *address*, and each byte contains a value. One storage access occurs at a time, and all accesses appear to occur in program order.

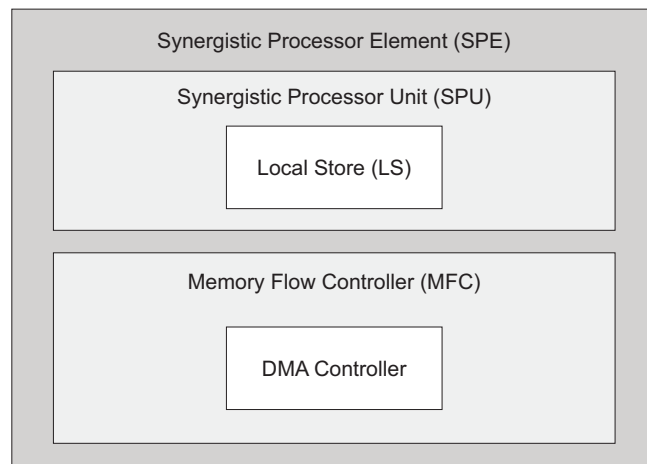
The L2 cache and the address-translation caches use replacement-management tables that allow software to control use of the caches. This software control over cache resources is especially useful for real-time programming.

## Synergistic Processor Elements

Each of the eight *Synergistic Processor Elements* (SPEs) is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD applications.

As shown in Figure 3 on page 8, each SPE consists of *two* main units:

- The Synergistic Processor Unit (SPU).
- The Memory Flow Controller (MFC).




---

Figure 3. Synergistic Processor Element (SPE) block diagram

The SPU deals with instruction control and execution. It includes a single register file with 128 registers (each one 128 bits wide), a unified (instructions and data) 256-KB local store (LS), an instruction-control unit, a load and store unit, two fixed-point units, a floating-point unit, and a channel-and-DMA interface. The SPU implements a new SIMD instruction set, the *SPU Instruction Set Architecture*, that is specific to the *Broadband Processor Architecture*.

Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the PPE. The SPU fetches instructions from its own LS, and it loads and stores data from and to its own LS. With respect to accesses by its SPU, the LS is unprotected and un-translated storage. The MFC contains a DMA controller that supports DMA transfers. Programs running on the SPU, the PPE, or another SPU, use the MFC's DMA transfers to move instructions and data between the SPU's LS and main storage. (Main storage is the effective-address space that includes main memory, other SPEs' LS, and memory-mapped registers such as memory-mapped I/O [MMIO] registers.) The MFC interfaces the SPU to the EIB, implements bus bandwidth-reservation features, and synchronizes operations between the SPU and all other processors in the system.

To support DMA transfers, the MFC maintains and processes queues of DMA commands. After a DMA command has been queued to the MFC, the SPU can continue to execute instructions while the MFC processes the DMA command autonomously and asynchronously. The MFC also can autonomously execute a sequence of DMA transfers, such as scatter-gather lists, in response to a DMA-list command. This autonomous execution of MFC DMA commands and SPU instructions allows DMA transfers to be conveniently scheduled to hide memory latency.

Each DMA transfer can be up to 16 KB in size. However, only the MFC's associated SPU can issue DMA-list commands. These can represent up to 2,048 DMA transfers, each one up to 16 KB in size. DMA transfers are coherent with respect to main storage. Virtual-memory address-translation information is provided to each MFC by the operating system running on the PPE. Attributes of system storage (address translation and protection) are governed by the page and

segment tables of the PowerPC Architecture. Although privileged software on the PPE can map LS addresses and certain MFC resources to the main-storage address space, enabling the PPE or other SPUs in the system to access these resources, this aliased memory is not coherent in the system.

The SPEs provide a deterministic operating environment. They do not have caches, so cache misses are not a factor in their performance. Pipeline-scheduling rules are simple, so it is easy to statically determine the performance of code. Although the LS is shared between DMA read and write operations, load and store operations, and instruction prefetch, DMA operations are accumulated and can only access the LS for at most one of every eight cycles. Instruction prefetch delivers at least 17 instructions sequentially from the branch target. Thus, the impact of DMA operations on loads and stores and program-execution times is, by design, limited.

---

## Programming Overview

The instruction set for the PPE is an extended version of the *PowerPC* instruction set. The extensions consist of the *Vector/SIMD Multimedia Extension* instruction set plus a few additions and changes to PowerPC instructions.

The instruction set for the SPE is similar to that of the PPE's *Vector/SIMD Multimedia Extension* instruction set. Although the PPE and the SPEs execute SIMD instructions, the two instruction sets are different, and programs for the PPE and SPEs must be compiled by different compilers.

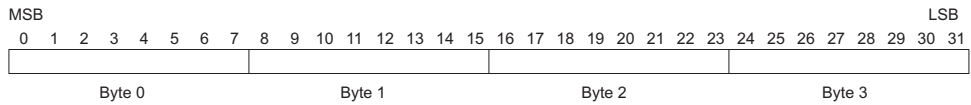
### Byte ordering and bit numbering

Storage of data and instructions in the Cell Broadband Engine is *big-endian*.

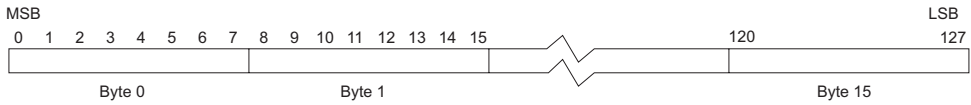
Big-endian ordering has the following characteristics:

- Most-significant byte is stored at the lowest address, and least-significant byte is stored at the highest address.
- Bit numbering within a byte goes from most-significant bit (bit 0) to least-significant bit (bit n). This differs from some other big-endian processors.

Figure 4 on page 10 shows a summary of the byte-ordering and bit-ordering in memory, as well as the bit-numbering conventions.



Bit and Byte Order for a 32-bit Word



Bit and Byte Order for a 128-bit Register

Figure 4. Big-endian byte and bit ordering

## SIMD vectorization

A vector is an instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be integer or floating-point values. Most Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called *SIMD operands* or *packed operands*.

SIMD processing exploits data-level parallelism. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. That is, a single instruction can be applied to multiple data elements in parallel.

Support for SIMD operations is pervasive in the Cell Broadband Engine. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instruction set. In the SPEs, they are supported by the SPU instruction set.

In both the PPE and SPEs, vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single register, and, for example, added to four other words in a different register in a single operation. Figure 5 on page 11 shows such an operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or 2 doublewords.

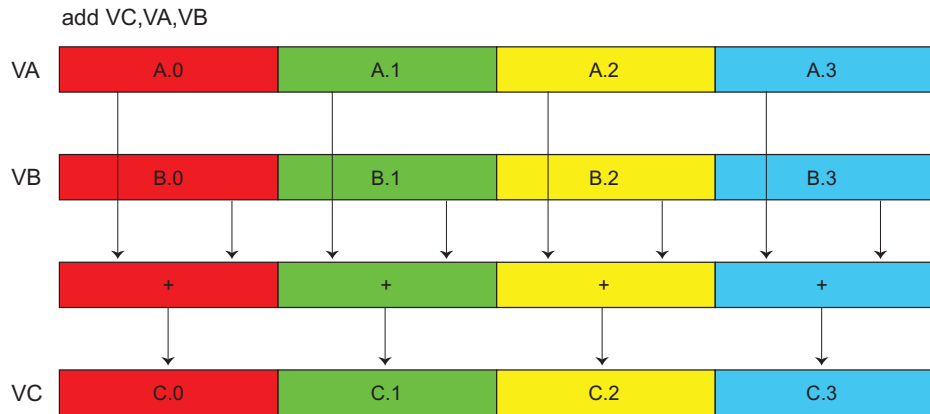


Figure 5. Four concurrent Add operations

The process of preparing a program for use on a vector processor is called *vectorization* or *SIMDization*. It can be done manually by the programmer, or it can be done by a compiler that does *auto-vectorization*.

Figure 6 shows another example of an SIMD operation— in this case, a byte-shuffle operation. Here, the bytes selected for the shuffle from the source registers, VA and VB, are based on byte entries in the control vector, VC, in which a 0 specifies VA and a 1 specifies VB. The result of the shuffle is placed in register VT.

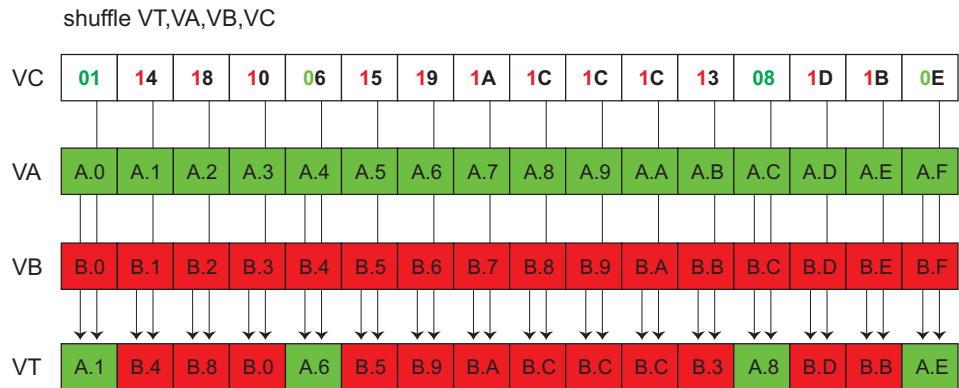


Figure 6. Byte-shuffle operation

## SIMD C-language intrinsics

Both the *Vector/SIMD Multimedia Extension* and *SPU* instruction sets have extensions that support C-language intrinsics. Intrinsics are C-language commands, in the form of function calls, that are convenient substitutes for one or more inline assembly-language instructions.

In a specific instruction set, most intrinsic names use a standard prefix in their mnemonic, and some intrinsic names incorporate the mnemonic of an associated

assembly-language instruction. For example, the Vector/SIMD Multimedia Extension intrinsic that implements the add Vector/SIMD Multimedia Extension assembly-language instruction is named `vec_add`, and the SPU intrinsic that implements the stop SPU assembly-language instruction is named `spu_stop`.

The PPE's Vector/SIMD Multimedia Extension instruction set and the SPE's SPU instruction set both have extensions that define somewhat different sets of intrinsics, but they all fall into four types of intrinsics. These are listed in Table 1-1. Although the intrinsics provided by the two instruction sets are similar in function, their naming conventions and function-call forms are different.

*Table 1. PPE and SPE intrinsic classes*

Types of Intrinsic	Definition	PPE	SPE
Specific	One-to-one mapping to a single assembly-language instruction.	X	X
Generic	Map to one or more assembly-language instructions, depending on types of input parameters.	X	X
Composite	Constructed from a sequence of Specific or Generic intrinsics.		X
Predicates	Evaluate SIMD conditionals.	X	

For more information about the PPE intrinsics, see “C/C++ language extensions (intrinsics)” on page 25.

For more information about the SPE intrinsics, see “SPU C/C++ language extensions (intrinsics)” on page 64.

## Threads and tasks

In a system running the Linux operating system, the main thread of a program is a Linux thread running on the PPE. The program's main Linux thread can spawn one or more Cell Broadband Engine Linux tasks.

A Cell Broadband Engine Linux task has one or more Linux threads associated with it that may execute on either a PPE or a SPE. An SPE thread is a Linux thread that is executing on a SPE. These terms are defined in Table 2.

The software threads described in this section are unrelated to the hardware multithreading capability of the PPE.

*Table 2. Definition of threads and tasks*

Term	Definition
Linux thread	A thread running in the Linux operating-system environment.
PPE thread	A Linux thread running on a PPE.
SPE thread	A Linux thread running on an SPE. Each such thread: <ul style="list-style-type: none"> <li>• has its own SPE context which includes the 128 x 128-bit register file, program counter, and MFC Command Queues.</li> <li>• can communicate with other execution units (or with effective-address memory through the MFC channel interface).</li> </ul>



Table 2. Definition of threads and tasks (continued)

Term	Definition
Cell Broadband Engine Linux task	<p>A task running on the PPE and SPE.</p> <ul style="list-style-type: none"> <li>• Each such task has one or more Linux threads.</li> <li>• All the Linux threads within the task share the task's resources.</li> </ul>

A Linux thread can interact directly with an SPE thread through the SPE's local store or its problem state. It can interact indirectly through effective-address (EA) memory or the interface provided by the SPE Runtime Management library subroutines.

The operating system defines the mechanism and policy for scheduling an available SPE. It must prioritize among all the Cell Broadband Engine Linux applications in the system, and it must schedule SPE execution independent from regular Linux threads. It is also responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support.

## The runtime environment

The PPE runs PowerPC applications and operating systems, which may include Vector/SIMD Multimedia Extension instructions.

The PPE requires an operating system that is extended to support the hardware features of Cell Broadband Engines, such as multiprocessing with the SPEs, access to the PPE Vector/SIMD Multimedia Extension functions, the Cell Broadband Engine interrupt controller, and all other functions on the Cell Broadband Engine.

The assumed development and operating-system environment for this tutorial are described in the "Preface" on page iii. In this operating environment, the PPE handles thread allocation and resource management among SPEs. The PPE's Linux kernel controls the SPUs' execution of programs.

SPE threads follow the M:N thread model, meaning M threads distributed over N processor elements. Typically SPE threads run to completion. However, the SPE threads are pre-emptible in accordance with the thread's scheduling policy and priority. Time slice quanta for the SPE threads is typically longer than PPE threads because of the SPE context switch is relatively heavy.

The Linux kernel manages virtual memory, including mapping each SPE's local store (LS) and problem state (PS) into the effective-address space. The kernel also controls virtual-memory mapping of MFC resources, as well as MFC segment-fault and page-fault handling. Large pages (16-MB pages), using the `hugetlbfs` Linux extension, are supported.

## Application partitioning

Programs running on the Cell Broadband Engine's nine processor elements typically partition the work among the available processor elements.

In determining when and how to distribute the workload and data, take into account the following considerations:

- processing-load distribution,
- program structure,

- program data flow and data access patterns,
- cost, in time and complexity of code movement and data movement among processors, and
- cost of loading the bus and bus attachments.

The main model for partitioning an application is *PPE-centric*, as shown in Figure 7.

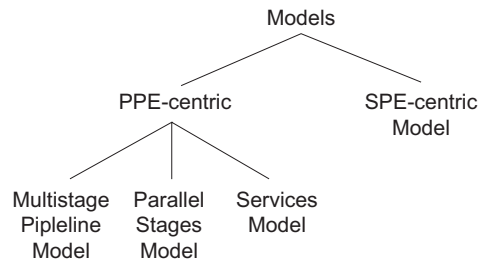


Figure 7. Application partitioning model

In the *PPE-centric model*, the main application runs on the PPE, and individual tasks are off-loaded to the SPEs. The PPE then waits for, and coordinates, the results returning from the SPEs. This model fits an application with serial data and parallel computation.

In the *SPE-centric model*, most of the application code is distributed among the SPEs. The PPE acts as a centralized resource manager for the SPEs. Each SPE fetches its next work item from main storage (or its own local store) when it completes its current work.

There are *three* ways in which the SPEs can be used in the PPE-centric model:

- the multistage pipeline model,
- the parallel stages model, and
- the services model.

The first two of these are shown in Figure 8 on page 15.

If a task requires sequential stages, the SPEs can act as a *multistage pipeline*. The left side of Figure 8 on page 15 shows a multistage pipeline. Here, the stream of data is sent into the first SPE, which performs the first stage of the processing. The first SPE then passes the data to the next SPE for the next stage of processing. After the last SPE has done the final stage of processing on its data, that data is returned to the PPE. As with any pipeline architecture, parallel processing occurs, with various portions of data in different stages of being processed.

Multistage pipelining is typically avoided because of the difficulty of load balancing. In addition, the multistage model increases the data-movement requirement because data must be moved for each stage of the pipeline.

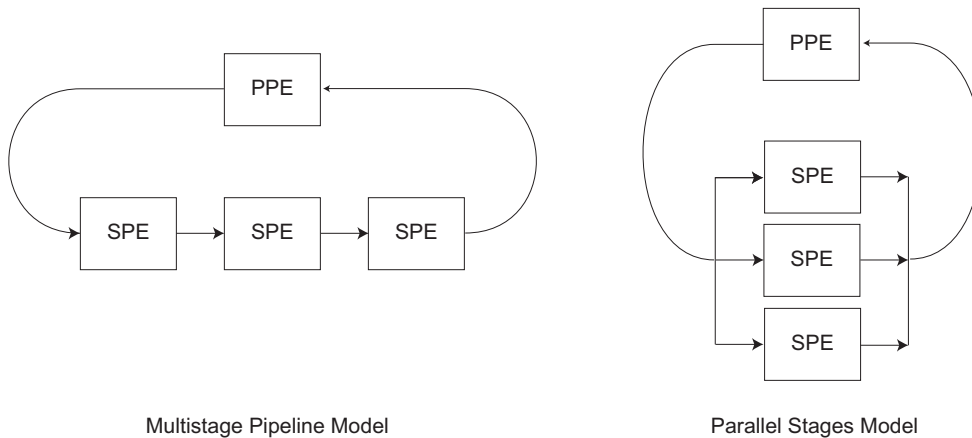


Figure 8. PPE-centric multistage pipeline model and parallel stages model

If the task to be performed is not a multistage task, but a task in which there is a large amount of data that can be partitioned and acted on at the same time, then it typically make sense to use SPEs to process different portions of that data in parallel. This *parallel stages model* is shown on the right side of Figure 8.

The third way in which SPEs can be used in a PPE-centric model is the *services model*. In the services model, the PPE assigns different services to different SPEs, and the PPE’s main process calls upon the appropriate SPE when a particular service is needed.

Figure 9 shows the *PPE-centric services model*. Here, one SPE processes data encryption, another SPE processes MPEG encoding, and a third SPE processes curve analysis. Fixed static allocation of SPU services should be avoided. These services should be virtualized and managed on a demand-initiated basis.

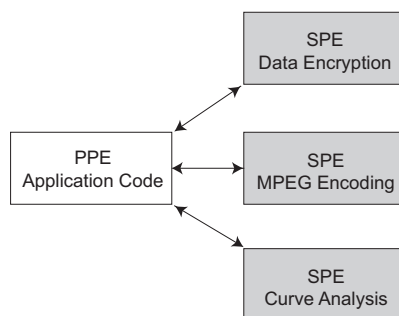


Figure 9. PPE-centric services model

For a more detailed view of programming models, see Chapter 4, “Programming models,” on page 123.

---

## The software development kit

A *software development kit* (SDK) is available for the Cell Broadband Engine.

The SDK contains the essential tools required for developing programs for the Cell Broadband Engine. “Preface” on page iii describes the assumptions with respect to the available SDK.

The SDK consists of numerous components including the following:

- System root image containing Linux execution environment for use within `systemsim`.
- GNU tools including C and C++ compilers, linkers, assemblers and binary utilities for both PPU and SPU.
- IBM `xlc` (C and C++) compiler for both PPU and SPU.
- IBM `xlf` (Fortran) compiler for both PPU and SPU.
- `newlib` for the SPU. `newlib` is a C standard library designed for use on embedded systems.
- `gdb` debuggers for both PPU and SPU with support for remote `gdbserver` debugging. The PPU debugger also provides combined, PPU and SPU, debugging.
- PPC64 Linux with CBE enhancements.
- SPE Runtime Management Library providing a standardized, low-level application programming interface for application access to the SPEs.
- Libraries to assist in the development and execution of *parallel applications*, including the following:
  - Accelerated Library Framework (ALF) library
  - Data Communication and Synchronization (DaCS) library
- Performance tools including:
  - `oprofile` – a system-wide profiler for Linux,
  - `CellPerfCount` – a low level tool to configure and access HW performance counters,
  - `FDPR-Pro` – a tool for gather information for feedback directed optimization,
  - `CodeAnalyzer` – examines executable files and displays detailed information about functions, basic blocks, and assembly instructions, and
  - `Visual Performance Analyzer (VPA)` – an Eclipse-based performance visualization toolkit.
  - `spu_timing` – a static timing analysis timing tool that instruments assembly source (either compiler or programmer generated) with expected, linear, instruction timing details.
  - `PDT` – a performance debugging tool which provides a tracing infrastructure for application timing analysis.
- An Eclipse-based Integrated Development Environment (IDE) to improve programmer productivity and integration of development tools.
- Standardized SIMD math libraries for the PPU’s Vector/SIMD Multimedia Extension and the SPU.
- Mathematical Acceleration Subsystem (MASS) libraries supporting both long and short (SIMD) vectors.
- Cell optimized domain-specific application libraries, including the following:
  - Basic Linear Algebra Subprograms (BLAS) library
  - Fast Fourier Transform (FFT) library
  - Monte Carlo Random Number Generator library

- |
  - Linear Algebra PACKage (LAPACK) library
  - Example source code containing programming examples, example libraries, benchmarks, and demos.



---

## Chapter 2. The PPE and the programming process

This chapter describes the PowerPC Processor Element (PPE) registers, the PPE's two instruction sets, and the C-language intrinsics for the PPE and Vector/SIMD Multimedia Extension instructions.

"The PowerPC Processor Element" on page 6 introduced the organization and functions of the PowerPC Processor Element (PPE). This chapter describes the relation between the PPE and *Synergistic Processor Element (SPE) address spaces*.

Examples are provided of:

- PPE-initiated DMA transfers between main storage and an SPE's local store (LS).
- PPE thread-creation for the SPE.

---

### PPE registers

This section describes the complete set of PowerPC Processor Element (PPE) *user (problem-state) registers*.

Figure 10 on page 20 shows all the PPE user (problem-state) registers. All computational instructions operate only on registers – there are *no* computational instructions that modify storage.

To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be:

1. loaded into a register,
2. modified,
3. stored back to the target location.

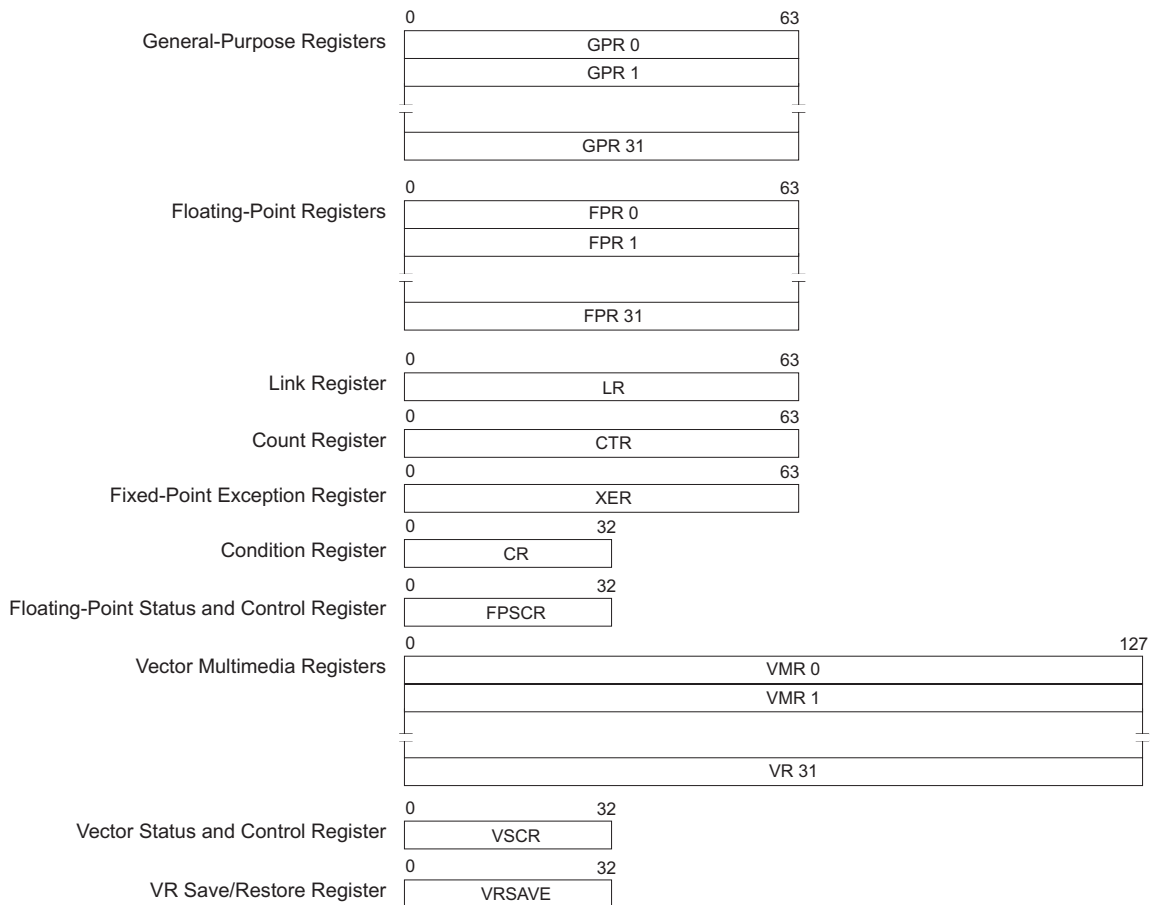


Figure 10. PPE user-register set

The PPE registers include:

- *General-Purpose Registers (GPRs)* – Fixed-point instructions operate on the full 64-bit width of the GPRs, of which there are 32. The instructions are mode-independent, except that in 32-bit mode, the processor uses only the low-order 32 bits for determination of a memory address and the carry, overflow, and record status bits.
- *Floating-Point Registers (FPRs)* – The 32 FPRs are 64 bits wide. The internal format of floating-point data is the IEEE 754 double-precision format. Single-precision results are maintained internally in the double-precision format.
- *Link Register (LR)* – The 64-bit LR can be used to hold the effective address of a branch target. Branch instructions with the link bit (LK) set to 1 (that is, subroutine-call instructions) copy the next instruction address into the LR. A Move To Special-Purpose Register instruction can copy the contents of a GPR into the LR.
- *Count Register (CTR)* – The 64-bit CTR can be used to hold either a loop counter or the effective address of a branch target. Some conditional-branch instruction



forms decrement the CTR and test it for a zero value. A Move To Special-Purpose Register instruction can copy the contents of a GPR into the CTR.

- *Fixed-Point Exception Register (XER)* – The 64-bit XER contains the carry and overflow bits and the byte count for the move-assist instructions. Most arithmetic operations have instruction forms for setting the carry and overflow bit.
- *Condition Register (CR)* – Conditional comparisons are performed by first setting a condition code in the 32-bit CR with a compare instruction or with a recording instruction. The condition code is then available as a value or can be tested by a branch instruction to control program flow. The CR consists of eight independent 4-bit fields grouped together for convenient save or restore during a context switch. Each field can hold status information from a comparison, arithmetic, or logical operation. The compiler can schedule CR fields to avoid data hazards in the same way that it schedules the use of GPRs. Writes to the CR occur only for instructions that explicitly request them; most operations have recording and non-recording instruction forms.
- *Floating-Point Status and Control Register (FPSCR)* – The processor updates the 32-bit FPSCR after every floating-point operation to record information about the result and any associated exceptions. The status information required by IEEE 754 is included, plus some additional information for exception handling.
- *Vector Registers (VRs)* – There are 32 128-bit-wide VRs. They serve as source and destination registers for all vector instructions.
- *Vector Status and Control Register (VSCR)* – The 32-bit VSCR is read and written in a manner similar to the FPSCR. It has 2 defined bits, a non-Java™ mode bit and a saturation bit; the remaining bits are reserved. Special instructions are provided to move the VSCR to a VR register.
- *Vector Save Register (VRSAVE)* – The 32-bit VRSAVE register assists user and privileged software in saving and restoring the architectural state across context switches.

---

## PPE instruction sets

The PowerPC Processor Element (PPE) supports *two* instruction sets: the *PowerPC instruction set* and the *Vector/SIMD Multimedia Extension instruction set*.

Although most of the coding for the Cell Broadband Engine will be in a high-level language like C or C++, an understanding of the PPE architecture and instruction sets adds considerably to a developer's ability to produce efficient, optimized code. This is particularly true because C-language intrinsics are provided for the PPE's Vector/SIMD Multimedia Extension instruction set, and these intrinsics map directly to one or more Vector/SIMD Multimedia Extension assembly-language instructions.

The PowerPC instruction set uses instructions that are 4 bytes long and word-aligned. It supports byte, halfword, word, and doubleword operand accesses between storage and its 32 general-purpose registers (GPRs). The instruction set also supports word and doubleword operand accesses between storage and a set of 32 floating-point registers (FPRs). Signed integers are represented in twos-complement form.

The Vector/SIMD Multimedia Extension instruction set uses instructions that, like PowerPC instructions, are 4 bytes long and word-aligned. However, all of its operands are 128 bits wide. Most of the Vector/SIMD Multimedia Extension

operands are vectors, including single-precision floating-point, integer, scalar, and fixed-point of vector-element sizes of 8,16, and 32 bits.

The sections that follow briefly summarize key points of the instruction sets. However, for a complete description of the PowerPC instruction sets, refer to these publications:

- *PowerPC Microprocessor Family, Programming Environments Manual for 64-Bit Microprocessors*
- *PowerPC Microprocessor Family, Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

## PowerPC instructions

Whenever instruction addresses are presented to the processor, the low-order 2 bits are ignored.

Similarly, whenever the processor develops an instruction address, the low-order 2 bits are zero. The address of either an instruction or a multiple-byte data value is its lowest-numbered byte. This address points to the most-significant end (big-endian convention). The little-endian convention is *not* supported.

Arithmetic for address computation is unsigned and ignores any carry out of bit 0 (the MSb).

For an overview of the big-endian bit and byte numbering used by the PPE, see “Byte ordering and bit numbering” on page 9.

### Addressing modes

All instructions, except branches, generate addresses by incrementing a program counter. All load and store instructions specify a base register.

The effective address in memory for a data value is calculated relative to the base register in one of three ways:

- *Register + Displacement* – The displacement forms of the load and store instructions calculate an address that is the sum of a displacement specified by the sign-extended 16-bit immediate field of the instruction plus the contents of the base register.
- *Register + Register* – The indexed forms of the load and store instructions calculate an address that is the sum of the contents of the index register, which is a GPR, plus the contents of the base register.
- *Register* – The Load String Immediate and Store String Immediate instructions use the unmodified contents of the base register to calculate an address.

Loads and stores can specify an update form that reloads the base register with the computed address, unless the base register is the target register of the load.

Branches are the only instructions that explicitly specify the address of the next instruction. A branch instruction specifies the effective address of the branch target in one of the following ways:

- *Branch Not Taken* – The byte address of the next instruction is the byte address of the current instruction, plus 4.
- *Absolute* – The word address of the next instruction is given in an immediate field of the branch instruction.

- *Relative* – The word address of the next instruction is given by the sum of the immediate field of the branch instruction and the word address of the branch instruction itself.
- *Link Register or Count Register* – The byte address of the next instruction is the effective byte address of the branch target specified in the Link Register or Count Register, respectively.

## Instruction types

The PowerPC Processor Element (PPE)'s *PowerPC instructions* can have up to *three* operands. Most computational instructions specify *two* source operands and *one* destination operand.

The PPE's PowerPC instructions include the following types:

- *Integer Instructions* – These include arithmetic, compare, logical, and rotate/shift instructions. They operate on byte, halfword, word, and doubleword operands.
- *Floating-Point Instructions* – These include floating-point arithmetic, multiply-add, compare, and move instructions, as well as instructions that affect the Floating-Point Status and Control Register (FPSCR). Floating-point instructions operate on single-precision and double-precision floating-point operands.
- *Load and Store Instructions* – These include integer and floating-point load and store instructions, with byte-reverse, multiple, and string options for the integer loads and stores.
- *Memory Synchronization Instructions* – These instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory-access mechanisms. The instruction types include load and store with reservation, synchronization, and enforce in-order execution of I/O. They are especially useful for multiprocessing.
- *Flow Control Instructions* – These include branch, Condition-Register logical, trap, and other instructions that affect the instruction flow.
- *Processor Control Instructions* – These instructions are used for synchronizing memory accesses and managing caches, Translation Lookaside Buffers (TLBs), segment registers, and other privileged processor states. They include move-to/from special-purpose register instructions.
- *Memory and Cache Control Instructions* – These instructions control caches, TLBs, and segment registers.
- *External Control Instructions* – These instructions allow a user-level program to communicate with a special-purpose device.

## Compatibility with existing PowerPC code

The *PowerPC Processor Element (PPE)* complies with version 2.0.2 of the PowerPC architecture, with only *minor exceptions*.

The following optional *user-mode* instructions *are* implemented:

- `fsqrt(.)` – Floating-point square root
- `fsqrts(.)` – Floating-point square root single
- `fres(.)` – Floating-point reciprocal estimate single, A-form
- `frsqrte(.)` – Floating-point reciprocal square root estimate, A-form
- `fsel(.)` – Floating-point select
- `mtocrf` – Move to one condition register field, XFX-form
- `mfocrf` – Move from one condition register field, XFX-form

The following optional instructions that are defined in the PowerPC Book I are *not* implemented. Use of these instructions will cause an illegal-instruction interrupt:

- mcrxr – Move to condition register from XER
- bccbr – Branch condition to CBR

The following instructions that are not defined in the PowerPC Architecture *are* implemented. Since these instructions are not part of the architecture, they should be considered highly implementation-specific.

- ldbrx – Load doubleword byte reverse indexed, X-form
- sdbrx – Store doubleword byte reverse indexed, X-form

In addition, the little endian option for data ordering is not available. A complete list of differences can be found in the *Cell Broadband Engine, Programming Handbook*.

## Vector/SIMD Multimedia Extension instructions

The 128-bit *Vector/SIMD Multimedia Extension unit (VXU)* operates concurrently with the PPU's fixed-point integer unit (FXU) and floating-point execution unit (FPU).

Like PowerPC instructions, the Vector/SIMD Multimedia Extension instructions are 4 bytes long and word-aligned. The Vector/SIMD Multimedia Extension instructions support simultaneous execution on multiple elements that make up the 128-bit vector operands. These vector elements may be byte, halfword, or word.

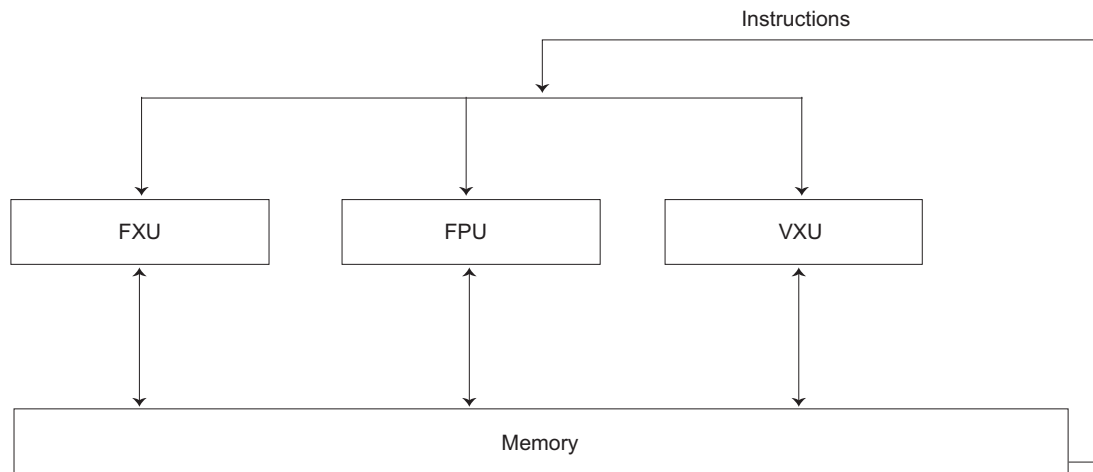


Figure 11. Concurrent execution of integer, floating-point, and vector units

The Vector/SIMD Multimedia Extension instructions are fully described in the *PowerPC Microprocessor Family, Vector/SIMD Multimedia Extension Technology Programming Environments* manual.

All Vector/SIMD Multimedia Extension instructions are designed to be easily “pipelined”. Parallel execution with the PPE’s integer and floating-point instructions is simplified by the fact that Vector/SIMD Multimedia Extension instructions:

- do *not* generate exceptions (other than data-storage interrupt exceptions on loads and stores),

- do *not* support unaligned memory accesses or complex functions, and
- share few resources or communication paths with the other PPE execution units.

### Addressing modes

The PowerPC Processor Element (PPE) supports not only basic load and store operations, but also load and store vector left- or right-indexed forms.

All Vector/SIMD Multimedia Extension load and store operations use the register + register indexed addressing mode, which forms the sum of the contents of an index GPR plus the contents of a base-address GPR. This addressing mode is very useful for accessing arrays.

In addition to the load and store operations, the Vector/SIMD Multimedia Extension instruction set provides a powerful set of element-manipulation instructions – for example, shuffle, permute (similar to the SPEs’ shuffle), rotate, and shift – to manipulate vector elements into the desired alignment and arrangement after the vectors have been loaded into vector registers.

### Instruction types

Most Vector/SIMD Multimedia Extension instructions have three or four 128-bit vector operands – two or three source operands and one result. Also, most instructions are SIMD in nature.

The instructions have been chosen for their utility in digital signal processing (DSP) algorithms, including 3D graphics.

The Vector/SIMD Multimedia Extension instructions include the following types:

- *Vector Integer Instructions* – These include vector arithmetic, compare, logical, rotate, and shift instructions. They operate on byte, halfword, and word vector elements. The instructions use saturation-clamping.
- *Vector Floating-Point Instructions* – These include floating-point arithmetic, multiply/add, rounding and conversion, compare, and estimate instructions. They operate on single-precision floating-point vector elements.
- *Vector Load and Store Instructions* – These include only basic integer and floating-point load and store instructions. No update forms of the load and store instruction are provided. They operate on 128-bit vectors.
- *Vector Permutation and Formatting Instructions* – These include vector pack, unpack, merge, splat, permute, select, and shift instructions.
- *Processor Control Instructions* – These include instructions that read and write the vector status and control register (VSCR).
- *Memory Control Instructions* – These include instructions for managing caches (user-level and supervisor-level). These instructions are “no-ops”.

## C/C++ language extensions (intrinsics)

A set of C-language extensions are available for PowerPC Processor Element (PPE) and Vector/SIMD Multimedia Extension programming.

These extensions include additional vector data types and a large set of scalar and vector commands (intrinsics). The intrinsics are essentially inline assembly-language instructions, in the form of function calls, that have syntax familiar to high-level programmers using the C language.

The intrinsics provide explicit control of the PPE or Vector/SIMD Multimedia Extension instructions without directly managing registers and scheduling

instructions, as assembly-language programming requires. A compiler that supports these C-language extensions will emit code optimized for the PPE and/or the Vector/SIMD Multimedia Extension architecture.

## Scalar intrinsics

A minimal set of specific *intrinsics*, to make the underlying PPU instruction set accessible from the C programming language, have been provided. These intrinsics are declared in the system header file `ppu_intrinsics.h`.

Table 3. PPE-specific scalar intrinsics

Intrinsic	Description
<code>__cctph()</code>	Change Thread Priority to High
<code>__cctpl()</code>	Change Thread Priority to Low
<code>__cctpm()</code>	Change Thread Priority to Medium
<code>d = __cntlz(a)</code>	Count Leading Doubleword Zeros
<code>d = __cntlzw(a)</code>	Count Leading Word Zeros
<code>__db10cyc()</code>	Delay 10 Cycles at Dispatch
<code>__db12cyc()</code>	Delay 12 Cycles at Dispatch
<code>__db16cyc()</code>	Delay 16 Cycles at Dispatch
<code>__db8cyc()</code>	Delay 8 Cycles at Dispatch
<code>__dcbf(pointer)</code>	Data Cache Block Flush
<code>__dcbst(pointer)</code>	Data Cache Block Store
<code>__dcbt(pointer)</code>	Data Cache Block Touch
<code>__dcbt_TH1000(eatrunc, d, ug, id)</code>	Start Streaming Data
<code>__dcbt_TH1010(g0, s, unitcnt, t, u, id)</code>	Stop Streaming Data
<code>__dcbtst(pointer)</code>	Data Cache Block Touch for Store
<code>__dcbz(pointer)</code>	Data Cache Block Set to Zero
<code>__eieio()</code>	Enforce In-Order Execution of I/O
<code>d = __fabs(a)</code>	Double Absolute Value
<code>d = __fabsf(a)</code>	Float Absolute Value
<code>d = __fcfid(a)</code>	Convert Doubleword to Double
<code>d = __fctid(a)</code>	Convert Double to Doubleword
<code>d = __fctidz(a)</code>	Convert Double to Doubleword with Round Towards Zero
<code>d = __fctiw(a)</code>	Convert Double to Word
<code>d = __fctiwz(a)</code>	Convert Double to Word with Round Toward Zero
<code>d = __fmadd(a,b,c)</code>	Double Fused Multiply and Add
<code>d = __fmadds(a,b,c)</code>	Float Fused Multiply and Add
<code>d = __fmsub(a,b,c)</code>	Double Fused Multiply and Subtract
<code>d = __fmsubs(a,b,c)</code>	Float Fused Multiply and Subtract
<code>d = __fmul(a,b)</code>	Double Multiply
<code>d = __fmuls(a,b)</code>	Float Multiply
<code>d = __fnabs(a)</code>	Double Negative Absolute Value
<code>d = __fnabsf(a)</code>	Float Negative Absolute Value

Table 3. PPE-specific scalar intrinsics (continued)

Intrinsic	Description
d = __fnmadd(a,b,c)	Double Fused Negative Multiply and Add
d = __fnmadds(a,b,c)	Float Fused Negative Multiply and Add
d = __fnmsub(a,b,c)	Double Fused Negative Multiply and Subtract
d = __fnmsubs(a,b,c)	Float Fused Negative Multiply and Subtract
d = __fres(a)	Float Reciprocal Estimate
d = __frsp(a)	Round to Single Precision
d = __fsel(a,b,c)	Floating Point Select of Double
d = __fsels(a,b,c)	Floating Point Select of Float
d = __fsqrt(a)	Double Square Root
d = __fsqrts(a)	Float Square Root
__icbi(pointer)	Instruction Cache Block Invalidate
__isync()	Instruction Sync
d = __ldarx(pointer)	Load Doubleword with Reserved
d = __ldbrx(pointer)	Load Reversed Doubleword
d = __lhbrx(pointer)	Load Reversed Halfword
d = __lwarx(pointer)	Load Word with Reserved
d = __lwbrx(pointer)	Load Reversed Word
__lwsync()	Light Weight Sync
d = __mffs()	Move from Floating-Point Status and Control Register
d = __mfspr(spr)	Move from Special Purpose Register
d = __mftb()	Move from Time Base
__mtfsb0(bt)	Unset Field of FPSCR
__mtfsb1(bt)	Set Field of FPSCR
__mtfsf(flm,b)	Set Fields of FPSCR
__mtfsfi(bf,u)	Set Field FPSCR from other Field
__mtspr(spr,value)	Move to Special Purpose Register
d = __mulhdu(a,b)	Multiply Double Unsigned Word, High Part
d = __mulhd(a,b)	Multiply Doubleword, High Part
d = __mulhwu(a,b)	Multiply Unsigned Word, High Part
d = __mulhw(a,b)	Multiply Word, High Part
__nop()	No Operation
__protected_stream_count(count,id)	Set the Number of Blocks to Stream
__protected_stream_go()	Start All Streams
__protected_stream_set(d,addr,id)	Set Up a Stream
__protected_stream_stop(id)	Stop a Stream
__protected_stream_stop_all()	Stop All Streams



Table 3. PPE-specific scalar intrinsics (continued)

Intrinsic	Description
<code>__protected_unlimited_stream_set(d,addr,id)</code>	Set Up an Unlimited Stream
<code>d = __rdcl(a,b,mb)</code>	Rotate Left Doubleword then Clear Left
<code>d = __rdcr(a,b,me)</code>	Rotate Left Doubleword then Clear Right
<code>d = __rdic(a,sh,mb)</code>	Rotate Left Doubleword Immediate then Clear
<code>d = __rdicl(a,sh,mb)</code>	Rotate Left Doubleword Immediate then Clear Left
<code>d = __rdicr(a,sh,me)</code>	Rotate Left Doubleword Immediate then Clear Right
<code>d = __rdimi(a,b,sh,mb)</code>	Rotate Left Doubleword Immediate then Mask Insert
<code>d = __rlwimi(a,b,sh,mb,me)</code>	Rotate Left Word Immediate the Mask Insert
<code>d = __rlwinm(a,sh,mb,me)</code>	Rotate Left Word Immediate then AND with Mask
<code>d = __rlwnm(a,v,mb,me)</code>	Rotate Left Word then AND with Mask
<code>d = __setflm(a)</code>	Save and Set the FPSCR
<code>__stbrx(pointer,b)</code>	Store Reversed Doubleword
<code>d = __stcx(pointer,b)</code>	Store Doubleword Conditional
<code>__stbrx(pointer,b)</code>	Store Reversed Halfword
<code>__stwbrx(pointer,b)</code>	Store Reversed Word
<code>d = __stwcx(pointer,b)</code>	Store Word Conditional
<code>__sync()</code>	Sync

## Vector data types

The Vector/SIMD Multimedia Extension model adds a set of fundamental data types, called *vector types*.

Vector types are shown in Table 4 on page 29. The represented values are in decimal (base-10) notation. The vector registers are 128 bits and can contain:

- Sixteen 8-bit values, signed or unsigned
- Eight 16-bit values, signed or unsigned
- Four 32-bit values, signed or unsigned
- Four single-precision IEEE-754 floating-point values

The vector types use the prefix `vector` in front of one of standard C data types—for example `vector signed int` and `vector unsigned short`. A vector type represents a vector of as many of the specified C data type as will fit in a 128-bit register. Hence, the `vector signed int` is a 128-bit operand containing four 32-bit signed ints. The `vector unsigned short` is a 128-bit operand containing eight unsigned values.

**Note:** Since the token, `vector`, is a keyword in the Vector/SIMD Multimedia Extension data types, you are recommended *not* to use the term elsewhere in the program (for example, as a variable name).



Table 4. Vector/SIMD Multimedia Extension data types

Vector Data Type	Meaning	Values
vector unsigned char	Sixteen 8-bit unsigned values	0 ... 255
vector signed char	Sixteen 8-bit signed values	-128 ... 127
vector bool char	Sixteen 8-bit unsigned boolean	0 (false), 255 (true)
vector unsigned short	Eight 16-bit unsigned values	0 ... 65535
vector unsigned short int	Eight 16-bit unsigned values	0 ... 65535
vector signed short	Eight 16-bit signed values	-32768 ... 32767
vector signed short int	Eight 16-bit signed values	-32768 ... 32767
vector bool short	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector bool short int	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector unsigned int	Four 32-bit unsigned values	0 ... $2^{32} - 1$
vector signed int	Four 32-bit signed values	$-2^{31} ... 2^{31} - 1$
vector bool int	Four 32-bit unsigned values	0 (false), $2^{31} - 1$ (true)
vector float	Four 32-bit single precision	IEEE-754 values
vector pixel	Eight 16-bit unsigned values	1/5/5/5 pixel

Introducing fundamental vector data types permits the compiler to provide stronger type-checking and supports overloaded operations on vector types.

### Vector intrinsics

Vector/SIMD Multimedia Extension intrinsics are grouped into *three* classes.

These classes are:

- *Specific Intrinsics* – Intrinsics that have a one-to-one mapping with a single assembly-language instruction
- *Generic Intrinsics* – Intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters
- *Predicates Intrinsics* – Intrinsics that compare values and return an integer that may be used directly as a value or as a condition for branching

The Vector/SIMD Multimedia Extension intrinsics and predicates use the prefix `vec_` in front of an assembly-language or operation mnemonic; predicate intrinsics use the prefixes `vec_all` and `vec_any`. When compiled, the intrinsics generate one or more Vector/SIMD Multimedia Extension assembly-language instructions.

The specific and generic intrinsics are shown in Table 5. The predicate intrinsics are shown in Table 6 on page 32.

Table 5. Vector/SIMD Multimedia Extension specific and generic intrinsics

Intrinsic	Description
<b>Arithmetic Intrinsics</b>	
<code>d = vec_abs(a)</code>	Vector Absolute Value
<code>d = vec_abss(a)</code>	Vector Absolute Value Saturated
<code>d = vec_add(a,b)</code>	Vector Add
<code>d = vec_addc(a,b)</code>	Vector Add Carryout Unsigned Word
<code>d = vec_adds(a,b)</code>	Vector Add Saturated

Table 5. Vector/SIMD Multimedia Extension specific and generic intrinsics (continued)

Intrinsic	Description
d = vec_avg(a,b)	Vector Average
d = vec_madd(a,b,c)	Vector Multiply Add
d = vec_madds(a,b,c)	Vector Multiply Add Saturated
d = vec_max(a,b)	Vector Maximum
d = vec_min(a,b)	Vector Minimum
d = vec_mladd(a,b,c)	Vector Multiply Low and Add Unsigned Half Word
d = vec_mradds(a,b,c)	Vector Multiply Round and Add Saturated
d = vec_msum(a,b,c)	Vector Multiply Sum
d = vec_msums(a,b,c)	Vector Multiply Sum Saturated
d = vec_mule(a,b)	Vector Multiply Even
d = vec_mulo(a,b)	Vector Multiply Odd
d = vec_nmsub(a,b,c)	Vector Negative Multiply Subtract
d = vec_sub(a,b)	Vector Subtract
d = vec_subc(a,b)	Vector Subtract Carryout
d = vec_subs(a,b)	Vector Subtract Saturated
d = vec_sum4s(a,b)	Vector Sum Across Partial (1/4) Saturated
d = vec_sum2s(a,b)	Vector Sum Across Partial (1/2) Saturated
d = vec_sums(a,b)	Vector Sum Saturated
<b>Rounding And Conversion Intrinsics</b>	
d = vec_ceil(a)	Vector Ceiling
d = vec_ctf(a,b)	Vector Convert from Fixed-Point Word
d = vec_cts(a,b)	Vector Convert to Signed Fixed-Point Word Saturated
d = vec_ctu(a,b)	Vector Convert to Unsigned Fixed-Point Word Saturated
d = vec_floor(a)	Vector Floor
d = vec_trunc(a)	Vector Truncate
<b>Floating-Point Estimate Intrinsics</b>	
d = vec_expte(a)	Vector Is 2 Raised to the Exponent Estimate Floating-Point
d = vec_log2(a)	Vector Log2 Estimate Floating-Point
d = vec_re(a)	Vector Reciprocal Estimate
d = vec_rsqtrt(a)	Vector Reciprocal Square Root Estimate
<b>Compare Intrinsics</b>	
d = vec_cmpb(a,b)	Vector Compare Bounds Floating-Point
d = vec_cmpeq(a,b)	Vector Compare Equal
d = vec_cmpge(a,b)	Vector Compare Greater Than or Equal
d = vec_cmpgt(a,b)	Vector Compare Greater Than
d = vec_cmple(a,b)	Vector Compare Less Than or Equal
d = vec_cmplt(a,b)	Vector Compare Less Than
<b>Logical Intrinsics</b>	
d = vec_and(a,b)	Vector Logical AND

Table 5. Vector/SIMD Multimedia Extension specific and generic intrinsics (continued)

Intrinsic	Description
d = vec_andc(a,b)	Vector Logical AND with Complement
d = vec_nor(a,b)	Vector Logical NOR
d = vec_or(a,b)	Vector Logical OR
d = vec_xor(a,b)	Vector Logical XOR
<b>Rotate and Shift Intrinsics</b>	
d = vec_rl(a,b)	Vector Rotate Left
d = vec_round(a)	Vector Round
d = vec_sl(a,b)	Vector Shift Left
d = vec_sld(a,b,c)	Vector Shift Left Double
d = vec_sll(a,b)	Vector Shift Left Long
d = vec_slo(a,b)	Vector Shift Left by Octet
d = vec_sr(a,b)	Vector Shift Right
d = vec_sra(a,b)	Vector Shift Right Algebraic
d = vec_srl(a,b)	Vector Shift Right Long
d = vec_sro(a,b)	Vector Shift Right by Octet
<b>Load and Store Intrinsics</b>	
d = vec_ld(a,b)	Vector Load Indexed
d = vec_lde(a,b)	Vector Load Element Indexed
d = vec_ldl(a,b)	Vector Load Indexed LRU
d = vec_lvlx(a,b)	Load Vector Left Indexed
d = vec_lvxl(a,b)	Load Vector Left Indexed Last
d = vec_lvr(a,b)	Load Vector Right Indexed
d = vec_lvrxl(a,b)	Load Vector Right Indexed Last
d = vec_lvsl(a,b)	Vector Load for Shift Left
d = vec_lvsr(a,b)	Vector Load Shift Right
d = vec_stvlx(a,b)	Store Vector Left Indexed
d = vec_stvxl(a,b)	Store Vector Left Indexed Last
d = vec_stvr(a,b)	Store Vector Right Indexed
d = vec_stvrxl(a,b)	Store Vector Right Indexed Last
vec_st(a,b,c)	Vector Store Indexed
vec_ste(a,b,c)	Vector Store Element Indexed
vec_stl(a,b,c)	Vector Store Indexed LRU
<b>Pack and Unpack Intrinsics</b>	
d = vec_pack(a,b)	Vector Pack
d = vec_packpx(a,b)	Vector Pack Pixel
d = vec_packs(a,b)	Vector Pack Saturated
d = vec_packsu(a,b)	Vector Pack Saturated Unsigned
d = vec_unpackh(a)	Vector Unpack High Element
d = vec_unpackl(a)	Vector Unpack Low Element
<b>Merge Intrinsics</b>	

Table 5. Vector/SIMD Multimedia Extension specific and generic intrinsics (continued)

Intrinsic	Description
d = vec_mergeh(a,b)	Vector Merge High
d = vec_mergel(a,b)	Vector Merge Low
<b>Permute and Select Intrinsics</b>	
d = vec_perm(a,b,c)	Vector Permute
d = vec_sel(a,b,c)	Vector Select
<b>Stream Intrinsics</b>	
vec_dss(a)	Vector Data Stream Stop
vec_dssall()	Vector Stream Stop All
vec_dst(a,b,c)	Vector Data Stream Touch
vec_dstst(a,b,c)	Vector Data Stream Touch for Store
vec_dststt(a,b,c)	Vector Data Stream Touch for Store Transient
vec_dsttt(a,b,c)	Vector Data Stream Touch Transient
<b>Move Intrinsics</b>	
d = vec_mvfscr	Vector Move from Vector Status and Control Register
vec_mtvscr(a)	Vector Move to Vector Status and Control Register
<b>Replicate Intrinsics</b>	
d = vec_splat(a,b)	Vector Splat
d = vec_splat_s8(a)	Vector Splat Signed Byte
d = vec_splat_s16(a)	Vector Splat Signed Half-Word
d = vec_splat_s32(a)	Vector Splat Signed Word
d = vec_splat_u8(a)	Vector Splat Unsigned Byte
d = vec_splat_u16(a)	Vector Splat Unsigned Half-Word
d = vec_splat_u32(a)	Vector Splat Unsigned Word
<b>Scalar Intrinsics</b>	
d = vec_extract(a,element)	Extract Vector Element from Vector
d = vec_insert(a,b,element)	Insert Scalar into Specified Vector Element
d = vec_promote(a,element)	Promote Scalar to a Vector
d = vec_splats(a)	Splat Scalar to Vector

Table 6. Vector/SIMD Multimedia Extension predicate intrinsics

Predicate	Description
<b>All Predicates</b>	
d = vec_all_eq(a,b)	All Elements Equal
d = vec_all_ge(a,b)	All Elements Greater Than or Equal
d = vec_all_gt(a,b)	All Elements Greater Than
d = vec_all_in(a,b)	All Elements in Bounds
d = vec_all_le(a,b)	All Elements Less Than or Equal
d = vec_all_lt(a,b)	All Elements Less Than
d = vec_all_nan(a)	All Elements Not a Number
d = vec_all_ne(a,b)	All Elements Not Equal

Table 6. Vector/SIMD Multimedia Extension predicate intrinsics (continued)

Predicate	Description
d = vec_all_nge(a,b)	All Elements Not Greater Than or Equal
d = vec_all_ngt(a,b)	All Elements Not Greater Than
d = vec_all_nle(a,b)	All Elements Not Less Than or Equal
d = vec_all_nlt(a,b)	All Elements Not Less Than
d = vec_all_numeric(a)	All Elements Numeric
<b>Any Predicates</b>	
d = vec_any_eq(a,b)	Any Element Equal
d = vec_any_ge(a,b)	Any Element Greater Than or Equal
d = vec_any_gt(a,b)	Any Element Greater Than
d = vec_any_le(a,b)	Any Element Less Than or Equal
d = vec_any_lt(a,b)	Any Element Less Than
d = vec_any_nan(a)	Any Element Not a Number
d = vec_any_ne(a,b)	Any Element Not Equal
d = vec_any_nge(a,b)	Any Element Not Greater Than or Equal
d = vec_any_ngt(a,b)	Any Element Not Greater Than
d = vec_any_nle(a,b)	Any Element Not Less Than or Equal
d = vec_any_nlt(a,b)	Any Element Not Less Than
d = vec_any_numeric(a)	Any Element Numeric
d = vec_any_out(a,b)	Any Element Out of Bounds

## Programming with Vector/SIMD Multimedia Extension intrinsics

Vector/SIMD Multimedia Extension data types and Vector/SIMD Multimedia Extension intrinsics can be used in a seamless way throughout a C-language program.

You do not need to setup, to enter a special mode, or to include a special header file.

### Example: incorporating Vector instructions into a PPE program

The sample program `vmx_sample` illustrates the ease with which vector instructions can be incorporated into a PPE program.

The program `vmx_sample` performs this processing:

1. “typedefs” a union of an array of four ints and a vector of signed ints. This is only done so we can refer to the values in two different ways. (Vector elements can also be accessed using the SPU intrinsic, `spu_extract`. For more information about SPU intrinsics, see “Intrinsic classes” on page 66.
2. Loads the literal value 2 into each of the four 32-bit fields of vector `vConst`.
3. Loads four different integer values into the fields of vector `v1`.
4. Calls the `vec_add` intrinsic, and the two vectors are added with the result being assigned to `v2`.

```
#include <stdio.h>
```

```
// Define a type we can look at either as an array of ints or as a vector.
```

```

typedef union {
    int iVals[4];
    vector signed int myVec;
} vecVar;

int main()
{
    vecVar v1, v2, vConst; // define variables

    // load the literal value 2 into the 4 positions in vConst,
    vConst.myVec = (vector signed int){2, 2, 2, 2};

    // load 4 values into the 4 element of vector v1
    v1.myVec = (vector signed int){10, 20, 30, 40};

    // call vector add function
    v2.myVec = vec_add( v1.myVec, vConst.myVec );

    // see what we got!
    printf("\nResults:\nv2[0] = %d, v2[1] = %d, v2[2] = %d, v2[3] = %d\n\n",
        v2.iVals[0], v2.iVals[1], v2.iVals[2], v2.iVals[3]);

    return 0;
}

```

See “Developing code for the Cell Broadband Engine” on page 41 for more information on how to run the example on the simulator.

Figure 12 shows the results of running the sample program.

---

```

[user@bringup /]# callthru source vmx_sample > vmx_sample
[user@bringup /]# chmod +x vmx_sample
[user@bringup /]# vmx_sample

Results:
v2[0] = 12, v2[1] = 22, v2[2] = 32, v2[3] = 42

[user@bringup /]# _

```

---

Figure 12. Running the Vector/SIMD Multimedia Extension sample program

### Example: array-summing

This example illustrates array-summing using a function that sums an input array of 16-byte values.

The following code contains *three* versions of a function that sums an input array of 16-byte values. For this kind of array-summing function, you have several options:

- You can unroll the scalar code to slightly improve the performance.
- You can use the Vector/SIMD Multimedia Extension to significantly improve the performance.
- You can eliminate the loop entirely.

The first option performs 16 iterations of the loop. The second option performs only four iterations of the loop but with four additions in each iteration. The third option uses Vector/SIMD Multimedia Extension intrinsics to eliminate the loop entirely.

```

// 16 iterations of a loop
int rolled_sum(unsigned char bytes[16])
{
    int i;
    int sum = 0;
    for (i = 0; i < 16; ++i) {
        sum += bytes[i];
    }
    return sum;
}

// 4 iterations of a loop, with 4 additions in each iteration
int unrolled_sum(unsigned char bytes[16])
{
    int i;
    int sum[4] = {0, 0, 0, 0};
    for (i = 0; i < 16; i += 4) {
        sum[0] += bytes[i + 0];
        sum[1] += bytes[i + 1];
        sum[2] += bytes[i + 2];
        sum[3] += bytes[i + 3];
    }
    return sum[0] + sum[1] + sum[2] + sum[3];
}

// Vectorized for Vector/SIMD Multimedia Extension
int vectorized_sum(unsigned char bytes[16])
{
    vector unsigned char vbytes;
    union {
        int i[4];
        vector signed int v;
    } sum;
    vector unsigned int zero = (vector unsigned int){0};

    // Perform a misaligned vector load of the 16 bytes.
    vbytes = vec_perm(vec_ld(0, bytes), vec_ld(16, bytes), vec_lvs1(0, bytes));

    // Sum the 16 bytes of the vector
    sum.v = vec_sums((vector signed int)vec_sum4s(vbytes, zero),
                    (vector signed int)zero);

    // Extract the sum and return the result.
    return (sum.i[3]);
}

```

---

## The PPE and the SPEs

This section describes the relationship between the *PowerPC Processor Element (PPE)* and the *Synergistic Processor Elements (SPEs)*.

### Storage Domains

Three types of storage domains are defined in the Cell Broadband Engine: one *main-storage domain*, eight *SPE local store domains*, and eight *SPE channel domains*.

The three types of storage domains are shown in Figure 13 on page 36. The main-storage domain, which is the entire effective-address space, can be configured by the PPE operating system to be shared by all processors and memory-mapped devices in the system (all I/O is memory-mapped).

However, the local-storage and channel problem-state (user-state) domains are private to the SPU, LS, and MFC of each SPE.

---

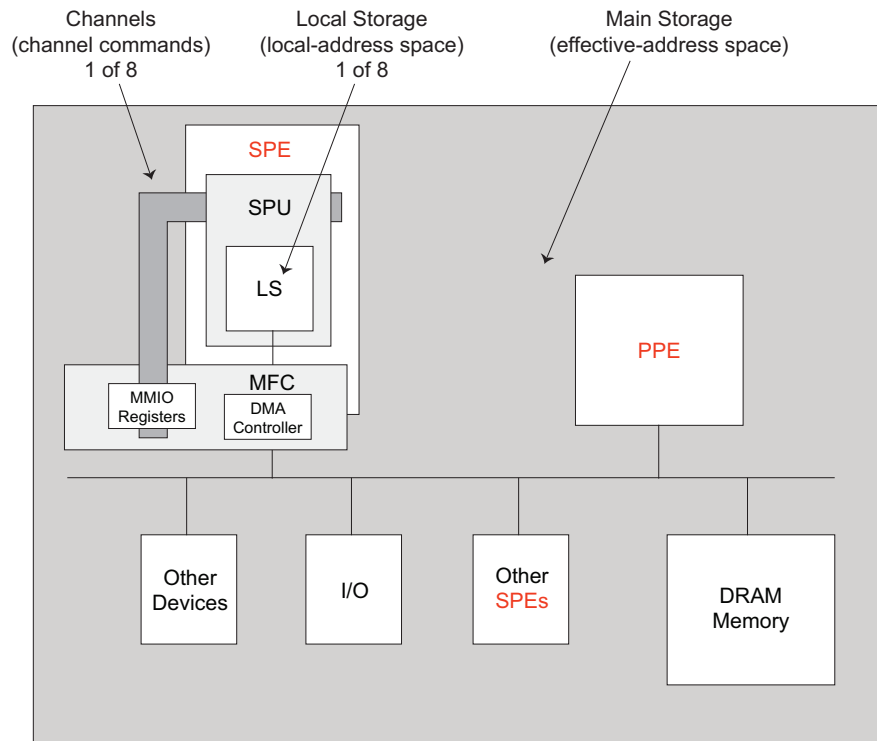


Figure 13. Storage domains defined in the Cell Broadband Engine

An SPE can only fetch instructions from its own LS, and loads and stores can only access the LS. An SPE or PPE performs data transfers between the SPE's LS and main storage primarily using DMA transfers controlled by the MFC DMA controller for that SPE. Software on the SPE's SPU interacts with the MFC through channels, which enqueue DMA commands and provide other facilities, such as mailboxes, signal notification, and access auxiliary resources.

An SPE program references its own LS using a Local Store Address (LSA). The LS of each SPE is also assigned a Real Address (RA) range within the system's memory map. This allows privileged software to map LS areas into the effective address (EA) space, where the PPE, other SPEs, and other devices that generate EAs can access the LS.

Each SPE's MFC serves as a data-transfer engine. DMA transfer requests contain both an LSA and an EA. Thus, they can address both an SPE's LS and main storage and thereby initiate DMA transfers between the domains. The MFC accomplishes this by maintaining and processing an MFC command queue. DMA requests can be sent to an MFC either by software on its associated SPU or on the PPE, or by any other processing device that has access to the MFC's MMIO problem-state registers.



The queued requests are converted into DMA transfers. Each MFC can maintain and process multiple in-progress DMA command requests and DMA transfers. The MFC can also autonomously manage a sequence of DMA transfers in response to a DMA-list command from its associated SPU. Each DMA command is tagged with a 5-bit Tag Group ID. Software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups.

The MFC supports naturally aligned transfer sizes of 1, 2, 4, or 8 bytes, and multiples of 16-bytes, with a maximum transfer size of 16 KB. Peak performance can be achieved for transfers when both the EA and LSA are 128-byte aligned and the size of the transfer is a multiple of 128 bytes.

Each MFC has an associated memory management unit (MMU) that holds and processes address-translation and access-permission information supplied by the PPE operating system. This MMU is distinct from the one used by the PPE. To process an effective address provided by a DMA command, the MMU uses the same method as the PPE memory-management functions. Thus, DMA transfers are coherent with respect to system storage. Attributes of system storage are governed by the page and segment tables of the PowerPC Architecture.

The PPE or other processing devices can initiate MFC commands on a particular MFC by accessing its *MFC Command-Parameter Registers*, shown in Table 7. These registers are mapped to the system's real-address space. The PPE performs MMIO reads and writes to access these registers. The registers are contained in each SPE's memory region, and DMA command requests are made by writing parameters to the registers.

*Table 7. MFC command-parameter registers for PPE-initiated DMA transfers*

Name	Mnemonic	Max. Entries	R/W	Width (bits)
<i>MFC Local-Storage Address</i>	MFC_LSA	1	W	32
<i>MFC Effective Address High</i>	MFC_EAH	1	W	32
<i>MFC Effective Address Low</i>	MFC_EAL	1	W	32
<i>MFC Transfer Size</i>	MFC_Size	1	W	32
<i>MFC Command Tag Identification</i>	MFC_TagID			
<i>MFC Class ID and Command Opcode</i>	MFC_ClassID_CMD	8	W	32
<i>MFC Command Status</i>	MFC_CMDStatus	1	R	32

**Note:** The MFC\_EAH and MFC\_EAL can be written in a single 64-bit store. Similarly, MFC\_Size, MFC\_TagID, and MFC\_ClassID\_CMD can also be written in a single 64-bit store.

## Issuing DMA commands from the PPE

To enqueue a DMA command from the PPE, access the MFC Command-Parameter Registers in this sequence:

1. Write the LS address to the MFC\_LSA register.
2. Write the effective address high and low parts to the MFC\_EAH and MFC\_EAL registers.
3. Write the transfer size and tag ID to the MFC\_Size and MFC\_TagID registers.
4. Write the class ID and command opcode to the MFC\_ClassID\_CMD registers.

5. Read the MFC\_CMDStatus register to determine the success or failure of the attempt to enqueue a DMA command.

The least-significant 2 bits of the command status value returned from the read of the MFC\_CMDStatus register indicate the success or error of the attempt to enqueue a DMA. The values of these two bits have the following meanings:

- 0 – Indicates that the enqueue was successful.
- 1 – Indicates that a sequence error occurred while enqueueing the DMA. For example, an interrupt occurred, then another DMA was started within an interrupt handler. In this case, the DMA enqueue sequence must be restarted at step 1.
- 2 – Indicates that the enqueue failed due to insufficient space in the command queue.
- 3 – Indicates that both errors occurred.

In the case of insufficient space, software could wait for space to become available before attempting the DMA transfer again, or software could simply continue attempting to enqueue the DMA until successful.

## Creating threads for the SPEs

Programs to be run on an SPE are most often written in C or C++ (or assembly language) and can use the SPE data types and intrinsics defined in the *C/C++ Language Extensions for Cell Broadband Engine Architecture*.

The SPU C/C++ Language Extensions are described in “SPU C/C++ language extensions (intrinsics)” on page 64. The SPE code modules must be written and compiled separately from the PPE code modules, using different compilers. A PPE module starts an SPE module running by creating a thread on the SPE, using the `spe_context_create`, `spe_program_load`, and `spe_context_run` library calls, provided in the SPE runtime management library.

The `spe_context_create` call creates a context for the SPE thread which contains the persistent information about a logical SPE. This information should not be accessed directly by the application. The signature and parameter synopsis for the `spe_create_thread` library call is:

```
spe_context_ptr_t spe_context_create(unsigned int flags,  
    spe_gang_context_ptr_t gang)
```

- `flags` – This is a bit-wise OR of modifiers that is applied when the new context is created. The following values are accepted:
  - 0 – No modifiers are applied.
  - `SPE_EVENTS_ENABLE` – Configure the context with event handling enabled.
  - `SPE_CFG_SIGNOTIFY1_OR` – Configure the SPU Signal Notification 1 Register to be in “logical OR” mode instead of the default “Overwrite” mode.
  - `SPE_CFG_SIGNOTIFY2_OR` – Configure the SPU Signal Notification 2 Register to be in “logical OR” mode instead of the default “Overwrite” mode.
  - `SPE_MAP_PS` – Request permission for memory-mapped access to the SPE thread’s problem state area.
  - `SPE_ISOLATE` – Specifies that the SPE will execute in the isolation mode.
  - `SPE_ISOLATED_EMULATE` – Specifies that the SPE will execute in an emulated isolation mode.
- `gang` – Collection of contexts in which the context being created should be a part of.

Before being able to run an SPE context, an SPE program has to be loaded into the context using the `spe_program_load` subroutine. The signature and parameter synopsis for the `spe_program_load` library call is:

```
int spe_program_load(spe_context_ptr spe, spe_program_handle_t *program)
```

- `spe` – The SPE context in which in specified program is to be loaded.
- `program` – Indicates the program to be loaded into the SPE context. This is an opaque pointer to an SPE Executable and Linking Format (ELF) image that has already been loaded and mapped into system memory. This pointer is normally provided as a symbol reference to an SPE ELF executable image that has been embedded into a PPE ELF object and linked with the calling PPE program. This pointer can also be established dynamically by loading a shared library containing an embedded SPE ELF executable, using `dlopen(2)` and `dlsym(2)`, or by using the `spe_image_open` function to load and map a raw SPE ELF executable.

An SPE context is executed on a physical SPE by calling the `spe_context_run` function. This subroutine causes the current PPE thread to transition to a SPE thread by passing its execution control from the PPE to the SPE whose context is scheduled to run on. The PPE resumes execution when the SPE stops.

**Note:** In order to achieve multiple threads of execution (PPE and SPE threads), separate “`pthread`” must be created for each thread of execution using `pthread_create`. An example is provided in “Producing a simple multi-threaded CBE program” on page 42.

The signature and parameter synopsis for the `spe_context_run` library call is:

```
int spe_context_run(spe_context_ptr_t spe, unsigned int *entry,
    unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)
```

- `spe` — The context to be run.
- `entry` — Pointer initial value of the instruction pointer in which the SPE program should start executing. If the value pointed to by `entry` is `SPE_DEFAULT_ENTRY`, the default entry for the main program obtained from loaded SPE image will be used. Upon return from the `spe_context_run` call, the value pointed to by `entry` contains the next instruction to be executed upon resumption of the program.
- `runflags` — This is a bit-wise OR of modifiers which request specific behavior when the SPE context is run. Flags include:
  - `0` — Default behavior. No modifiers are applied.
  - `SPE_RUN_USER_REGS` — Specifies that the SPE setup registers, `r3`, `r4`, and `r5`, are initialized with the 48 bytes pointed to by `argp`.
  - `SPE_NO_CALLBACKS` — Specifies that register SPE library callbacks should not be automatically executed. This includes “PPE-assisted library calls” that are provided by the SPE Runtime library.
- `argp` — An optional pointer to application specific data. It is passed as the second parameter of the SPU program.
- `envp` — An optional pointer to environment specific data. It is passed as the third parameter of the SPU program.
- `stopinfo` — An optional pointer to a structure of type `spe_stop_info_t` that provides information as to the reason why the SPE stopped running. See library documentation for more details on this structure.

The following code sample shows PPE code creating a SPE context, loading a SPE program into the context and running the program from the current thread.

```

#include <libspe2.h>
extern spe_program_handle_t spe_code;
...
spe_context_ptr_t ctx;
unsigned int entry = SPE_DEFAULT_ENTRY;

if ((ctx = spe_context_create(0, NULL)) == NULL) {
    perror("Failed creating SPE context");
    exit(1);
}
if (spe_program_load(ctx, &spe_code)) {
    perror("Failed loading program");
    exit(1);
}
if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
    perror("Failed running context");
    exit(1);
}

```

## Communication between the PPE and SPEs

The PPE communicates with the SPEs through privileged-state and problem-state MMIO registers supported by the MFC of each SPE.

These registers are accessed by the associated SPE through its channel mechanism (see “Channels” on page 55), which consist of unidirectional registers and queues and support logic. The three primary communication mechanisms between the PPE and SPEs are mailboxes, signal notification registers, and DM)

Mailboxes are queues for exchanging 32-bit messages. Two mailboxes (the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox) are provided for sending messages from the SPE to the PPE. One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to the SPE. Table 8 lists the mailbox channels and their associated MMIO registers.

**Note:** Mailboxes can also be used as a communications mechanism between SPEs. This is accomplished by an SPE DMAing data into the other SPE’s mailbox using the effective addressed problem state mapping.

Table 8. Mailbox channels and MMIO registers

Name	Channel				MMIO Register			
	Mnemonic	Max. entries	R/W	Width (bits)	Mnemonic	Max. entries	R/W	Width (bits)
SPU Write Outbound Mailbox	SPU_WrOutMbox	1	W	32	SPU_Out_Mbox	1	R	32
SPU Read Inbound Mailbox	SPU_RdInMbox	4	R	32	SPU_In_Mbox	4	W	32
SPU Write Outbound Interrupt Mailbox	SPU_WrOutIntrMbox	1	W	32	SPU_Out_Intr_Mbox	1	R	32

SPU signal-notification channels are inbound (to an SPE) 32-bit registers. They can be configured for one-to-one signaling or many-to-one signaling. An SPE read of one of its two signal-notification channels clears the channel. A PPE MMIO read does not clear the channel. Table 9 on page 41 lists the signal-notification channels and associated MMIO registers.

Table 9. Signal notification channels and MMIO registers

Name	Channel				MMIO Register			
	Mnemonic	Max. entries	R/W	Width (bits)	Mnemonic	Max. entries	R/W	Width (bits)
<i>SPU Signal Notification 1</i>	SPU_RdSigNotify1	1	R	32	SPU_Sig_Notify_1	1	R/W	32
<i>SPU Signal Notification 2</i>	SPU_RdSigNotify2	1	R	32	SPU_Sig_Notify_2	1	R/W	32

The PPE is often used as an application controller, managing and distributing work to the SPEs. A large part of this task is loading main storage with the data to be processed, and then notifying the SPE by either writing to the SPU Read Inbound Mailbox or writing to one of the SPE's signal notification registers.

Mailboxes are also useful when the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMAs to complete, and then writes to an SPU Write Outbound Mailbox to notify the PPE that its computation is complete. The PPE can use either a mailbox or a signal to let an SPE know that the PPE has placed computational results in main storage via DMA.

---

## Developing code for the Cell Broadband Engine

There can be several types of programs, including PPE programs, SPE programs, and Cell Broadband Engine programs (PPE programs with embedded SPE programs).

The PPE and SPE programs use different compilers. The correct compiler, compiler flags, and libraries must be used for the intended processor and program type. The PPE typically sets up, starts, and stops an SPE. Communication between the PPE and SPEs is an important consideration.

To aid in simplifying the process of producing programs for the Cell Broadband Engine, the SDK's Samples (see "The software development kit" on page 16) provides a build environment based upon the make utility. For additional details on the SDK's build environment, consult the file `README_build_env.txt` located in `/opt/cell/sdk/buildutils`.

Programmers can declare the types of programs in the makefile, and the correct compiler, compiler options, and libraries will be used for the build. The most important target types are `PROGRAM_ppu` and `PROGRAM_spu`, for building PPE programs and SPE programs, respectively. To use makefile definitions supplied by the SDK for producing programs, include the following line at the bottom of the makefile:

```
include ../../../../buildutils/make.footer
```

Insert as many instances of `../` as necessary to reach the top of the directory tree where `buildutils` resides. Alternatively, `make.footer` can be sourced directly (useful when working on projects within the Eclipse IDE framework), by defining `CELL_TOP` environment variable and sourcing the `make.footer` as follows:

```
include $(CELL_TOP)/buildutils/make.footer
```

The makefiles in the SDK Samples support both methods of importing the `make.footer`.

Figure 14 shows a sample directory structure and makefiles for a system with a PPE program and an SPE program. This sample project `sampleproj` has a project directory and two subdirectories. The `ppu` directory contains the source code and makefile for the PPE program. The `spu` directory has the source code and makefile for the SPE program. The makefile in the project directory executes the makefiles in the two subdirectories. This is only one of the possible project directory structures.

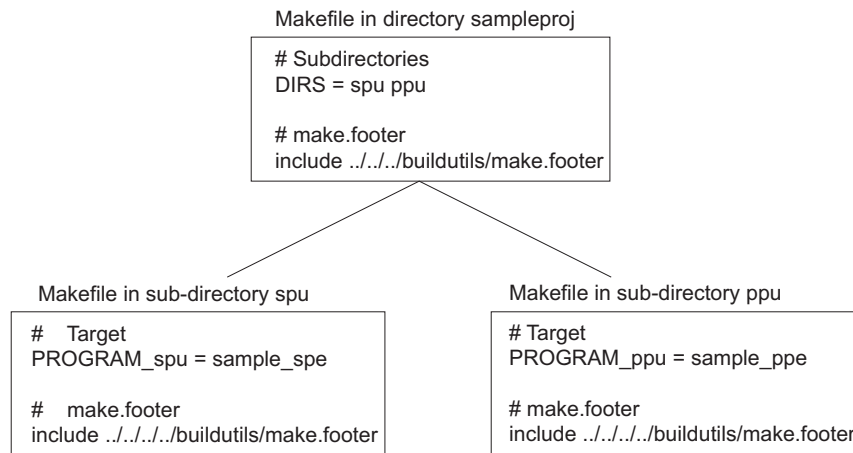


Figure 14. Sample project directory structure and makefiles

## Producing a simple multi-threaded CBE program

To produce a simple program for the CBE, you should follow the steps listed below (this example is included in the SDK in `/opt/cell/sdk/src/tutorial/simple`).

The project is called `simple`. For this example, the PPE code will be built in the project directory, instead of a `ppu` sub-directory.

1. Create a directory named `simple`.
2. In directory `simple`, create a file named `Makefile` using the following code:

```

#####
# Subdirectories
#####

DIRS := spu

#####
# Target
#####

PROGRAM_ppu := simple

#####
# Local Defines
#####

```

```

IMPORTS          := spu/lib_simple_spu.a -lspe2 -lpthread
# imports the embedded simple_spu library
# allows consolidation of spu program into ppe binary

#####
# make.footer
#####

# make.footer is in the top of the SDK
#ifdef CELL_TOP
include $(CELL_TOP)/builddutils/make.footer
else
include ../../../../builddutils/make.footer
endif

```

3. In directory simple, create a file simple.c using the following code:

```

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t simple_spu;

#define MAX_SPU_THREADS 16

void *ppu_thread_function(void *arg) {
    spe_context_ptr_t ctx;
    unsigned int entry = SPE_DEFAULT_ENTRY;

    ctx = *((spe_context_ptr_t *)arg);
    if (spe_context_run(ctx,&entry, 0, NULL, NULL, NULL) < 0) {
        perror("Failed running context");
        exit (1);
    }
    pthread_exit(NULL);
}

int main()
{
    int i,spu_threads;
    spe_context_ptr_t ctxs[MAX_SPU_THREADS];
    pthread_t threads[MAX_SPU_THREADS];

    /* Determine the number of SPE threads to create */
    spu_threads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
    if (spu_threads > MAX_SPU_THREADS) spu_threads = MAX_SPU_THREADS;

    /* Create several SPE-threads to execute 'simple_spu' */
    for(i=0; i<spu_threads; i++) {
        /* Create context */
        if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
            perror ("Failed creating context");
            exit (1);
        }
        /* Load program into context */
        if (spe_program_load (ctxs[i],&simple_spu)) {
            perror ("Failed loading program");
            exit (1);
        }
        /* Create thread for each SPE context */
        if (pthread_create (&threads[i], NULL,&ppu_thread_function,&ctxs[i])) {
            perror ("Failed creating thread");
            exit (1);
        }
    }
}

```



```

/* Wait for SPU-thread to complete execution. */
for (i=0; i<spu_threads; i++) {
    if (pthread_join (threads[i], NULL)) {
        perror("Failed pthread_join");
        exit (1);
    }
}

printf("\nThe program has successfully executed.\n");

return (0);
}

```

4. Create a directory named spu.
5. In the directory spu, create a file named Makefile using the following code:

```

#####
# Target
#####

PROGRAMS_spu    := simple_spu

# created embedded library
LIBRARY_embed   := lib_simple_spu.a

#####
# make.footer
#####

# make.footer is in the top of the SDK
ifdef CELL_TOP
    include $(CELL_TOP)/builddutils/make.footer
else
    include ../../../../builddutils/make.footer
endif

```

6. In the same directory, create a file simple\_spu.c using the following code:

```

#include <stdio.h>

int main(unsigned long long id)
{

    /* The first parameter of an spu program will always be the spe_id of the spe
     * thread that issued it.
     */
    printf("Hello Cell (0x%llx)\n", id);

    return 0;
}

```

7. Compile the program by entering the following command at the command line while in the simple directory:

```
make
```

This CBE program then creates SPE threads that output “Hello Cell (#)\n” to the systemsim output window, where # is the spe\_id of the SPE thread that issued the print.

## Running the program in the simulator

Now that we have compiled the program, it can now be executed either on a CBE system or a simulation of a CBE system. In this case, we will use the *IBM Full System Simulator* for the Cell Broadband Engine as a simulation of a CBE system by starting the simulator, importing the program, and executing it.



To start the IBM Full System Simulator for the Cell Broadband Engine with a graphics user interface:

1. Create a private, non-root, simulator execution environment.

```
mkdir sim
cd sim
cp /opt/ibm/systemsim-cell/run/cell/linux/.systemsim.tcl .
export PATH=/opt/ibm/systemsim-cell/bin:$PATH
```

2. Start the simulator with a graphical user interface:

```
systemsim -g
```

3. Two new windows will appear on the screen. The first is a command-line/console window labeled `mysim` in the window's title bar. The second is the simulator graphical user interface (GUI) window. These windows are shown in Figure 15.

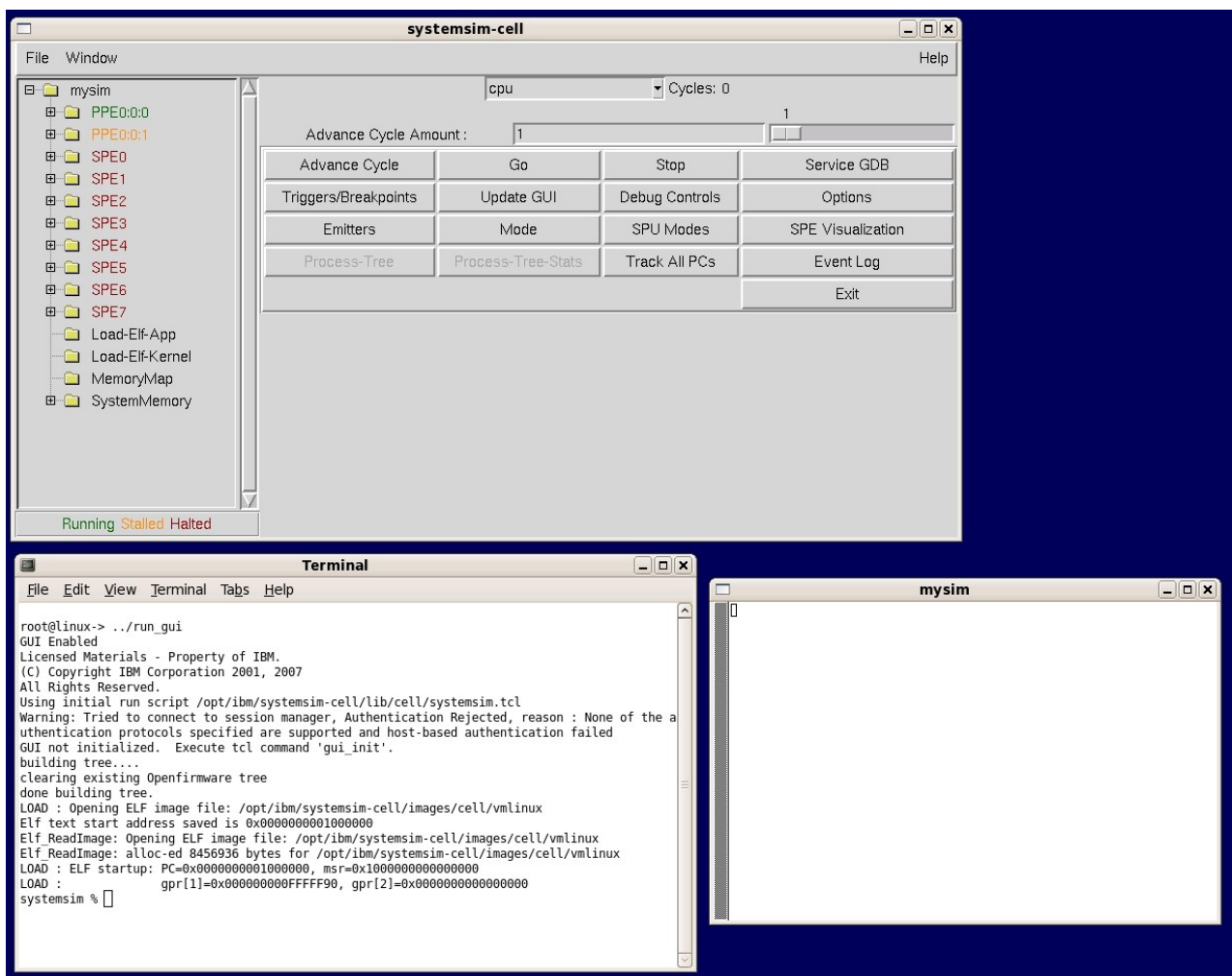
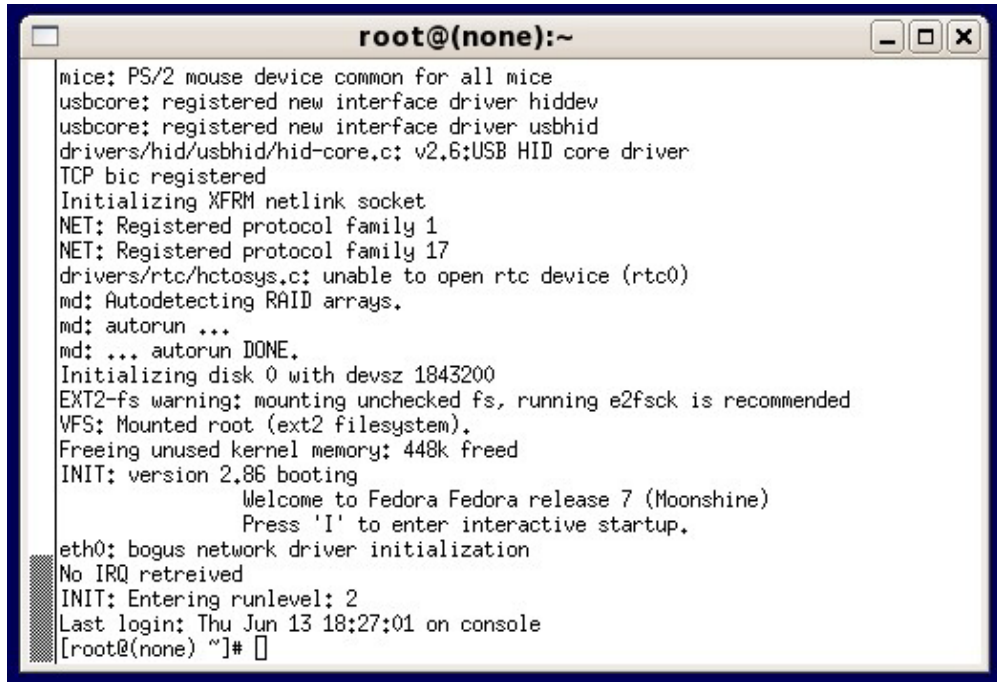


Figure 15. Windows visible after starting the simulator GUI

The window labeled `mysim` is an uart window that, when Linux boots, it becomes a Linux console window. When the console window first appears, it is empty and there is no user prompt, because Linux has not yet been booted on the simulated system.

The window in which the simulator was started (`systemsim -g`) is the simulator command-line window.

4. Boot the Linux operating system on the simulator by clicking the Go button on the graphical user interface (GUI). To make the simulator run “quickly”, click the Fast Mode button prior to Go. This forces the simulator to bypass its standard analysis and statistic collection features. The console window will begin to display the Linux booting process. When Linux has finished booting on the simulator, a command prompt will be visible in the window. Figure 16 shows the window on completion of the boot process.



```
root@(none):~
mice: PS/2 mouse device common for all mice
usbcore: registered new interface driver hiddev
usbcore: registered new interface driver usbhid
drivers/hid/usbhid/hid-core.c: v2.6:USB HID core driver
TCP bic registered
Initializing XFRM netlink socket
NET: Registered protocol family 1
NET: Registered protocol family 17
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
md: Autodetecting RAID arrays.
md: autorun ...
md: ... autorun DONE.
Initializing disk 0 with devsz 1843200
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 448k freed
INIT: version 2.86 booting
           Welcome to Fedora Fedora release 7 (Moonshine)
           Press 'I' to enter interactive startup.
eth0: bogus network driver initialization
No IRQ retrieved
INIT: Entering runlevel: 2
Last login: Thu Jun 13 18:27:01 on console
[root@(none) ~]#
```

Figure 16. Console window on completion of Linux boot

The simulator is now ready to import the sample program into its environment. Before doing that, however, you can confirm that the program is not in the simulator environment, by entering the `ls` command at the prompt in the console window, and observing that `simple` is not listed in the directory listing.

5. Import the program from the base simulator hosting environment into the simulator environment by entering the following command:

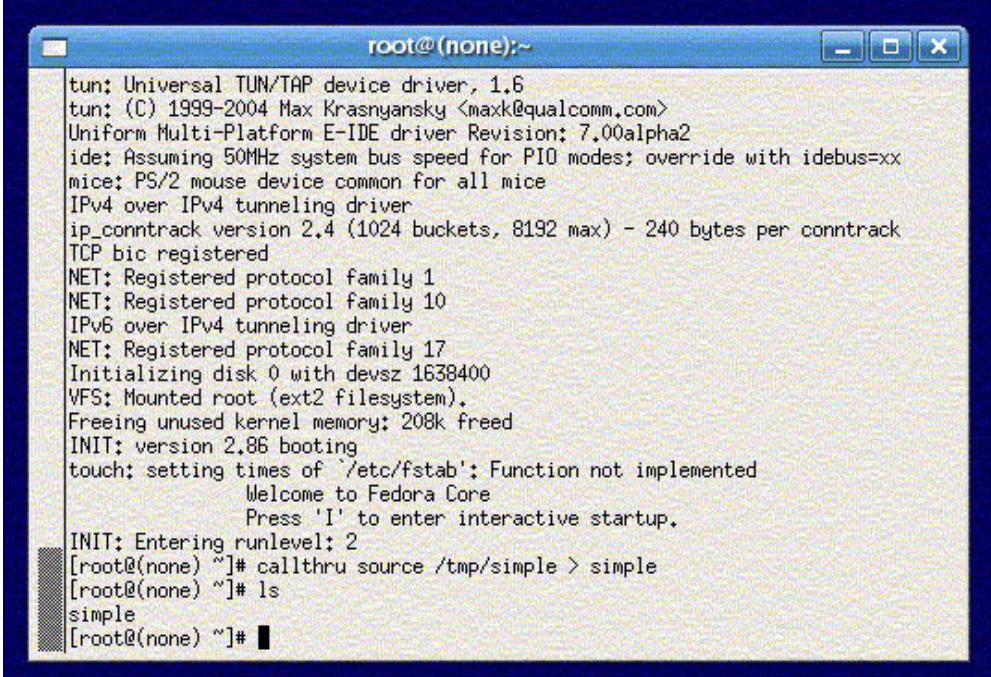
```
callthru source /tmp/simple > simple
```

This command tells the simulator environment to “call through” to the simulator hosting environment’s `/tmp` directory, retrieve the file called `simple`, and copy that file to the simulator file system. If you now enter an `ls` command in the console window, you will see `simple` listed in the current directory. Figure 17 on page 47 shows the process of loading the program into the simulation environment.

Alternatively, one can permanently add or delete files to the `sysroot` disk image by performing a loop device mount the `sysroot` disk image and copying or removing files from the mounted image, prior to booting the simulation environment. For example, the following sequence:

```
mount -o loop /opt/ibm/systemsim-cell/image/cell/sysroot_disk /mnt
cp /tmp/simple /mnt/simple
umount /mnt
```

copies the simple executable from the host system's /tmp directory to the sysroot's / directory.



```
root@(none):~
tun: Universal TUN/TAP device driver, 1.6
tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
Uniform Multi-Platform E-IDE driver Revision: 7.00alpha2
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
mice: PS/2 mouse device common for all mice
IPv4 over IPv4 tunneling driver
ip_contrack version 2.4 (1024 buckets, 8192 max) - 240 bytes per conntrack
TCP bic registered
NET: Registered protocol family 1
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
Initializing disk 0 with devsz 1638400
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 208k freed
INIT: version 2.86 booting
touch: setting times of `/etc/fstab': Function not implemented
      Welcome to Fedora Core
      Press 'I' to enter interactive startup.
INIT: Entering runlevel: 2
[root@(none) ~]# callthru source /tmp/simple > simple
[root@(none) ~]# ls
simple
[root@(none) ~]# █
```

Figure 17. Loading the program into the simulation environment

Even though the file had execute permissions in the base simulator hosting environment, the newly imported file in the emulator environment does not.

6. Add execute permissions to the program file `simple` by issuing the following command:

```
chmod +x simple
```

7. Execute the program by issuing the following command:

```
./simple
```

The output of the program will appear in the console window. Figure 18 on page 48 shows the output of running the sample program.



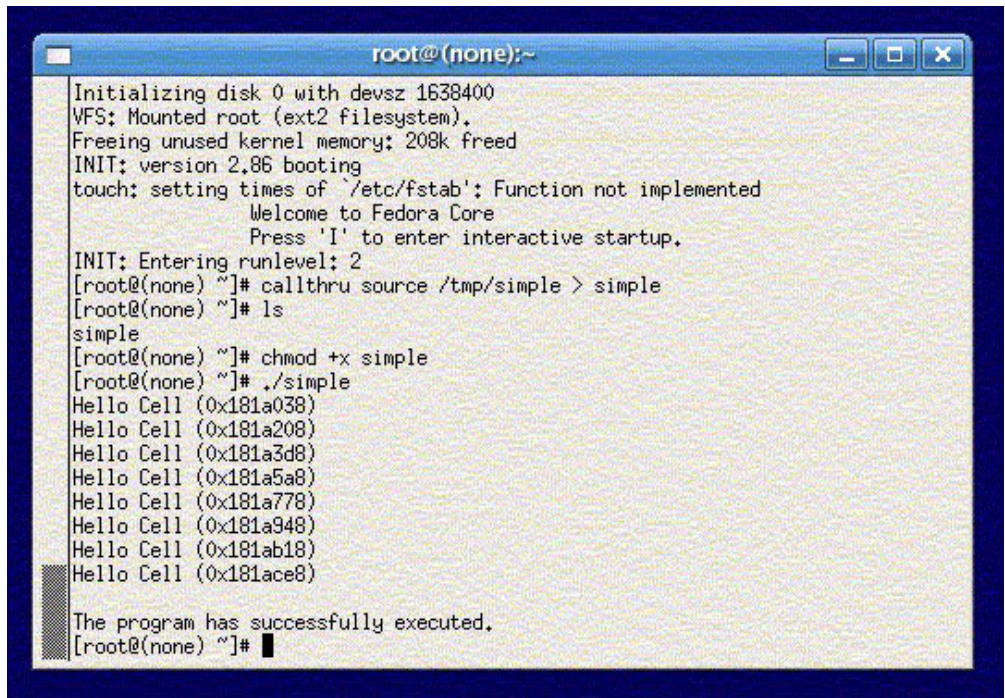


Figure 18. Running the sample program

## Debugging programs

Debugging a program is often the most challenging part of programming, especially with multithreaded programs. The SDK contains several tools for debugging, the most important of which are the gdb debugger and the *IBM Full System Simulator for the Cell Broadband Engine*.

The gdb debugger is a command-line debugger available as part of the GNU development environment. Because of the Cell Broadband Engine's unique characteristics, gdb has been modified so that there are actually two versions of the debugger – ppu-gdb for debugging PPE and combined PPE and SPE programs, and spu-gdb for debugging SPE programs. For additional information on using ppu-gdb and spu-gdb, consult the *Software Development Kit, Programmer's Guide*.

The other tool for debugging a Cell Broadband Engine program is the *IBM Full System Simulator for the Cell Broadband Engine*. This simulator lets you view many aspects of the simulated running program in GUI mode. You can also control many aspects of the simulator using Tcl commands. The simulator is described more fully in Chapter 5, "The simulator," on page 129.

---

## Chapter 3. Programming the SPEs

The eight identical Synergistic Processor Elements (SPEs) are optimized for compute-intensive applications in which a program's data and instruction needs can be anticipated and transferred into the local store (LS) by DMA while the SPE computes using previously transferred data and instructions.

The streaming data sets in 3D graphics, media, and broadband communications are examples of applications that run well on SPEs. However, the SPEs are not optimized for running programs that have significant branching, such as an operating system. Each SPE supports only a single program context at any one time. Typically, the operating system runs on the PPE, and user-mode threads are executed on the SPEs.

The SPEs achieve high performance, in part, by eliminating the overhead of load and store address translation, hardware-managed caches, out-of-order instruction issue, and branch prediction. Instead, the SPEs capitalize on the high computational efficiencies that can be obtained for streaming-data applications by providing a large (128-entry by 128-bit) unified register file, dual-instruction issue, and high DMA bandwidth between the LS and main storage.

Each SPE supports the single-instruction, multiple-data (SIMD) instruction architecture, described in the *SPU Instruction Set Architecture*. Although details of this instruction set are given in the sections that follow, an SPE is normally programmed in a high-level language like C or C++. The SPU instruction set is supported by a rich set of language extensions for C/C++, described in the *C/C++ Language Extensions for Cell Broadband Engine Architecture specification*. These extensions define SIMD data types and intrinsics (commands, in the form of function calls) that map to one or more assembly-language instructions, giving programmers very convenient and productive control over code performance without the need for assembly-language programming.

---

### SPE configuration

This section describes the main components of a Synergistic Processor Element (SPE).

The main components are shown in Figure 19 on page 50. Their functions include:

- *Synergistic Processor Unit (SPU)* — The SPU executes SPU instructions fetched from its 256-KB LS. The SPU fills its LS with instructions and data using DMA transfers initiated from SPU or PPE software.
- *Memory Flow Controller (MFC)* — The MFC provides the interface, by means of the Element Interconnect bus (EIB), between the SPU and main storage. The MFC performs DMA transfers between the SPU's LS and main storage, and it supports mailbox and signal-notification messaging between the SPE and the PPE and other devices. The SPU communicates with its MFC through SPU channels. The PPE and other devices (including other SPEs) communicate with an MFC through memory-mapped I/O (MMIO) registers associated with the SPU's channels.

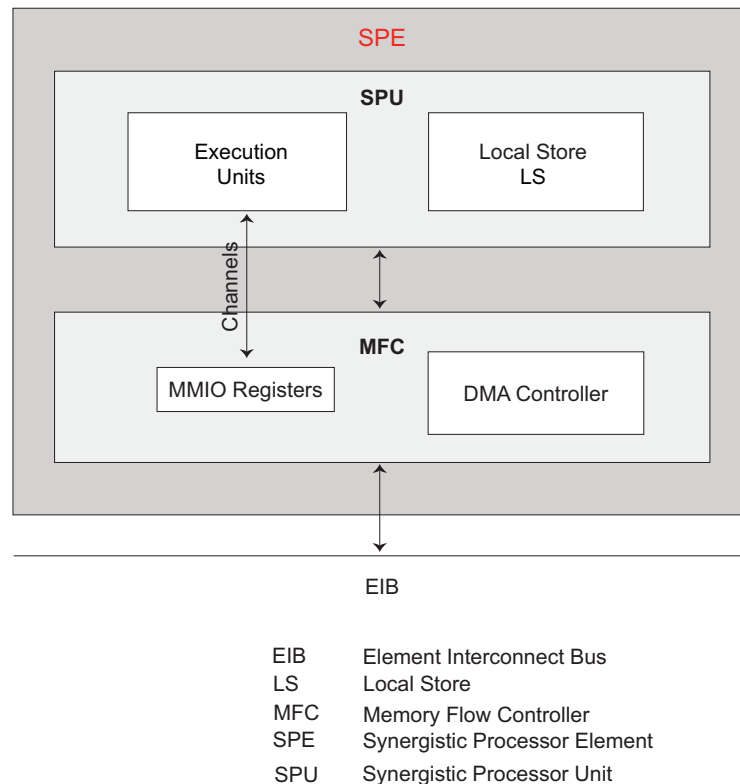


Figure 19. SPE architectural block diagram

## Synergistic Processor Unit

Each of the eight SPEs is an independent processor with its own program counter, register file, and 256-KB LS.

An SPE operates directly on instructions and data in its LS. It fills its LS by requesting DMA transfers from its MFC, which manages the DMA transfers. The SPU has specialized units for executing load and store, fixed-point, floating-point unit (single-precision and double-precision), and channel-interface instructions.

The large 128-entry, 128-bit wide register file, and its flat architecture (all operand types stored in a single register file), allows for instruction-latency hiding without speculation. The register file is unified—meaning that all data types (integer, single-precision and double-precision floating-point, scalars, vectors, logicals, bytes, and others) use the same register file. The register file also stores return addresses, results of comparisons, and so forth. As a consequence of the large, unified register file, expensive hardware techniques such as out-of-order processing or deep speculation are not needed to achieve high performance.

LS addresses can be aliased by PPE privileged software onto the main-storage (effective-address) space. DMA transfers between the LS and main storage are coherent in the system. A pointer to a data structure created on the PPE can be passed to an SPU, and the SPU can use this pointer to issue a DMA command to

bring the data structure into its LS. PPE software can use locking instructions and mailboxes for synchronization and mutual exclusion.

The SPU architecture has the following restrictions:

- No direct (SPU-program addressable) access to main storage. The SPU accesses main storage only by using the MFC's DMA transfers.
- No direct access to system control, such as page-table entries. PPE privileged software provides the SPU with the address-translation information that its MFC needs.
- With respect to accesses by its SPU, the LS is unprotected and un-translated storage.

## SPE registers

This section describes the Synergistic Processor Element (SPE) user registers.

The complete set of SPE user registers is shown in Figure 20. All computational instructions operate only on registers—there are no computational instructions that modify storage. The SPE registers include:

- *General-Purpose Registers (GPRs)* — All data types can be stored in the 128-bit GPRs, of which there are 128.
- *Floating-Point Status and Control Register (FPSCR)* — The processor updates the 128-bit FPSCR after every floating-point operation to record information about the result and any associated exceptions.

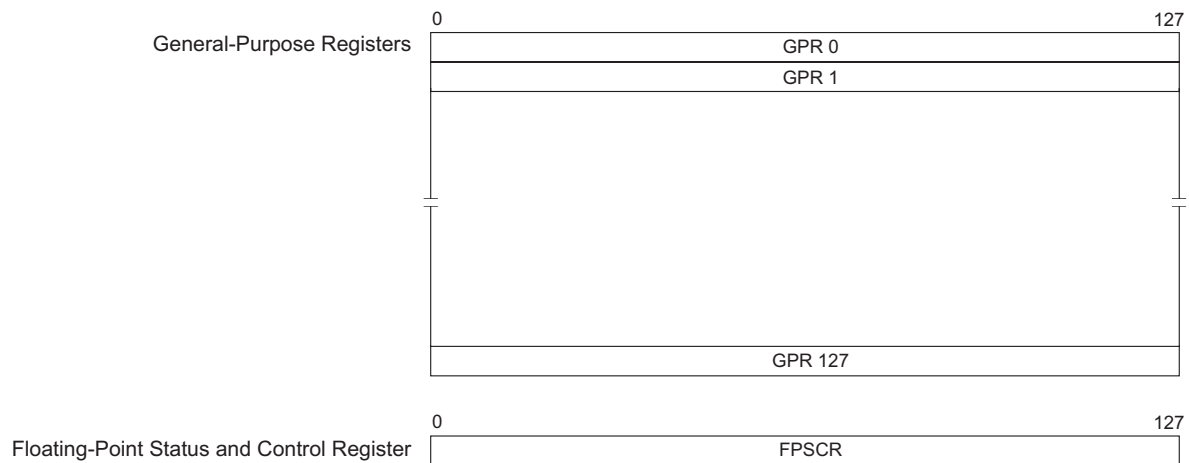


Figure 20. SPE user-register set

## Floating-point operations

The SPU executes both single-precision and double-precision floating-point operations. Single-precision instructions are performed in 4-way SIMD fashion, fully pipelined, whereas double-precision instructions are partially pipelined.

The data formats for single-precision and double-precision instructions are those defined by IEEE Standard 754, but the results calculated by single-precision instructions are not fully compliant with IEEE Standard 754.

For single-precision operations, the range of normalized numbers is extended beyond the IEEE standard. The representable, nonzero numbers range from  $X_{\min} = 2^{126}$  to  $X_{\max} = (2^{-23})2^{128}$ . If the exact result overflows (that is, if it is larger in magnitude than  $X_{\max}$ ), the rounded result is set to  $X_{\max}$  with the appropriate sign. If the exact result underflows (that is, if it is smaller in magnitude than  $X_{\min}$ ), the rounded result is forced to zero. A zero result is always a positive zero.

Single-precision floating-point operations implement IEEE 754 arithmetic with the following changes:

- Only one rounding mode is supported: round towards zero, also known as truncation.
- Denormal operands are treated as zero, and denormal results are forced to zero.
- Numbers with an exponent of all ones are interpreted as normalized numbers and not as infinity or not-a-number (NaN).

Double-precision operations do not support the IEEE precise trap (exception) mode. If a double-precision denormal or not-a-number (NaN) result does not conform to IEEE Standard 754, then the deviation is recorded in a sticky bit in the FPSCR register, which can be accessed using the `fscrdd` and `fscrwr` instructions or the `spu_mffpscr` and `spu_mtfpscr` intrinsics.

Double-precision instructions are performed as two double-precision operations in 2-way SIMD fashion. However, the SPU is capable of performing only one double-precision operation per cycle. Thus, the SPU executes double-precision instructions by breaking up the SIMD operands and executing the two operations in consecutive instruction slots in the pipeline. Although double-precision instructions have 13-clock-cycle latencies, only the final seven cycles are pipelined. No other instructions are dual-issued with double-precision instructions, and no instructions of any kind are issued for six cycles after a double-precision instruction is issued.

## Local Store

The *local store* (LS) can be regarded as a software-controlled cache that is filled and emptied by DMA transfers.

Key features of the LS include:

- Holds instructions and data
- 16-bytes-per-cycle load and store bandwidth, quadword aligned only
- 128-bytes-per-cycle DMA-transfer bandwidth
- 128-byte instruction prefetch per cycle

Competition might occur for access to the LS by:

- loads,
- stores,
- DMA reads,
- DMA writes,
- instruction fetches.

The SPU arbitrates access to the LS according the following priorities (with the highest priority first):

1. DMA reads and writes by the PPE or an I/O device.
2. SPU loads and stores.
3. Instruction prefetch.



Table 10 summarizes the LS-arbitration priorities and transfer sizes. DMA reads and writes always have highest priority. Because hardware supports 128-bit DMA reads and writes, these operations occupy, at most, one of every eight cycles (one of sixteen for DMA reads, and one of sixteen for DMA writes) to the LS. Thus, except for highly optimized code, the impact of DMA reads and writes on LS availability for loads, stores, and instruction fetches can be ignored.

*Table 10. LS-Access Arbitration Priority and Transfer Size*

Transaction	Transfer Size (Bytes)	Priority	Maximum Local Store Occupancy (SPU Cycle)	Access Path
MMIO	≤ 16	1-Highest	1/8	Line Interface
DMA	≤ 128	1		
DMA-List Transfer-Element Fetch	128	1	1/4	Quadword Interface
ECC Scrub	16	2	1/10	
SPU Load/Store	16	3	1	
Hint Fetch	128	3	1	Line Interface
Inline Fetch	128	4-Lowest	1/16 for inline code	

After DMA reads and writes, the next-highest user-initiated priority is given to load and store instructions. The rationale for doing so is that load and store instructions usually help a program's progress, whereas instruction fetches are often speculative. The SPE supports only 16-byte load and store operations that are 16-byte-aligned. It uses a second instruction (byte shuffle) to place bytes in a different order if, for example, the program requires only a 4-byte quantity or a quantity with a different data alignment. To store something that is not aligned, use a read-modify-write operation.

The lowest priority for LS access is given to instruction fetches, of which there are three types: flush-initiated fetches, inline prefetches, and hint fetches. Instruction fetches load 32 instructions per SPU request by accessing all banks of the LS simultaneously. Because the LS is single-ported, it is important that DMA and instruction-fetch activity transfer as much useful data as possible in each LS request.

### Pipelines and dual-issue rules

The SPU has two pipelines, named even (pipeline 0) and odd (pipeline 1). Into these pipelines, the SPU can issue and complete up to two instructions per cycle, one in each of the pipelines.

Whether an instruction goes to the even or odd pipeline depends on its instruction type, which is related to the execution unit that performs the function. Each execution unit is assigned to one of the two pipelines. Table 11 summarizes the instruction types, latencies, and pipeline assignments.

*Table 11. SPU Instruction Latency and Pipeline, by Instruction Class*

Instruction Class	Description	Latency (clock cycles)	Pipeline
LS	Load and store	6	Odd
HB	Branch hints	15	Odd

Table 11. SPU Instruction Latency and Pipeline, by Instruction Class (continued)

Instruction Class	Description	Latency (clock cycles)	Pipeline
BR	Branch resolution	4	Odd
CH	Channel interface, special-purpose registers	6	Odd
SP	Single-precision floating-point	6	Even
DP	Double-precision floating-point	13 <sup>2</sup>	Even
FI	Floating-point integer	7	Even
SH	Shuffle	4	Odd
FX	Simple fixed-point	2	Even
WS	Word rotate and shift	4	Even
BO	Byte operations	4	Even
NOP	No operation (execute)	-	Even
LNOP	No operation (load)	-	Odd

**Note:**

1. Inline or correctly hinted branches have zero-cycle delay. The mispredicted branch penalty is 18-19 clock cycles.
2. Six cycles of a double-precision floating-point operation are instruction-issue stalls. No instructions of any kind are issued for six cycles after a double-precision floating point instruction is issued.

The SPU issues all instructions in program order according to the pipeline assignment. Each instruction is part of a doubleword-aligned instruction pair called a *fetch group*.

A fetch group can have one or two valid instructions, but it must be aligned to doubleword boundaries. This means that the first instruction in the fetch group is from an even word address, and the second instruction from an odd word address.

The SPU processes fetch groups one at a time, continuing to the next fetch group when the current instruction group becomes empty. An instruction becomes issueable when register dependencies are satisfied and there is no structural hazard (resource conflict) with prior instructions or DMA or error-correcting code (ECC) activity.

Dual-issue occurs when a fetch group has two issueable instructions in which the first instruction can be executed on the even pipeline and the second instruction can be executed on the odd pipeline. If a fetch group cannot be dual-issued, but the first instruction can be issued, the first instruction is issued to the proper execution pipeline and the second instruction is held until it can be issued. A new fetch group is loaded after both instructions of the current fetch group are issued.

## Memory flow controller

The primary functions of the Memory Flow Controller (MFC) are to connect the SPU to the EIB and support DMA transfers between main storage and the LS.

Figure 19 on page 50 shows the functions of the MFC.

The MFC maintains and processes queues of DMA commands from its SPU or from the PPE or other devices. The MFC’s DMA controller (DMAC) executes the DMA commands. This allows the SPU to continue execution in parallel with the MFC’s DMA transfers.

The DMA and other MFC commands, and the command queues, are described in “MFC commands” on page 76.

To make DMA transfers between main storage and the LS possible, privileged software on the PPE provides the LS and MFC resources, such as memory-mapped I/O (MMIO) registers, with effective-address aliases in main storage. This enables software on the PPE or other SPUs and devices to access the MFC resources and control the SPU. Privileged software on the PPE also provides address-translation information to the MFC for use in DMA transfers. DMA transfers are coherent with respect to system storage. Attributes of system storage (address translation and protection) are governed by the page and segment tables of the PowerPC Architecture.

The MFC supports channels and associated MMIO registers for the purposes of enqueueing and monitoring DMA commands, monitoring SPU events, performing interprocessor-communication via mailboxes and signal-notification, accessing auxiliary resources such as the decremter (timer), and other functions.

In addition to supporting DMA transfers, channels, and MMIO registers, the MFC also supports bus-bandwidth reservation features and synchronizes operations between the SPU and other processing units in the system.

## Channels

*Channels* are unidirectional message-passing interfaces that support 32-bit messages and commands. Many of the channels provide communications between the SPE and its MFC, which in turn, mediates communication with the PPE and other devices.

Table 12 lists the channels and their attributes. Reserved and privileged channels are omitted.

Software on the SPU uses special *channel instructions* (shown in Table 13 on page 57) to read and write channel registers and queues.

Software on the PPE and other devices use load and store instructions to read and write to MFC’s MMIO registers that are associated with the SPU’s channels.

Table 12. SPE Channels

Channel	Name	Mnemonic	Size (bits)	R/W	Blocking
<b>SPU Events</b>					
0	SPU Read Event Status	SPU_RdEventStat	32	R	Yes
1	SPU Write Event Mask	SPU_WrEventMask	32	W	No
2	SPU Write Event Acknowledgment	SPU_WrEventAck	32	W	No
<b>SPU Signal Notification</b>					
3	SPU Signal Notification 1	SPU_RdSigNotify1	32	R	Yes
4	SPU Signal Notification 2	SPU_RdSigNotify2	32	R	Yes
<b>SPU Decrementer</b>					

Table 12. SPE Channels (continued)

Channel	Name	Mnemonic	Size (bits)	R/W	Blocking
7	SPU Write Decrementer	SPU_WrDec	32	W	No
8	SPU Read Decrementer	SPU_RdDec	32	R	No
<b>MFC Multisource Synchronization</b>					
9	MFC Write Multisource Synchronization Request	MFC_WrMSSyncReq	32	W	Yes
<b>SPU and MFC Read Mask</b>					
11	SPU Read Event Mask	SPU_RdEventMask	32	R	No
12	MFC Read Tag-Group Query Mask	MFC_RdTagMask	32	R	No
<b>SPU State Management</b>					
13	SPU Read Machine Status	SPU_RdMachStat	32	R	No
14	SPU Write State Save-and-Restore	SPU_WrSRR0	32	W	No
15	SPU Read State Save-and-Restore	SPU_RdSRR0	32	R	No
<b>MFC Command Parameters</b>					
16	MFC Local Store Address	MFC_LSA	32	W	No
17	MFC Effective Address High	MFC_EAH	32	W	No
18	MFC Effective Address Low or List Address	MFC_EAL	32	W	No
19	MFC Transfer Size or List Size	MFC_Size	16	W	No
20	MFC Command Tag Identification	MFC_TagID	16	W	No
21	MFC Command Opcode or ClassID	MFC_Cmd	32	W	Yes
<b>MFC Tag Status</b>					
22	MFC Write Tag-Group Query Mask	MFC_WrTagMask	32	W	No
23	MFC Write Tag Status Update Request	MFC_WrTagUpdate	32	W	Yes
24	MFC Read Tag-Group Status	MFC_RdTagStat	32	R	Yes
25	MFC Read List Stall-and-Notify Tag Status	MFC_RdListStallStat	32	R	Yes
26	MFC Write List Stall-and-Notify Tag Acknowledgement	MFC_WrListStallAck	32	W	No
27	MFC Read Atomic Command Status	MFC_RdAtomicStat	32	R	Yes
<b>SPU Mailboxes</b>					
28	SPU Write Outbound Mailbox	SPU_WrOutMbox	32	W	Yes
29	SPU Read Inbound Mailbox	SPU_RdInMbox	32	R	Yes
30	SPU Write Outbound Interrupt Mailbox	SPU_WrOutIntrMbox	32	W	Yes

Each channel has a corresponding count that indicates the remaining capacity (the maximum number of outstanding transfers) in that channel. This count is decremented when a channel instruction is issued to the channel, and the count increments when an action associated with that channel completes. Each channel is implemented with either blocking or non-blocking semantics.

Blocking channels cause the SPE to stall (suspend execution in a low-power state) when the SPE reads or writes a channel with a count of zero.

Key features of the SPE channel operations include:

- All transactions on the channel interface are unidirectional.
- Each channel transaction is independent of any other transaction.
- Sequential read and write transactions are supported.
- External access to control MMIO registers has higher priority than channel operations.
- Channel operations are done in program order.
- Channel read operations to reserved channels return zeros.
- Channel write operations to reserved channels have no effect.
- Reading of channel counts on reserved channels returns zero.

## Channel instructions

The *SPE Instruction Set Architecture* defines *three* channel instructions: *rdch*, *wrch*, and *rchcnt*.

A summary of the SPE Instruction Set Architecture is shown in “SPE instruction set” on page 60. The *rdch*, *wrch*, and *rchcnt* channel instructions are shown in Table 13.

Software running on an SPE uses the channel instructions to write parameters and enqueue the MFC commands, as described in “MFC commands” on page 76.

Table 13 includes both the SPE assembly-language instructions and their corresponding C-language intrinsics.

The intrinsics are described in “SPE C/C++ language extensions (intrinsics)” on page 64.

Table 13. SPE Channel Instructions

Instruction	Assembler Instruction	C-Language Intrinsic	Description
Read Channel	<i>rdch</i>	<i>spu_readch</i> <i>spu_readchqw</i>	Causes data to be read from the addressed channel and stored into the selected General-Purpose Register (GPR).
Write Channel	<i>wrch</i>	<i>spu_writch</i> <i>spu_writchqw</i>	Causes data to be read from the selected GPR and stored in the addressed channel
Read Channel Count	<i>rchcnt</i>	<i>spu_readchcnt</i>	Causes the count associated with the addressed channel to be stored in the selected GPR.

If the write channel is *nonblocking*, a *wrch* instruction can be issued regardless of the value of the channel count for that channel. If the write channel is *blocking*, then a *wrch* instruction that is issued when the count for that channel is equal to zero will stall the SPE. Stalling on a *wrch* instruction can be useful because it saves power, but to avoid stalling, software should first read the channel count to ensure that it is not zero before issuing a *wrch* instruction.

The method used to determine the channel count is dependent on the program. The program can poll the channel count for that register, using the *rchcnt*

instruction, or the program can issue a wrch instruction. If the program issues a wrch instruction, the SPE stalls, waiting until an acknowledgment is received from the write channel.

When an SPE program needs to receive information, it uses a rdch instruction. Usually, this information is held in an SPE register. The information can be loaded into this register through the channel interface using a read-data-load transaction.

- If the read channel is nonblocking, then a rdch instruction can be issued regardless of the value of the channel count for that channel.
- In the SPE, if the channel is a blocking channel, the SPE does not read from this register until the channel count for that register indicates that the data is valid (that is, when the count is greater than zero).
- If the count is zero, then there is no data in the channel and the SPE stalls until actions associated with that channel occur.

These actions can include the updating of the MFC\_RdTagStat channel (see Table 12 on page 55), the PPE writing data to the corresponding MMIO register (such as a mailbox channel), or other actions. The method used to determine the count depends on the program. The program can:

- poll the channel count for that register using the rchcnt instruction, or
- issue the rdch instruction.

If the program issues a rdch instruction, the SPE stalls, waiting until valid data is loaded.

The channel instructions are architected as 128 bits wide, but in the Cell Broadband Engine, channel instructions set use only the 32 bits from the preferred slot (the left-most word, word element 0) in the register.

## Mailboxes

Mailboxes are queues that support exchanges of 32-bit messages between an SPE and other devices. Each mailbox queue has an SPE channel assignment as well as a corresponding MMIO register assignment.

Two 1-entry mailbox queues are provided for sending messages from the SPE:

- SPU Write Outbound Mailbox
- SPU Write Outbound Interrupt Mailbox

One 4-entry mailbox queue is provided for sending messages to the SPE:

- SPU Read Inbound Mailbox

Each mailbox has an *SPE channel assignment* (see Table 12 on page 55) as well as a corresponding *MMIO register*. To access the mailbox, an SPE program uses rdch and wrch instructions (see Table 13 on page 57). The PPE and other processors use load and store instructions to access the corresponding MMIO addresses.

Data written by an SPE program to one of these mailboxes using a wrch instruction is available to any processor or device that reads the corresponding MMIO register. Data written by a device to the SPU Read Inbound Mailbox using an MMIO write is available to an SPE program by reading that mailbox using a rdch or rchcnt instruction. An MMIO read from either of the SPU Write Outbound Mailboxes, or a write to the SPU Read Inbound Mailbox, can be programmed to set an SPE event. The event can in turn cause an SPE interrupt. A wrch instruction to the SPU Write Outbound Interrupt Mailbox can also be programmed to cause an interrupt to a processor or other device.

Each time a PPE program writes to the 4-entry SPU Read Inbound Mailbox queue, the channel count for that channel increments. Each time a SPU program reads the mailbox queue, the channel count decrements. The mailbox is a FIFO queue; the SPE program reads the oldest data first. If the PPE program writes more than four times before the SPE program reads the data, then the channel count stays at four, and the fourth location contains the last data written by the PPE. For example, if the PPE program writes five times before the SPE program reads the data, then the data read is the first, second, third, and fifth data elements. The fourth data element has been overwritten.

Mailbox operations are blocking operations: a write to a outbound mailbox register that is already full stalls the SPE until a slot is created in the mailbox by a PPE read. Similarly, a SPE read from an empty inbound mailbox is stalled until the PPE (or an SPE) writes to the mailbox. If the channel capacity count is zero for a channel that is configured as a blocking channel, then a channel instruction issued to that channel causes the SPE to stall and to stop issuing instructions until the channel is read. To prevent stalling in this case, the SPE program needs to read the count register associated with the particular mailbox and decide whether or not to read from or write to the mailbox.

There are at least three ways to deal with anticipated mailbox messages:

- The SPE software reads the channel (`rdch`), which will block until something arrives.
- The SPE software reads from the channel's count (`rchcnt`), which will return the count (zero or one); the software can then decide what to do.
- The SPE software sets up its interrupt facility to respond to mailbox events.

Although the mailboxes are primarily intended for communication between the PPE and the SPEs, they can also be used for communication between an SPE and other SPEs, processors, or devices. For this to happen, however, privileged software needs to allow one SPE to access the mailbox register in another SPE. If software does not allow this, then only system memory communications are available for SPE-to-SPE communications.

## Signal notification

Signal-notification channels, or *signals*, are inbound (to an SPE) registers. They can be used by other SPEs, the PPE, or other devices to send information, such as a buffer-completion synchronization flag, to an SPE.

Each SPE has two 32-bit signal-notification registers, each of which has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor. Unlike mailbox messaging, signal senders use one of three special MFC send-signal commands to send a signal:

- `sndsig`
- `sndsigf`
- `sndsigb`

These are described in “MFC commands” on page 76.

An SPE can only read its local signal-notification channels. The PPE or other processors can write or read the corresponding MMIO register. This allows the target SPE to do polling, blocking, or set up an interrupt as ways of responding to signals. An SPE read of one of its two signal-notification channels clears the channel atomically. An MMIO read does not clear a channel. An SPE read from the signaling channel will be stalled when no signal is pending at the time of the read.



A signal-notification channel can be configured by software to be in *overwrite mode* or *OR mode*. In overwrite mode (also called one-to-one signaling), sending a signal (writing to the MMIO address) overwrites previous contents. In OR mode (also called many-to-one signaling), sending a signal ORs the new 1 bits into the current contents. In the case of one-to-one signaling, there is usually no substantial difference in performance between signaling and using a mailbox.

The differences between mailboxes and signal-notification channels include:

- *Capacity* — Signal-notification channels are registers. Mailboxes are queues.
- *Direction* — Each SPE supports signal-notification channels that are only inbound (to the SPE). Their mailboxes support both outbound and inbound communication. However, an SPE can send signals to another SPE using MFC send-signal commands.
- *Interrupts* — One of the mailboxes interrupts the PPE. Signal-notification channels have no such automatic feature.
- *Many-to-One* — Signal-notification channels (but not mailboxes) can be configured as many-to-one (OR mode) or as one-to-one (overwrite mode).
- *Unique Commands* — Signal-notification channels have specific MFC send-signal commands (`sndsig`, `sndsigf`, and `sndsigb`) for writing to them (see “MFC commands” on page 76).
- *Reset* — Reading a signal-notification register automatically resets (clears) its bits.
- *Count* — The channel counts have different meaning. Mailbox channel counts indicate the number of available (unoccupied) entries in the mailbox queue. The signal-notification channel count indicates whether there are any pending (unserved) signals.
- *Number* — Each SPE has two signal-notification channels versus three mailboxes.

---

## SPU instruction set

The *SPU Instruction Set Architecture (ISA)* fully documents the instructions supported by the SPEs. This section summarizes the ISA.

Programmers writing in a high-level language like C or C++ can use the intrinsics described in “SPU C/C++ language extensions (intrinsics)” on page 64 to improve their control over the SPE hardware. Because the functions performed by these intrinsics are closely related to the assembly-language instructions of the *SPU Instruction Set Architecture*, this overview may be helpful in understanding the utility of the intrinsics.

The SPU ISA operates primarily on SIMD vector operands, both fixed-point and floating-point, with support for some scalar operands. The PPE and the SPE both execute SIMD instructions, but the two processors execute different instruction sets, and programs for the PPE and SPEs must be compiled by different compilers.

## Data layout in registers

The SPE supports *big-endian* data ordering, an ordering in which the lowest-address byte and lowest-numbered bit are the most-significant (high) byte and bit, respectively.

Bits in registers are numbered in ascending order from left to right, with bit 0 representing the most-significant bit (MSb) and bit 127 the least-significant bit (LSb) as shown in the figure below. The SPE architecture does not define or use



little endian data ordering.

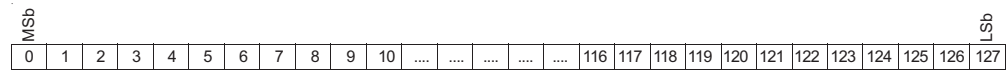


Figure 21. Big-endian ordering supported by the SPE

The SPU hardware defines the following data types:

- byte — 8 bits
- halfword — 16 bits
- word — 32 bits
- doubleword — 64 bits
- quadword — 128 bits

These data types are indicated by shading in Figure 22. The left-most word (bytes 0, 1, 2, and 3) of a register is called the *preferred scalar slot* (also shown in Figure 22).

When instructions use or produce scalar operands or addresses, the values are in the preferred slot. A set of store assist instructions is available to help store bytes, halfwords, words, and doublewords.

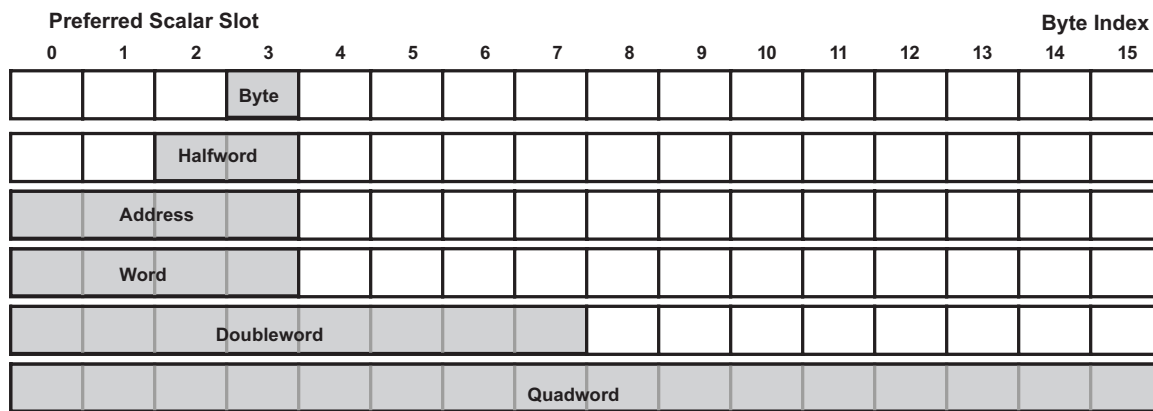


Figure 22. Register layout of data types and preferred (scalar) slot

The SPE programming model defines the vector data types shown in Table 14 for the C programming language. These data types are all 128 bits long and contain from 1 to 16 elements per vector.

Table 14. Vector Data Types

Vector Data Type	Content
vector unsigned char	Sixteen 8-bit unsigned chars
vector signed char	Sixteen 8-bit signed chars
vector unsigned short	Eight 16-bit unsigned halfwords
vector signed short	Eight 16-bit signed halfwords
vector unsigned int	Four 32-bit unsigned words

Table 14. Vector Data Types (continued)

Vector Data Type	Content
vector signed int	Four 32-bit signed words
vector unsigned long long	Two 64-bit unsigned doublewords
vector signed long long	Two 64-bit signed doublewords
vector float	Four 32-bit single-precision floats
vector double	Two 64-bit double precision floats
qword	quadword (16-byte)

## Instruction types

There are 204 instructions in the *SPU Instruction Set Architecture*, and they are grouped into 11 classes according to their functionality.

These instruction classes are shown in Table 15.

Table 15. SPU Instruction Types

Type	Number
Memory Load and Store	16
Constant Formation	6
Integer and Logical Operations	59
Shift and Rotate	31
Compare, Branch, and Halt	40
Hint-for-Branch	3
Floating-Point	28
Control	8
SPU Channel	3
SPU Interrupt Facility	7
Synchronization and Ordering	3

Figure 23 on page 63 shows one example of an SPU SIMD instruction — the floating-point add instruction, *fa*. This instruction simultaneously adds four pairs of floating-point vector elements, stored in registers *ra* and *rb*, and produces four floating-point results, written to register *rt*.

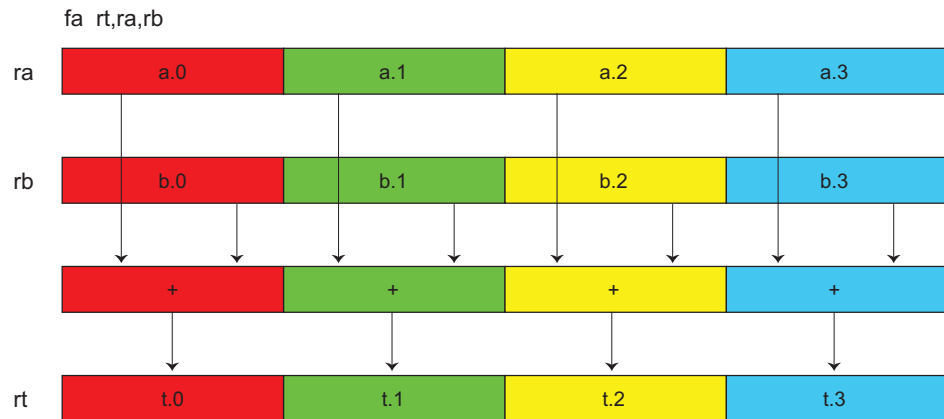


Figure 23. SIMD floating-point Add instruction function

Depending on the programmer's performance requirements and code size restraints, advantages can be gained by properly grouping data in an SIMD vector. Figure 24 shows a natural way of using SIMD vectors to store the homogenous data values ( $x$ ,  $y$ ,  $z$ ,  $w$ ) for the three vertices ( $a$ ,  $b$ ,  $c$ ) of a triangle in a 3D-graphics application. This arrangement is called an *array of structures* (AOS), because the data values for each vertex are organized in a single structure, and the set of all such structures (vertices) is an array.



Figure 24. Array-of-structures data organization for one triangle

The data-packing approach that is shown in Figure 24 often produces small code sizes, but it typically executes poorly and generally requires significant loop-unrolling to improve its efficiency. If the vertices contain fewer components than the SIMD vector can hold (for example, three components instead of four), SIMD efficiencies are compromised.

Another method of organizing data in SIMD vectors is a *structure of arrays* (SOA). Here, each corresponding data value for each vertex is stored in a corresponding location in a set of vectors. Think of the data as if it were scalar, and the vectors are populated with independent data across the vector. This is different from the previous example, where the four values of each vertex are stored in one vector. Figure 25 on page 64 shows the use of SIMD vectors to represent the  $x$ ,  $y$ ,  $z$  vertices for four triangles. Not only are the data types the same across the vector, but now

their data interpretation is the same. Depending on the algorithm, software might execute more efficiently with this SIMD data organization than with the organization shown in Figure 24 on page 63.

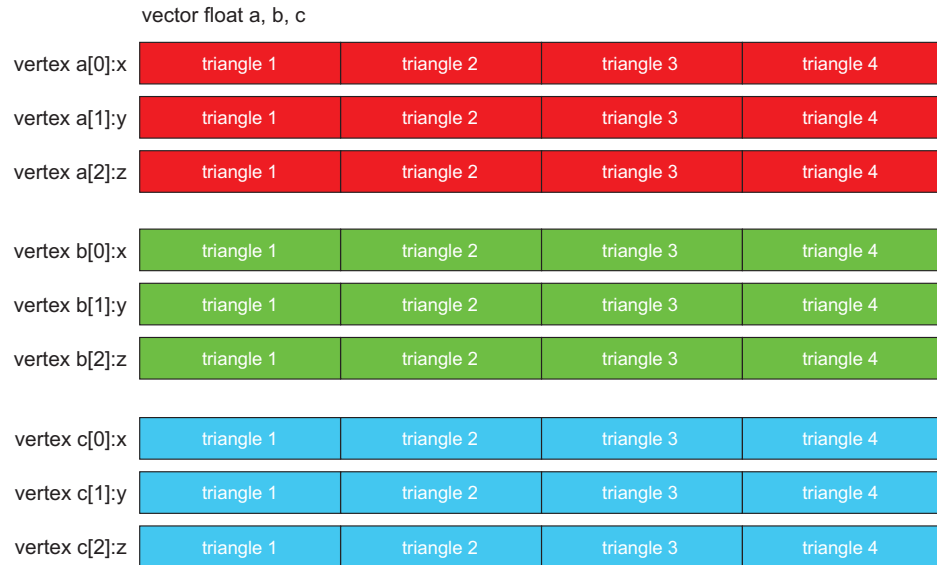


Figure 25. Structure-of-arrays data organization for four triangles

For further details about the SPU instructions, refer to these documents:

- The *SPU Instruction Set Architecture*,
- The *SPU Assembly Language Specification*.

## SPU C/C++ language extensions (intrinsics)

A large set of SPU C/C++ *language extensions (intrinsics)* make the underlying *SPU Instruction Set Architecture* and hardware features conveniently available to C programmers. These intrinsics can be used in place of assembly-language code when writing in the C or C++ languages.

The intrinsics are essentially in-line assembly-language instructions in the form of C-language function calls. They provide the programmer with explicit control of the SPE SIMD instructions without directly managing registers. A well-written compiler that supports these intrinsics will emit efficient code for the SPE architecture. The techniques used by compilers to generate efficient code include:

- Register coloring
- Instruction scheduling (dual-issue optimization)
- Data loads and stores
- Loop blocking, fusion, unrolling
- Correct up-stream placement of branch hints
- Literal vector construction

For example, an SPU compiler provides the intrinsic `t = spu_add(a, b)` as a substitute for the assembly-language instruction `fa rt, ra, rb`. The compiler will generate a floating-point add instruction (`fa rt, ra, rb`) for the SPU intrinsic

`t = spu_add(a, b)`, assuming `t`, `a`, and `b` are vector float variables. The system header file (`spu_intrinsics.h`) defines the SPU language extension intrinsics.

The intrinsics are defined fully in the *C/C++ Language Extensions for Cell Broadband Engine Architecture* specification. The PPU and the SPU instruction sets have similar, but distinct, SIMD intrinsics. It is important to understand the mapping between the PPU and SPU SIMD intrinsics when developing applications on the PPE that will eventually be ported to the SPEs.

## Assembly language versus intrinsics comparison: an example

The ease of implementing a DMA transfer using intrinsics versus assembly-language instructions is illustrated in the example-implementation of the `dma_transfer` subroutine that is provided in this section.

The `dma_transfer` subroutine issues a DMA command with transfer size bytes from the LS address `lsa`, to or from the 64-bit effective address specified by `eah` | `eal`. The DMA command specified by the `dma` parameter is tagged using the specified `tag_id` parameter.

```
extern void dma_transfer(volatile void *lsa, // local store address
                        unsigned int eah,   // high 32-bit effective address
                        unsigned int eal,   // low 32-bit effective address
                        unsigned int size,  // transfer size in bytes
                        unsigned int tag_id, // tag identifier (0-31)
                        unsigned int cmd);  // DMA command
```

The *Application Binary Interface* (ABI)-compliant assembly-language implementation of the subroutine would be:

```
.text
.global dma_transfer
dma_transfer:
wrch    $MFC_LSA, $3
wrch    $MFC_EAH, $4
wrch    $MFC_EAL, $5
wrch    $MFC_Size, $6
wrch    $MFC_TagID, $7
wrch    $MFC_Cmd, $8
bi     $0
```

A comparable C implementation using the SPU intrinsic, `spu_writch`, for the write-channel (`wrch`) instruction would be:

```
#include <spu_intrinsics.h>

void dma_transfer(volatile void *lsa, unsigned int eah, unsigned int eal,
                 unsigned int size, unsigned int tag_id, unsigned int cmd)
{
    spu_writch(MFC_LSA, (unsigned int)lsa);
    spu_writch(MFC_EAH, eah);
    spu_writch(MFC_EAL, eal);
    spu_writch(MFC_Size, size);
    spu_writch(MFC_TagID, tag_id);
    spu_writch(MFC_Cmd, cmd);
}
```

This particular function could be more simply written using the `spu_mfcdma64` composite intrinsic, as:

```
#include <spu_intrinsics.h>

void dma_transfer(volatile void *lsa, unsigned int eah, unsigned int eal,
```

```

        unsigned int size, unsigned int tag_id, unsigned int cmd)
{
    spu_mfcdma64(1sa, eah, eal, size, tag_id, cmd);
}

```

## Intrinsic classes

SPU intrinsics are grouped into the *three* classes that are described in this section.

- *Specific Intrinsics* — Intrinsics that have a one-to-one mapping with a single assembly-language instruction. Programmers rarely need these intrinsics for implementing inline assembly code because the Joint Software Reference Environment (JSRE) has adopted gcc-style inline assembly.
- *Generic Intrinsics* — Intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters.
- *Composite Intrinsics* — Convenience intrinsics constructed from a sequence of specific or generic intrinsics.

Intrinsics are not provided for all assembly-language instructions. Some assembly-language instructions (for example, branches, branch hints, and interrupt return) are naturally accessible through the C/C++ language semantics. Many SPU intrinsics are different than PPE intrinsics (see “Differences between PPE and SPE SIMD support” on page 72).

### Specific intrinsics

Specific intrinsics have a one-to-one mapping with a single assembly-language instruction.

All specific intrinsics are named using the SPU assembly instruction prefixed by the string, `si_`. For example, the specific intrinsic that implements the `stop` assembly instruction is named `si_stop`.

Specific intrinsics are provided for all instructions except branch, branch-hint, and interrupt-return instructions. All specific intrinsics are also available in the form of generic intrinsics, except for the specific intrinsics shown in Table 16. The specific intrinsics shown in this table fall into three categories:

- Instructions generated using basic variable-referencing (that is, using vector and scalar loads and stores),
- Instructions used for immediate vector construction,
- Instructions that have limited usefulness and are not expected to be used except in rare conditions.

Table 16. *Specific intrinsics not available as generic intrinsics*

Intrinsic	Description
<b>Generate Controls for Sub-Quadword Insertion Intrinsics</b>	
<code>d = si_cbd(a, imm)</code>	Generate controls for byte insertion (d form)
<code>d = si_cbx(a, b)</code>	Generate controls for byte insertion (x form)
<code>d = si_cdd(a, imm)</code>	Generate controls for doubleword insertion (d form)
<code>d = si_cdx(a, b)</code>	Generate controls for doubleword insertion (x form)
<code>d = si_chd(a, imm)</code>	Generate controls for halfword insertion (d form)
<code>d = si_chx(a, b)</code>	Generate controls for halfword insertion (x form)
<code>d = si_cwd(a, imm)</code>	Generate controls for word insertion (d form)
<code>d = si_cwx(a, b)</code>	Generate controls for word insertion (x form)

Table 16. Specific intrinsics not available as generic intrinsics (continued)

Intrinsic	Description
<b>Constant Formation Intrinsics</b>	
d = si_il(imm)	Immediate load word
d = si_ila(imm)	Immediate load address
d = si_ilh(imm)	Immediate load halfword
d = si_ilhu(imm)	Immediate load halfword upper
d = si_iohl(a, imm)	Immediate or halfword lower
<b>No Operation Intrinsics</b>	
si_lnop()	No operation (load)
si_nop()	No operation (execute)
<b>Memory Load and Store Intrinsics</b>	
d = si_lqa(imm)	Load quadword (a form)
d = si_lqd(a, imm)	Load quadword (d form)
d = si_lqr(imm)	Load quadword instruction relative
d = si_lqx(a, b)	Load quadword (x form)
si_stqa(a, imm)	Store quadword (a form)
si_stqd(a, b, imm)	Store quadword (d form)
si_stqr(a, imm)	Store quadword instruction relative
si_stqx(a, b, c)	Store quadword (x form)
<b>Control Intrinsics</b>	
si_stopd(a, b, c)	Stop and signal with dependencies

Specific intrinsics accept only the following types of arguments:

- Immediate literals, as an explicit constant expression or as a symbolic address.
- Enumerations.
- Quadword arguments (variables of type `qword`).

Arguments of other types must be cast to the `qword` data type. When using specific intrinsics, it might be necessary to cast from scalar types to the `qword` data type, or from the `qword` data type to scalar types. Similar to casting between vector data types, specific cast intrinsics have no effect on an argument that is stored in a register. All specific casting intrinsics are of the following form:

```
d = casting_intrinsic(a)
```

For example, to add 3 to the integer `i`:

```
int i;
i = si_to_int (si_ai (si_from_int(i), 3));
```

Table 17 lists the specific casting intrinsics.

Table 17. Specific Casting Intrinsics

Intrinsic	Description
si_to_char	Cast byte element 3 of <code>qword</code> to <code>char</code> .
si_to_uchar	Cast byte element 3 of <code>qword</code> to unsigned <code>char</code> .
si_to_short	Cast halfword element 1 of <code>qword</code> to <code>short</code> .

Table 17. Specific Casting Intrinsic (continued)

Intrinsic	Description
si_to_ushort	Cast halfword element 1 of qword to unsigned short.
si_to_int	Cast word element 0 of qword to int.
si_to_uint	Cast word element 0 of qword to unsigned int.
si_to_ptr	Cast word element 0 of qword to a void pointer.
si_to_llong	Cast doubleword element 0 of qword to long long.
si_to_ullong	Cast doubleword element 0 of qword to unsigned long long.
si_to_float	Cast word element 0 of qword to float.
si_to_double	Cast doubleword element 0 of qword to double.
si_from_char	Cast char to byte element 3 of qword.
si_from_uchar	Cast unsigned char to byte element 3 of qword.
si_from_short	Cast short to halfword element 1 of qword.
si_from_ushort	Cast unsigned short to halfword element 1 of qword.
si_from_int	Cast int to word element 0 of qword.
si_from_uint	Cast unsigned int to word element 0 of qword.
si_from_ptr	Cast void pointer to word element 0 of qword.
si_from_llong	Cast long long to doubleword element 0 of qword.
si_from_ullong	Cast unsigned long long to doubleword element 0 of qword.
si_from_float	Cast float to word element 0 of qword.
si_from_double	Cast double to doubleword element 0 of qword.

## Generic intrinsics

Generic intrinsics map to one or more assembly-language instructions, as a function of the type of its input parameters. Generic intrinsics are often implemented as compiler built-ins.

All of the generic intrinsics are prefixed by the string `spu_`. For example, the intrinsic that implements the stop assembly instruction is named `spu_stop`.

Generic intrinsics are provided for *all* SPU instructions, except for the following:

- branch
- branch hint
- interrupt return
- generate control for insertion (used for scalar stores)
- constant formation
- no-op
- memory load and store
- stop and signal with dependencies (`stopd`)

Many generic intrinsics accept scalars as one of their operands. These correspond to intrinsics that map to instructions with immediate values.



Table 18 lists the generic intrinsics.

Table 18. Generic SPU Intrinsics

Intrinsic	Description
<b>Constant Formation Intrinsics</b>	
d = spu_splats(a)	Replicate scalar a into all elements of vector d
<b>Conversion Intrinsics</b>	
d = spu_convtf(a, scale)	Convert integer vector to float vector
d = spu_convts(a, scale)	Convert float vector to signed int vector
d = spu_convtu(a, scale)	Convert float vector to unsigned float vector
d = spu_extend(a)	Sign extend vector
d = spu_rountf(a)	Round double vector to float vector
<b>Arithmetic Intrinsics</b>	
d = spu_add(a, b)	Vector add
d = spu_addx(a, b, c)	Vector add extended
d = spu_genb(a, b)	Vector generate borrow
d = spu_genbx(a, b, c)	Vector generate borrow extended
d = spu_genc(a, b)	Vector generate carry
d = spu_gencx(a, b, c)	Vector generate carry extended
d = spu_madd(a, b, c)	Vector multiply and add
d = spu_mhhadd(a, b, c)	Vector multiply high high and add
d = spu_msub(a, b, c)	Vector multiply and subtract
d = spu_mul(a, b)	Vector multiply
d = spu_mulh(a, b)	Vector multiply high
d = spu_mulhh(a, b)	Vector multiply high high
d = spu_mulo(a, b)	Vector multiply odd
d = spu_mulsr(a, b)	Vector multiply and shift right
d = spu_nmadd(a, b, c)	Negative vector multiply and add
d = spu_nmsub(a, b, c)	Negative vector multiply and subtract
d = spu_re(a)	Vector floating-point reciprocal estimate
d = spu_rsqrte(a)	Vector floating-point reciprocal square root estimate
d = spu_sub(a, b)	Vector subtract
d = spu_subx(a, b, c)	Vector subtract extended
<b>Byte Operation Intrinsics</b>	
d = spu_absd(a, b)	Vector absolute difference
d = spu_avg(a, b)	Vector average
d = spu_sumb(a, b)	Vector sum bytes into shorts
<b>Compare, Branch, and Halt Intrinsics</b>	
d = spu_bisled(func)	Branch indirect and set link if external data
d = spu_cmpabseq(a, b)	Vector compare absolute equal
d = spu_cmpabsgt(a, b)	Vector compare absolute greater than
d = spu_cmpeq(a, b)	Vector compare equal
d = spu_cmpgt(a, b)	Vector compare greater than

Table 18. Generic SPU Ininsics (continued)

Intrinsic	Description
(void) spu_hcmpeq(a, b)	Halt if compare equal
(void) spu_hcmpgt(a, b)	Halt if compare greater than
d = spu_testsv(a, values)	Element-wise test for special value
<b>Bit and Mask Ininsics</b>	
d = spu_cntb(a)	Vector count ones for bytes
d = spu_cntlz(a)	Vector count leading zeros
d = spu_gather(a)	Gather bits from elements
d = spu_maskb(a)	Form select byte mask
d = spu_maskh(a)	Form select halfword mask
d = spu_maskw(a)	Form select word mask
d = spu_sel(a, b, pattern)	Select bits
d = spu_shuffle(a, b, pattern)	Shuffle bytes of a vector
<b>Logical Ininsics</b>	
d = spu_and(a, b)	Vector bit-wise AND
d = spu_andc(a, b)	Vector bit-wise AND with complement
d = spu_eqv(a, b)	Vector bit-wise equivalent
d = spu_nand(a, b)	Vector bit-wise complement of AND
d = spu_nor(a, b)	Vector bit-wise complement of OR
d = spu_or(a, b)	Vector bit-wise OR
d = spu_orc(a, b)	Vector bit-wise OR with complement
d = spu_orx(a)	Bit-wise OR word elements
d = spu_xor(a, b)	Vector bit-wise exclusive OR
<b>Rotate Ininsics</b>	
d = spu_rl(a, count)	Element-wise bit rotate left
d = spu_rlmask(a, count)	Element-wise bit rotate left and mask
d = spu_rlmaska(a, count)	Element-wise bit algebraic rotate and mask
d = spu_rlmaskqw(a, count)	Bit rotate and mask quadword
d = spu_rlmaskqwbyte(a, count)	Byte rotate and mask quadword
d = spu_rlmaskqwbytebc(a, count)	Byte rotate and mask quadword using bit rotate count
d = spu_rlqw(a, count)	Bit rotate quadword left
d = spu_rlqwbyte(a, count)	Byte rotate quadword left
d = spu_rlqwbytebc(a, count)	Byte rotate quadword left using bit rotate count
d = spu_sr(a, count)	Vector Shift Right by Bits
d = spu_sra(a, count)	Vector Shift Right Algebraic by Bits
d = spu_srqw(a, count)	Quadword Shift Right by Bits
d = spu_srqwbyte(a, count)	Quadword Shift Right by Bytes
d = spu_srqwbytebc(a, count)	Quadword Shift Right by Bytes from Bit Shift Count
<b>Shift Ininsics</b>	
d = spu_sl(a, count)	Element-wise bit shift left

Table 18. Generic SPU Intrinsic (continued)

Intrinsic	Description
d = spu_slqw(a, count)	Bit shift quadword left
d = spu_slqwbyte(a, count)	Byte shift quadword left
d = spu_slqwbytebc(a, count)	Byte shift quadword left using bit shift count
<b>Control Intrinsic</b>	
(void) spu_idisable()	Disable interrupts
(void) spu_ienable()	Enable interrupts
(void) spu_mffpscr()	Move from floating-point status and control register
(void) spu_mfspr(register)	Move from special-purpose register
(void) spu_mtfpscr(a)	Move to floating-point status and control register
(void) spu_mtspr(register, a)	Move to special-purpose register
(void) spu_dsync()	Synchronize data
(void) spu_stop(type)	Stop and signal
(void) spu_sync()	Synchronize
<b>Scalar Intrinsic</b>	
d = spu_extract(a, element)	Extract vector element from vector
d = spu_insert(a, b, element)	Insert scalar into specified vector element
d = spu_promote(a, element)	Promote scalar to vector
<b>Channel Control Intrinsic</b>	
d = spu_readch(channel)	Read word channel
d = spu_readchqw(channel)	Read quadword channel
d = spu_readchcnt(channel)	Read channel count
(void) spu_writtech(channel, a)	Write word channel
(void) spu_writtechqw(channel, a)	Write quadword channel

## Composite SPU intrinsic

Composite intrinsic are constructed from a sequence of specific or generic intrinsic.

All of the composite intrinsic are prefixed by the string spu\_. Table 19 lists the composite intrinsic.

Table 19. Composite SPU intrinsic

Intrinsic	Description
spu_mfcdma32(ls, ea, size, tagid, cmd)	Initiate DMA to or from 32-bit effective address
spu_mfcdma64(ls, eahi, ealow, size, tagid, cmd)	Initiate DMA to or from 64-bit effective address
spu_mfcstat(type)	Read MFC tag status

For further information about the SPU intrinsic, refer to the *C/C++ Language Extensions for Cell Broadband Engine Architecture* document.

## Promoting scalar data types to vector data types

The SPU loads and stores one quadword at-a-time. When instructions use or produce scalar operands (including addresses), the value is kept in the preferred scalar slot of a SIMD register.

Scalar (sub quadword) loads and stores require several instructions to format the data for use on the SIMD architecture of the SPE.

Scalar loads must be rotated into the preferred slot. Scalar stores require a read, scalar insert, and write operation. These extra formatting instructions reduce performance.

Vector operations on scalar data are not efficient. The following strategies can be used to make operations on scalar data more efficient:

- Change the scalars to quadword vectors. By eliminating the three extra instructions associated with loading and storing scalars, code size and execution time can be reduced.
- Cluster scalars into groups, and load multiple scalars at a time using a quadword memory access. Manually extract or insert the scalars as needed. This will eliminate redundant loads and stores.

SPU intrinsics are provided in the C/C++ Language Extensions to efficiently promote scalars to vectors, or vectors to scalars. These intrinsics are listed in Table 20.

Table 20. Intrinsic for Changing Scalar and Vector Data Types

Instruction	Description
d = spu_insert	Insert a scalar into a specified vector element.
d = spu_promote	Promote a scalar to a vector.
d = spu_extract	Extract a vector element from its vector.

## Differences between PPE and SPE SIMD support

This section describes the architectural and language-extension differences between PPE and SPE SIMD support.

### Architectural differences between PPE and SPE SIMD support

The PPE processes SIMD operations in the VXU within its PPU. The operations are those of the Vector/SIMD Multimedia Extension instruction set.

The SPEs process SIMD operations in their SPU. The operations are those of the SPU instruction set.

The major differences between the PPE and SPE architectures are summarized in Table 21.

Table 21. PPE and SPE Architectural Comparison

Feature	PPE	SPE
Number of SIMD registers	32 (128-bit)	128 (128-bit)
Organization of register files	separate fixed-point, floating-point, and vector registers	unified
Load latency	variable (cache)	fixed

Table 21. PPE and SPE Architectural Comparison (continued)

Feature	PPE	SPE
Addressability	2 <sup>64</sup> bytes	256-KB local store 2 <sup>64</sup> bytes via DMA
Instruction set	more orthogonal	optimized for single-precision float
Single-precision	IEEE 754-1985	extended range
Doubleword	no doubleword SIMD	double-precision floating-point SIMD

### Language-extension differences between PPE and SPE SIMD support

The SPE's *SPU instruction set* is similar to that of the PPE's *Vector/SIMD Multimedia Extension instruction set*, in that both operate on 128-bit SIMD vectors.

However, from a programmer's perspective, these instruction sets are quite different, and their respective language extensions have different intrinsics and data types.

Table 22 specifies the supported vector data types for each of the SIMD engines (PPE and SPE) in the Cell Broadband Engine, where:

- an "x" signifies *support*
- a "—" signifies *no support*

Table 22. PPE versus SPU Vector Data Types

Vector Data Type	PPE	SPU
vector unsigned char	x	x
vector signed char	x	x
vector bool char	x	—
vector unsigned short	x	x
vector signed short	x	x
vector bool short	x	—
vector pixel	x	—
vector unsigned int	x	x
vector signed int	x	x
vector bool int	x	—
vector float	x	x
vector unsigned long long	—	x
vector signed long long	—	x
vector double	—	x

The key differences are:

- Only the Vector/SIMD Multimedia Extension instruction set supports pixel vectors.
- Only the SPU instruction set supports doubleword vectors.

The SPU's quadword data type is excluded from the list because it is a type-agnostic register reference instead of a specific vector data type. The quadword data type is used exclusively as an operand in *specific intrinsics* — those which have a one-to-one mapping with a single assembly-language instruction. For details, see “Intrinsic classes” on page 66.

Also, the Vector/SIMD Multimedia Extension instruction set provides these operations that are not directly supported by a single instruction in the SPU instruction set:

- Saturating math
- Sum-across
- $\log_2$  and  $2^x$
- Ceiling and floor
- Complete byte instructions

Likewise, the SPU instruction set provides these operations that are not directly supported by a single instruction in the Vector/SIMD Multimedia Extension instruction set:

- Immediate operands
- Double-precision floating-point
- Sum of absolute difference
- Count ones in bytes
- Count leading zeros
- Equivalence
- Nand
- Or complement
- Extend sign
- Gather bits
- Form select mask
- Integer multiply and accumulate
- Multiply subtract
- Multiply float
- Shuffle byte special conditions
- Carry and borrow generate
- Sum bytes across
- Extended shift range

These differences between the Vector/SIMD Multimedia Extension and SPU instruction sets must be kept in mind when porting code from the PPE to the SPE. Ported programs need to consider not only equivalent instructions but also code performance. See “Porting SIMD code from the PPE to the SPEs” on page 98 for more on porting code.

To improve code portability between PPE and SPU programs, `spu_intrinsics.h` provides single-token typedefs for vector keyword data types. These typedefs are shown in Table 23 on page 75.

These single-token types serve as class names for extending generic intrinsics for mapping to-and-from Vector/SIMD Multimedia Extension intrinsics and SPU intrinsics.

Table 23. Single-Token Vector Keyword Data Types

Vector Keyword Data Type	Single-Token Typedef
vector unsigned char	vec_uchar16
vector signed char	vec_char16
vector unsigned short	vec_ushort8
vector signed short	vec_short8
vector unsigned int	vec_unit4
vector signed int	vec_int4
vector unsigned long long	vec_ullong2
vector signed long long	vec_llong2
vector float	vec_float4
vector double	vec_double2

## Compiler directives

Like compiler intrinsics, *compiler directives* are crucial programming elements.

The restrict qualifier is well-known in many C/C++ implementations, and it is part of the SPU language extension. When the restrict keyword is used to qualify a pointer, it specifies that all accesses to the object pointed to are done through the pointer. For example:

```
void *memcpy(void * restrict s1, void * restrict s2, size_t n);
```

By specifying s1 and s2 as pointers that are restricted, the programmer is specifying that the source and destination objects (for the memory copy) do not overlap.

Another directive is `__builtin_expect`. Since branch mispredicts are relatively expensive, `__builtin_expect` provides a way for the programmer to direct branch prediction. This example:

```
int __builtin_expect(int exp, int value)
```

returns the result of evaluating `exp`, and means that the programmer expects `exp` to equal `value`. The value can be a constant for compile-time prediction, or a variable used for run-time prediction.

Two more directives are the aligned attribute, and the `_align_hint` directive. The aligned attribute is used to ensure proper DMA alignment, for efficient data transfer. The syntax is the same as in many implementations of gcc:

```
float factor __attribute__((aligned (16))); //aligns "factor" to a quadword
```

The `_align_hint` directive helps compilers “auto-vectorize”. Although it looks like an intrinsic, it is more properly described as a compiler directive, since no code is generated as a result of using the directive. The example:

```
_align_hint(ptr, base, offset)
```

informs the compiler that the pointer, `ptr`, points to data with a base alignment of `base`, with a byte offset from the base alignment of `offset`. The base alignment must be a power of two. Giving 0 as the base alignment implies that the pointer

has no known alignment. The offset must be less than the base, or, zero. The `_align_hint` directive should not be used with pointers that are not naturally aligned.

---

## MFC commands

The MFC supports a set of *MFC commands*. These commands provide the main mechanism that enables code executing in an SPU to access main storage and maintain synchronization with other processors and devices in the system.

The MFC is described in “Memory flow controller” on page 54. MFC commands can be issued either by code running on the MFC’s associated SPU or by code running on the PPE or other device, as follows:

- Code running on the SPU issues an MFC command by executing a series of writes using *channel instructions*, which are described in Table 13 on page 57.
- Code running on the PPE or other devices issues an MFC command by performing a series of *stores and loads to memory-mapped I/O (MMIO) registers* in the MFC.

The commands are queued in one of two independent MFC command queues:

- *MFC SPU Command Queue* — For channel-initiated commands by the associated SPU
- *MFC Proxy Command Queue* — For MMIO-initiated commands by the PPE or other device

MFC commands that transfer data are referred to as DMA commands. The data-transfer direction for MFC DMA commands is always referenced from the perspective of an SPE. Therefore, commands that transfer data into an SPE (from main storage to local store), are considered get commands, and transfers of data out of an SPE (from local store to main storage) are considered put commands.

The MFC DMA commands are shown in Table 24. This table also indicates whether the commands are supported for SPEs (by means of a corresponding channel) and for the PPE (by means of a corresponding MMIO register), or both.

The suffixes associated with the MFC DMA commands are shown in Table 25 on page 78.

The MFC synchronization commands are shown in Table 26 on page 79.

The MFC atomic commands are shown in Table 27 on page 79.

*Table 24. MFC DMA Command*

Mnemonic	Supported By	Description
<b>Put Commands</b>		
put	PPE, SPE	Moves data from local store to the effective address.
puts	PPE	Moves data from local store to the effective address and starts the SPU after the DMA operation completes.
putf	PPE, SPE	Moves data from local store to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).



Table 24. MFC DMA Command (continued)

Mnemonic	Supported By	Description
putb	PPE, SPE	Moves data from local store to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putfs	PPE	Moves data from local store to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putbs	PPE	Moves data from local store to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putl	SPE	Moves data from local store to the effective address using an MFC list.
putlf	SPE	Moves data from local store to the effective address using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putlb	SPE	Moves data from local store to the effective address using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>Get Commands</b>		
get	PPE, SPE	Moves data from the effective address to local store.
gets	PPE	Moves data from the effective address to local store, and starts the SPU after the DMA operation completes.
getf	PPE, SPE	Moves data from the effective address to local store with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getb	PPE, SPE	Moves data from the effective address to local store with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
getfs	PPE	Moves data from the effective address to local store with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after the DMA operation completes.
getbs	PPE	Moves data from the effective address to local store with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after the DMA operation completes.

Table 24. MFC DMA Command (continued)

Mnemonic	Supported By	Description
getl	SPE	Moves data from the effective address to local store using an MFC list.
getlf	SPE	Moves data from the effective address to local store using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getlb	SPE	Moves data from the effective address to local store using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

The suffixes in Table 25 are associated with the MFC DMA commands, and extend or refine the function of a command. For example, a put command moves data from local store to the effective address. A puts command moves data from local store to the effective address and starts the SPU after the DMA operation completes.

- Commands with an s suffix can only be issued to the MFC Proxy command queue.
- Commands with a l suffix and all the MFC atomic commands can only be issued by the SPE (to the MFC SPU command queue).
- All other commands described in this section can be issued by either the SPE or the PPE.

Commands issued by the PPE are issued on behalf of the SPE and are sent to the MFC Proxy command queue.

Table 25. MFC Command Suffixes

Suffix	Description
s	Starts the execution of the SPU at the current location indicated by the SPU Next Program Counter Register after the data has been transferred into or out of the local store.
f	Tag-specific fence. Commands with a tag-specific fence are locally ordered with respect to all previously-issued commands within the same tag group and command queue.
b	Tag-specific barrier. Commands with a tag-specific barrier are locally ordered with respect to all previously-issued commands within the same tag group and command queue and all subsequently-issued commands to the same command queue with the same tag.
l	List command. Executes a list of DMA transfer elements located in local store. The maximum number of elements is 2,048, and each element describes a transfer of up to 16 KB.

Table 26. MFC Synchronization Commands

Command	Supported By	Description
barrier	PPE, SPE	Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the barrier command within the same command queue. The barrier command has no effect on the immediate DMA commands: getllar, putllc, and putlluc.
mfceieio	PPE, SPE	Controls the ordering of get commands with respect to put commands, and of get commands with respect to get commands accessing storage that is caching inhibited and guarded. Also controls the ordering of put commands with respect to put commands accessing storage that is memory coherence required and not caching inhibited.
mfcsync	PPE, SPE	Controls the ordering of DMA put and get operations within the specified tag group with respect to other processing units and mechanisms in the system.
sndsig	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE.
sndsigb	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE with barrier.
sndsigf	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE with fence.

Table 27. MFC Atomic Commands

Command	Supported By	Description
getllar	SPE	Get lock line and create a reservation (executed immediately).
putllc	SPE	Put lock line conditional on a reservation (executed immediately).
putlluc	SPE	Put lock line unconditional (executed immediately).
putqlluc	SPE	Put lock line unconditional (queued form).

## DMA-command tag groups

All DMA commands except getllar, putllc, and putlluc can be tagged with a 5-bit Tag Group ID.

By assigning a DMA command or group of commands to different tag groups, the status of the entire tag group can be determined within a single command queue (the MFC SPU Command Queue or the MFC Proxy Command Queue).

Software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups. Tagging is optional but can be useful when using barriers to control the ordering of MFC commands within a single command queue.

DMA commands within a tag group can be synchronized with a fence or barrier option by appending an f or b, respectively, to the command mnemonic. Execution of a fenced command option is delayed until all previously issued commands within the same tag group have been performed. Execution of a barrier command

option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.

## Synchronizing DMA transfers

MFC commands can be used to control the order in which DMA storage accesses are performed.

The MFC synchronization commands are shown in Table 26 on page 79. There are:

- four *atomic* commands (`getllar`, `putllc`, `putlluc`, and `putqluc`),
- three *send-signal* commands (`sndsig`, `sndsigf`, and `sndsigb`),
- three *barrier* commands (`barrier`, `mfcsync`, and `mfceieio`).

## MFC input and output macros

The C/C++ Language Extensions for Cell Broadband Engine architecture specification also defines a set of optional convenience macros to assist in accessing the SPU and MFC facilities available through the channel interface.

These macros, specified in `spu_mfcio.h`, can either be implemented as macros or as built-in functions within the compiler.

Table 28. MFC Input and Output Macros

Macro	Description
<b>Effective Address Utilities</b>	
<code>mfc_ea2h(ea)</code>	Extract higher 32-bits from effective address
<code>mfc_ea2l(ea)</code>	Extract lower 32-bits from effective address
<code>mfc_hl2ea(high, low)</code>	Concatenate higher and lower 32-bits of an effective address
<code>mfc_ceil128(value)</code>	Round up value to the next multiple of 128
<b>DMA Commands</b>	
<code>mfc_put(ls, ea, size, tag, tid, rid)</code>	Move data from local storage to effective address
<code>mfc_putb(ls, ea, size, tag, tid, rid)</code>	Move data from local storage to effective address with barrier
<code>mfc_putf(ls, ea, size, tag, tid, rid)</code>	Move data from local storage to effective address with fence
<code>mfc_get(ls, ea, size, tag, tid, rid)</code>	Move data from effective address to local storage
<code>mfc_getb(ls, ea, size, tag, tid, rid)</code>	Move data from effective address to local storage with barrier
<code>mfc_getf(ls, ea, size, tag, tid, rid)</code>	Move data from effective address to local storage with fence
<b>List DMA Commands</b>	
<code>mfc_putl(ls, ea, list, list_size, tag, tid, rid)</code>	Move data from local storage to effective address using MFC list
<code>mfc_putlb(ls, ea, list, list_size, tag, tid, rid)</code>	Move data from local storage to effective address using MFC list with barrier
<code>mfc_putlf(ls, ea, list, list_size, tag, tid, rid)</code>	Move data from local storage to effective address listing MFC list with fence

Table 28. MFC Input and Output Macros (continued)

Macro	Description
<code>mfc_getl(ls, ea, list, list_size, tag, tid, rid)</code>	Move data from effective address to local storage using MFC list
<code>mfc_getlb(ls, ea, list, list_size, tag, tid, rid)</code>	Move data from effective address to local storage using MFC list with barrier
<code>mfc_getlf(ls, ea, list, list_size, tag, tid, rid)</code>	Move data from effective address to local storage using MFC list with fence
<b>Atomic Update Commands</b>	
<code>mfc_getllar(ls, ea, tid, rid)</code>	Get lock line and create reservation
<code>mfc_putllc(ls, ea, tid, rid)</code>	Put lock line if reservation for effective address exists
<code>mfc_putlluc(ls, ea, tid, rid)</code>	Put lock line unconditional
<code>mfc_putqluc(ls, ea, tag, tid, rid)</code>	Put queued lock line unconditional
<b>SL1 Storage Control Commands</b>	
<code>mfc_sdcrt(ea, size, tag, tid, rid)</code>	SL1 Data Cache Range Touch
<code>mfc_sdcrtst(ea, size, tag, tid, rid)</code>	SL1 Data Cache Range Touch for Store
<code>mfc_sdcrz(ea, size, tag, tid, rid)</code>	SL1 Data Cache Range Set to Zero
<code>mfc_sdcrst(ea, size, tag, tid, rid)</code>	SL1 Data Cache Range Store
<code>mfc_sdcrf(ea, size, tag, tid, rid)</code>	SL1 Data Cache Range Flush
<b>Synchronization Commands</b>	
<code>mfc_sndsig(ls, ea, tag, tid, rid)</code>	Send signal
<code>mfc_sndsigb(ls, ea, tag, tid, rid)</code>	Send signal with barrier
<code>mfc_sndsigf(ls, ea, tag, tid, rid)</code>	Send signal with fence
<code>mfc_barrier(tag)</code>	Enqueue <code>mfc_barrier</code> command into DMA queue
<code>mfc_eieio(tag, tid, rid)</code>	Enqueue <code>mfc_eieio</code> command into DMA queue
<code>mfc_sync(tag)</code>	Enqueue <code>mfc_sync</code> command into DMA queue
<b>DMA Status</b>	
<code>mfc_stat_cmd_queue()</code>	Check number of available entries in MFC DMA queue
<code>mfc_write_tag_mask(mask)</code>	Set tag mask to select tag groups to be included in query operation
<code>mfc_read_tag_mask()</code>	Read tag mask indicating groups to be included in query operation
<code>mfc_write_tag_update(ts)</code>	Request the tag status to be updated
<code>mfc_write_tag_update_immediate()</code>	Request that tag status be updated immediately
<code>mfc_write_tag_update_any()</code>	Request that tag status be updated when any tag groups complete
<code>mfc_write_tag_update_all()</code>	Request that tag status be updated when all tag groups complete
<code>mfc_stat_tag_update()</code>	Check availability of tag Update Request Status channel

Table 28. MFC Input and Output Macros (continued)

Macro	Description
mfc_read_tag_status()	Wait for an updated tag status
mfc_read_tag_status_immediate()	Wait for the updated tag status of any enabled group
mfc_read_tag_status_any()	Wait for no outstanding operations for any enabled groups
mfc_read_tag_status_all()	Wait for no outstanding operations for all enabled groups
mfc_stat_tag_status()	Check availability of MFC_RdTagStat channel
mfc_read_list_stall_status()	Read list DMA stall-and-notify status
mfc_stat_list_stall_status()	Check availability of List DMA stall-and-notify status
mfc_write_list_stall_ack(tag)	Acknowledge tag group containing stalled DMA list commands
mfc_read_atomic_status()	Check availability of atomic command status
<b>Multisource Synchronization Request</b>	
mfc_write_multi_src_sync_request()	Request multisource synchronization
mfc_stat_multi_src_sync_request()	Check status of multisource synchronization request
<b>SPU Signal Notification</b>	
spu_read_signal1()	Atomically read and clear Signal Notification 1 channel
spu_stat_signal1()	Check if pending signals exist on Signal Notification 1 channel
spu_read_signal2()	Atomically read and clear Signal Notification 2 channel
spu_stat_signal2()	Check if pending signals exist on Signal Notification 2 channel
<b>SPU Mailboxes</b>	
spu_read_in_mbox()	Read next data entry in the SPU Inbound Mailbox
spu_stat_in_mbox()	Get the number of data entries in the SPU Inbound Mailbox
spu_write_out_mbox(data)	Send data to the SPU Outbound Mailbox
spu_stat_out_mbox()	Get the available capacity of the SPU Outbound Mailbox
spu_write_out_intr_mbox(data)	Send data to the SPU Outbound Interrupt Mailbox
spu_stat_out_intr_mbox()	Get the available capacity of the SPU Outbound Interrupt Mailbox
<b>SPU Decrementer</b>	
spu_read_decrementer()	Read the current value of the decrementer
spu_write_decrementer(count)	Load a value into the decrementer
<b>SPU Events</b>	
spu_read_event_status()	Read the event status or stall until status is available

Table 28. MFC Input and Output Macros (continued)

Macro	Description
spu_stat_event_status()	Check availability of event status
spu_write_event_mask(mask)	Select events to be monitored by event status
spu_write_event_ack(ack)	Acknowledge events
spu_read_event_mask()	Read Event Status Mask
<b>SPU State Mangement</b>	
spu_read_machine_status()	Read current SPU machine status
spu_write_srr0(srr0)	Write to the SPU Save Restore Register 0
spu_read_srr0()	Read the SPU Save Restore Register 0

To facilitate cooperative use of MFC tag ID's amongst multiple autonomous software components, a *tag manager* is provided. The functions provided by the tag manager are specified in Table 29.

Table 29. MFC Tag Manager Functions

Function	Description
mfc_tag_reserve()	Reserve a single tag for exclusive use
mfc_tag_release()	Release a single tag from exclusive use
mfc_multi_tag_reserve()	Reserve a sequential group of tags for exclusive use
mfc_multi_tag_release()	Release a sequential group of tags from exclusive use

To facilitate interrupt safe use of the MFC facilities, the following functions are provided for programmers to guard critical sequences of MFC actions from an interrupt handler use of the MFC.

Table 30. Interrupt Safe Critical Section Functions

Function	Description
mfc_begin_critical_section()	Begin Interrupt-Safe Critical Section
mfc_end_critical_section(stat)	End Interrupt-Safe Critical Section

## Coding methods and examples

The sections included here describe some coding methods, with examples in SPU assembly language, C language, SPU C-language intrinsics, and MFC commands, or in a combination thereof.

These *instruction and command sets* are summarized in:

- SPU assembly language — (see “SPU instruction set” on page 60)
- SPU C-language intrinsics — (see “SPU C/C++ language extensions (intrinsics)” on page 64)
- MFC commands — (see “MFC commands” on page 76)

## DMA transfers

DMA commands transfer data between the LS and main storage.

Main storage is addressed by an effective address (EA) operand in a DMA command. The LS is addressed by the local store address (LSA) operand in a DMA command. The size of a single DMA transfer is limited to 16 KB:

- put commands move data from LS to main storage.
- get commands move data from main storage to LS.

The LS data is accessed sequentially with a minimum step of one quadword.

Software on an SPE accesses its MFC's DMA-transfer facilities through the channels listed in "Channels" on page 55. To enqueue a DMA command, SPE software writes the MFC *Command Parameter Channel Registers* with the wrch instruction (described in "Channel instructions" on page 57) in the following sequence:

1. Write the EA-high (EAH) to the MFC\_EAH channel.
2. Write the EA-low (EAL) to the MFC\_EAL channel.
3. Write the transfer size to the MFC\_Size channel.
4. Write the tag ID to the MFC\_TagID channel.
5. Write the class ID and command opcode to the MFC\_Cmd channel.

The following examples shows how to initiate a DMA transfer from an SPE.

```
extern void dma_transfer(volatile void *lsa,    // local store address
                        unsigned int eah,      // high 32-bit effective address
                        unsigned int eal,      // low 32-bit effective address
                        unsigned int size,     // transfer size in bytes
                        unsigned int tag_id,   // tag identifier (0-31)
```

An ABI-compliant assembly-language implementation of the subroutine is:

```
.text
.global dma_transfer
dma_transfer:
wrch    $MFC_LSA, $3
wrch    $MFC_EAH, $4
wrch    $MFC_EAL, $5
wrch    $MFC_Size, $6
wrch    $MFC_TagID, $7
wrch    $MFC_Cmd, $8
bi      $0
```

A comparable C implementation using the SPU composite intrinsic spu\_mfcdma64 is:

```
#include <spu_intrinsics.h>
void dma_transfer(volatile void *lsa, unsigned int eah, unsigned int eal,
                 unsigned int size, unsigned int tag_id, unsigned int cmd)
{
    spu_mfcdma64(lsa, eah, eal, size, tag_id, cmd);
}
```

The performance of a DMA data transfer is best when the source and destination addresses are aligned on a cache line boundary are at least a cache line sized.

Quadword-offset-aligned data transfers generate full cache-line bus requests for every unrolling, except possibly the first and last unrolling.

Transfers that start or end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first or last bus request, respectively.



## Simple SPE-initiated DMA transfers example

This simple example demonstrates how to move data between main storage and SPE local store using SPE initiated DMA transfers, specifically, the following techniques are used:

- The tag manager is used on the SPE to reserve and release tag ID.
- SPU program uses the DMA get command to transfer data from main memory to local memory
- SPU program uses the DMA put command to transfer data from SPU local memory to main memory.
- SPU program waits for the DMA commands to complete.

In this example, the PPU program allocates a buffer in main storage, loads the buffer with a Fibonacci sequence, and then starts the SPU program with the address of the buffer passed to the SPU program as an argument. Once the SPU program started, the PPU program waits for the SPU program to finish. The SPU program uses the DMA get command to transfer the buffer into its local memory. It then examines the content of the buffer, verifies that the buffer contains the Fibonacci sequence, increments every number in the buffer by one, then transfers the buffer back to main storage using the DMA put command.

This example uses the MFC macros specified in the C/C++ Language Extensions for Cell Broadband Architecture specification.

The full listing of the program can be found in `/opt/cell/sdk/src/tutorial/dma/simple`.

## Transferring large buffer from main memory to SPE LS examples

A DMA list is a sequence of *transfer elements* (or list elements) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers between a single area of LS and possibly discontinuous areas in main storage.

Since the SPE local store is limited, many applications will need to transfer small pieces of data in the SPE for processing at a time. The following examples illustrate different methods one can use to transfer large amount of data from main storage to SPE local memory for processing.

All the examples use the same PPU program. The PPU program allocates a large buffer in main storage, loads the buffer with a sequence of numbers (from 0 to the size of the buffer), then starts the SPU program. Note that the large buffer is much bigger than the SPU local memory size. The PPU program also allocates a control block structure that contains necessary information for the SPU to fetch the large buffer.

The definition for the control block structure can be found at `/opt/cell/sdk/src/tutorial/dma/dma_example.h`

The full listing for the PPU driver program can be found at `/opt/cell/sdk/src/tutorial/dma/dma_example_multi_spe.c`

## Single buffering

This example program demonstrates how to use the DMA calls on Cell to transfer a large array of data from main memory to SPE local storage, process, and write the result back to main memory using single buffering technique. The processing is done in place with only one local buffer.

In this example, the SPU program allocates a `CHUNK_SIZE` elements buffer in the local memory to store the input buffer and output buffer. The `process_data_simd` function processes the data in place and stores the result back to the buffer. The SPU program does the following:

- Allocates SPU local storage buffer that is aligned on 128 byte boundary. The `local_buffer` buffer is for storing input data from system memory and output data.
- Reserves a MFC tag for DMA usage.
- Initiates a DMA transfer to get the control block. Waits for the transfer to complete.
- Calculates the number of blocks (chunks) that this SPU is assigned to process.
- For each of the block of data (number of blocks computed in the previous step):
  - Issues DMA get command to transfer each block of data from system memory to the SPE local store and waits for the transfer to finish.
  - Processes the incoming data.
  - Issues DMA put command to transfer the result from SPE local store to system memory and waits for the transfer to finish.

The full listing of the SPU program can be found at `/opt/cell/sdk/src/tutorial/dma/single_buffer/`

## Double buffering

The previous `single_buffer` example wasted a great deal of time waiting for DMA transfers to complete. We can speed up the process significantly by allocating two buffers and overlapping computation on one buffer with data transfer in the other. This technique is called double buffering. The following figure shows a flow diagram for this scheme.

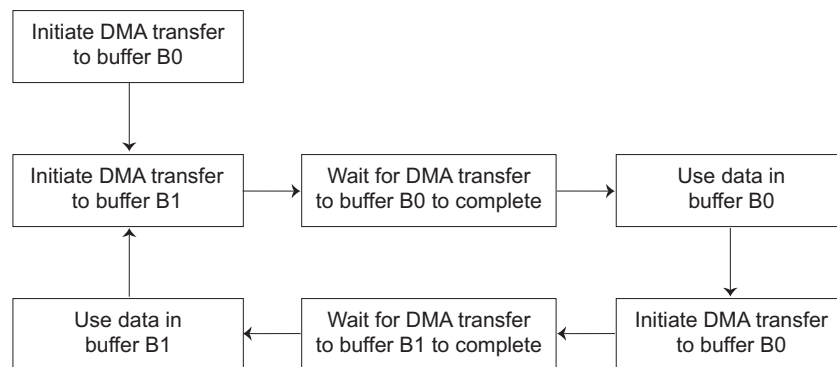


Figure 26. DMA transfers using a double-buffering method

To use double buffering effectively, follow these rules for DMA transfers on the SPE:

- Use multiple LS buffers.
- Use unique DMA tag IDs, one for each LS buffer or logical group of LS buffers.
- Use fenced command options to order the DMA transfers within a tag group.
- Use barrier command options to order DMA transfers within the MFC's DMA controller.

The purpose of double buffering is to maximize the time spent in the compute phase of a program and minimize the time spent waiting for DMA transfers to complete. Let  $\tau_t$  represent the time required to transfer a buffer B, and let  $\tau_c$  represent the time required to compute on data contained in that buffer. In general, the higher the ratio  $\tau_t/\tau_c$ , the more performance benefit an application will realize from a double-buffering scheme.

The following example demonstrates how to transfer a large array of data from main memory to SPE local storage, process, and write the result back to main memory using double buffering technique. This example assumes that the resulting output cannot overwrite the input buffer in local store. Four buffers are allocated on the stack, two for input and two for output. We also reserve two MFC tags to group the transfers.

The following table shows the different activities that happen at each time step. For each of the MFC command that is issued, the table also shows the MFC tag id to be used with that command.

*Table 31. Time step description*

Time step	Action	Tag
0	get in_buf0	0
1	get in_buf1 wait in_buf0 out_buf0 = compute (in_buf0)	1 0
2	put out_buf0 get in_buf0 wait in_buf1 out_buf1 = compute (in_buf1)	0 0 1
3	put out_buf1 get in_buf1 wait in_buf0 out_buf0 out_buf0 = compute (in_buf0)	1 1 0
4	put out_buf0 get in_buf0 wait in_buf1 out_buf1 out_buf1 = compute (in_buf1)	0 0 1
5	put out_buf1 get in_buf1 wait in_buf0 out_buf0 out_buf0 = compute (in_buf0)	1 1 0
6	put out_buf0 wait in_buf1 out_buf1 out_buf1 = compute (in_buf1)	0 1
7	put out_buf1 wait out_buf1	1 1

A full listing of this program can be found at: /opt/cell/sdk/src/tutorial/dma/double\_buffer\_in\_out

## Double buffering in place

The double\_buffer example listed above, while allows users to overlap computation and data transferring, requires four local store buffers. Since the SPE local memory is only 256 KB, this puts a lot of demand on this limited resource. If the processing of a buffer can be performed in place, the number of buffers can be reduced to two or three.

This example shows how to use the DMA calls on Cell to transfer a large array of data from main memory to SPE local storage, process, and write the result back to system memory using double buffering technique. This example assumes that the algorithm can be done in-place and only one single set of I/O buffers is required, so only two buffers are needed to be allocated in the SPU local store.

The following table shows the different activities that happen at each time step. For each of the MFC command that is issued, the table also shows the MFC tag id to be used with that command.

Table 32. Time step description

Time step	Action	Tag
0	get buf0	0
1	get buf1 wait buf0 buf0 = compute (buf0)	1 0
2	put buf0 getb buf0 wait buf1 buf1 = compute (buf1)	0 0 1
3	put buf1 getb buf1 wait buf0 buf0 = compute (buf0)	1 1 0
4	put buf0 getb buf0 wait buf1 buf1 = compute (buf1)	0 0 1
5	put buf1 getb buf1 wait buf0 buf0 = compute (buf0)	1 1 0
6	put buf0 wait buf1 buf1 = compute (buf1)	0 1
7	put buf1 wait buf1	1 1

To avoid a possible race condition, we use the MFC command getb (DMA get with barrier). The next input buffer must be ordered with respect to the put of the output buffer. For example, in time step 2, using the getb command guarantees that buf0 will not be overrun with new data from the getb command before the put command on the same buffer is finished.

A full listing of this program can be found at: /opt/cell/sdk/src/dma/  
double\_buffer

## Triple buffering in place

Another method of multi-buffering that uses less SPU local memory than full-blown double buffering is triple buffering. This method assumes that the buffer processing is done in place.

In this example, we allocate three different buffers on the stack. The three buffers are used both as input and output buffers. We also reserve three MFC tag ids to group the DMA transfers based on the buffer usage.

The following table shows the different activities that happen at each time step.

Table 33. Time step description

Time step	Action	Tag
0	get buf0	0
1	get buf1 wait buf0 buf0 = compute (buf0)	1 0
2	put buf0 get buf2 wait buf1 buf1 = compute (buf1)	0 2 1
3	put buf1 get buf0 wait buf2 buf2 = compute (buf2)	1 0 2
4	put buf2 get buf1 wait buf0 buf0 = compute (buf0)	2 1 0
5	put buf0 get buf2 wait buf1 buf1 = compute (buf1)	0 2 1
6	put buf1 wait buf2 buf2 = compute (buf2)	1 2
7	put buf2 wait buf2	2 2

A full listing of this program can be found at: /opt/cell/sdk/src/dma/  
triple\_buffer

## Triple buffering

Another multi-buffering method that uses less SPU local memory than full-blown double buffering and still allows for separate input and output buffers is triple buffering.

In this example, we allocate three different local store buffers on the stack. The three buffers are used as input and output buffers interchangeably. We also reserve three MFC tags to group the DMA transfers based on the buffer usage.

The following table shows the different activities that happen at each time step. For each of the MFC command that is issued, the table also shows the MFC tag id to be used with that command.

Table 34. Time step description

Time step	Action	Tag
0	get buf0	0
1	get buf1 wait buf0 buf2 = compute (buf0)	1 0
2	put buf2 getb buf2 wait buf1 buf0 = compute (buf1)	2 2 1
3	put buf0 getb buf0 wait buf2 buf1 = compute (buf2)	0 0 2
4	put buf1 getb buf1 wait buf0 buf2 = compute (buf0)	1 1 0
5	put buf2 getb buf2 wait buf1 buf0 = compute (buf1)	2 2 1
6	put buf0 wait buf2 buf1 = compute (buf2)	0 2
7	put buf1 wait buf1	1 1

To avoid a possible race condition, we use the MFC command `getb` (DMA get command with barrier). The next input buffer must be ordered with respect to the previous `put` of the output buffer. For example, in time step 2, using the `getb` command guarantees that `buf2` will not be overrun with new data from the `getb` command before the `put` command on the same buffer is finished.

A full listing of this program can be found at: `/opt/cell/sdk/src/dma/triple_buffer`

## DMA-list transfers

A DMA list is a sequence of *transfer elements* (or list elements) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers between a single area of LS and possibly discontinuous areas in main storage.

Such DMA lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA-list command, such as `getl` or `putl`.

DMA-list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

### Creating the DMA list

Each transfer element in the DMA list contains a transfer size, the low half of an effective address, and a stall-and-notify bit that can be used to suspend list execution after transferring a list element whose stall-and-notify bit is set.

Each DMA transfer specified in a list can transfer up to 16 KB of data, and the list can have up to 2,048 (2 K) transfer elements.

Software creates the list and stores it in the LS. Lists must be stored in the LS on an 8-byte boundary. The form of a transfer element is {LTS, EAL}.

- The first word (LTS) is the list transfer size, the most-significant bit of which serves as an optional stall-and-notify flag.
- The second word (EAL) is the low-order 32-bits of an EA.

Transfer elements are processed sequentially, in the order they are stored. If the stall-and-notify flag is set for a transfer element, the MFC will stop processing the DMA list after performing the transfer for that element until the SPE program clears the DMA List Command Stall-And-Notify Event from the SPU Read Event Status Channel. This gives programs an opportunity to modify subsequent transfer elements before they are processed by the MFC.

### Initiating the transfers specified in the DMA list

After the list is stored in the LS, the execution of the list is initiated by a DMA-list command, such as `getl` or `putl`, from the SPE whose LS contains the list.

DMA-list commands, such as single-transfer DMA commands, require that parameters are written to the MFC Command Parameter channels in the way described in “DMA transfers” on page 83. However, a DMA-list command requires two different types of parameters than those required by a single-transfer DMA command:

- `MFC_EAL` : This parameter must be written with the *starting local store address (LSA) of the list*, rather than with the EAL. (The EAL is specified in each transfer element.)
- `MFC_Size` : This parameter must be written with the *size of the list*, rather than the transfer size. (The transfer size is specified in each transfer element.) The list size is equal to the number of transfer elements, multiplied by the size of the transfer-element structure (8 bytes).

The starting LSA and the EA-high (EAH) are specified only once, in the DMA-list command that initiates the transfers. The LSA is internally incremented based on the amount of data transferred by each transfer element. However, if the starting LSA for each transfer element in a list does not begin on a 16-byte boundary, then hardware automatically increments the LSA to the next 16-byte boundary.

The EAL for each transfer element is in the 4-GB area defined by EAH.

### DMA-list transfers: programming example

The C-language sample program included here creates a DMA list and, in the last line, uses an `spu_mfcdma32` intrinsic to issue a single DMA-list command (`getl`) to transfer a main-storage region into LS.

```
/* dma_list_sample.c - SPU MFC-DMA list sample code.
 *
 * This sample defines a transfer-element data structure, which
 * contains the element's transfer size and low-order 32 bytes of the effective
 * address. Also defined in the structure, but not used by this sample,
 * is the DMA-list stall-and-notify bit, which can be used to indicate
 * that the MFC should suspend list execution after transferring a list
 * element whose stall-and-notify bit is set.
 */

#include <spu_mfcio.h>
```

```

struct dma_list_elem {
    union {
        unsigned int all32;
        struct {
            unsigned nbytes: 31;
            unsigned stall: 1;
        } bits;
    } size;
    unsigned int ea_low;
};

struct dma_list_elem list[16] __attribute__((aligned (8)));

void get_large_region(void *dst, unsigned int ea_low, unsigned int nbytes)
{
    unsigned int i = 0;
    unsigned int tagid = 0;
    unsigned int listsize;

    /* get_large_region
     * Use a single DMA list command request to transfer
     * a "large" memory region into LS. The total size to
     * be copied may be larger than the MFC's single element
     * transfer limit of 16kb.
     */

    if (!nbytes)
return;

    while (nbytes > 0) {
        unsigned int sz;

        sz = (nbytes < 16384) ? nbytes : 16384;
        list[i].size.all32 = sz;
        list[i].ea_low = ea_low;

        nbytes -= sz;
        ea_low += sz;
        i++;
    }

    /* Specify the list size and initiate the list transfer
     */

        listsize = i * sizeof(struct dma_list_elem);
        spu_mfcdma32(dst, (unsigned int)&list[0], listsize, tagid, MFC_GETL_CMD);
    }
}

```

## Transferring data from main memory to SPE LS using DMA List

The following examples demonstrate different methods one can use DMA list to transfer data from main storage to SPE local memory for processing.

All the examples use the same PPU program from the “Transferring large buffer from main memory to SPE LS”. Instead of initiating multiple individual DMAs from the SPE to transfer the data from main memory to SPE LS, we are using DMA lists.



### Single buffering with list

This example is very similar to the single buffering example in the previous section. However, instead of using individual DMA transfers to get the data from main memory to SPE local memory, we are creating DMA list elements to gather data from main memory to SPE LS.

To store the DMA list elements, we reserve some space on the stack for a DMA list. The example assumes that the processing is done with two separate buffers, one for input and one for output. However, since this example processes the data sequentially, we only need one DMA list element array.

The full listing of the SPU program can be found at `/opt/cell/sdk/src/dma/single_buffer_list_in_out`

### Double buffering with list

This example is very similar to the double buffering example in the previous section. However, instead of using individual DMA transfers to get the data from main memory to SPE local memory, we are creating DMA list elements to gather data from main memory to SPE LS.

We are reserving space on the stack for four DMA list element arrays, two for input and two for output.

The full listing of the SPU program can be found at `/opt/cell/sdk/src/dma/double_buffer_list_in_out`

### Double buffering with list in place

This example is very similar to the double buffering example with the processing done in place in the previous section. However, instead of using individual DMA transfers to get the data from main memory to SPE local memory, we are creating DMA list elements to gather data from main memory to SPE LS.

Note that even though we are using only two buffers for storing input and output data in this multi-buffering scheme, we need to reserve space on the stack for four DMA list element arrays, two for input and two for output.

The full listing of the SPU program can be found at `/opt/cell/sdk/src/dma/double_buffer_list`

### Triple buffering with list

This example is very similar to the triple buffering example in the previous section. However, instead of using individual DMA transfers to get the data from main memory to SPE local memory, we are creating DMA list elements to gather data from main memory to SPE LS.

Note that even though we are using three buffers for storing input and output data in this multi-buffering scheme, we need to reserve space on the stack for four DMA list element arrays, two for input and two for output.

The following table shows the different activities that happen at each time step. For each of the MFC command that is issued, the table also shows the MFC tag id to be used with that command.

A full listing of this program can be found at `/opt/cell/sdk/src/dma/triple_buffer_list_in_out`

## Destructive DMA list example

The SPEs come with fairly limited local storage, 256KB for each SPE. Efficiently manage this small space is necessary to achieve high memory performance. One of the recommended techniques to save space on the SPE is destructive DMA list.

Destructive DMA list technique enables users to save SPE local storage by constructing the list element array at the end of the target input buffer. On transfer, the list element array will be overwritten by the data. The CBE architecture guarantees that all list elements within the list command are started and issued in sequence so that as long as every list element is of non-zero size, the list array is guaranteed to not be overwritten before it is processed.

This example assumes that the data processing step is done with two separate buffers, one for input and one for output. We reserve space on the stack for the output DMA list element array. For the input DMA list array, we set the pointer to the end of the local buffer. Here is the code segment for setting the input DMA list element array to the appropriate local storage address:

```
mfc_list_element_t* dma_list_in;
unsigned int tmp_addr;

/* calculate the address of the local buffer where we can point the
 * dma_list_in pointer to */
tmp_addr = (unsigned int)((local_buffer_in + sizeof(float)*CHUNK_SIZE * NUM_LIST_ELEMENTS) -
    (sizeof (mfc_list_element_t) * NUM_LIST_ELEMENTS));
dma_list_in = (mfc_list_element_t*) (tmp_addr);
```

The full listing of the SPU program can be found at `/opt/cell/sdk/src/dma/destructive_dma_list`

## Vectorizing a loop

A compiler that automatically merges scalar data into a parallel-packed SIMD data structure is called an *auto-vectorizing* compiler. Such compilers must handle all the high-level language constructs, and therefore do not always produce optimal code.

A simple example of vectorizing a loop is shown below. The original loop multiplies two arrays, term by term. The arrays are assumed to remain scalar outside of the subroutine `vmult`.

```
/* Scalar version */
int mult(float *array1, float *array2, float *out, int arraySize) {
    int i;
    for (i = 0; i < arraySize; i++) {
        out[i] = array1[i] * array2[i];
    }
    return 0;
}

/* Vectorized version */
int vmult(float *array1, float *array2, float *out, int arraySize) {
    /* This code assumes that the arrays are quadword-aligned. */
    /* This code assumes that the arraySize is divisible by 4. */

    int i, arraySizebyfour;
    arraySizebyfour = arraySize >> 2; /* arraySize/4 vectors */
    vector float *varray1 = (vector float *) (array1);
    vector float *varray2 = (vector float *) (array2);
    vector float *vout = (vector float *) (out);

    for (i = 0; i < arraySizebyfour; i++) {
        /*spu_mul is an intrinsic that multiplies vectors */
```

```

    vout[i] = spu_mul(varray1[i], varray2[i]);
}

return 0;
}

```

## Reducing the impact of branches

The SPU hardware assumes linear instruction flow, and produces no stall penalties from sequential instruction execution. A *branch instruction* has the potential of disrupting the assumed sequential flow.

Correctly predicted branches execute in one cycle, but a mispredicted branch (conditional or unconditional) incurs a penalty of approximately 18-19 cycles. Considering the typical SPU instruction latency of two-to-seven cycles, mispredicted branches can seriously degrade program performance. Branches also create scheduling barriers, reducing the opportunity of for dual issue and covering up dependency stalls.

The most effective means of reducing the impact of branches is to eliminate them using three primary methods — inlining, unrolling, and predication. The next effective means of reducing the impact of branches is to use the branch-hint instructions.

If a branch hint is provided, software speculates that the instruction branches to the target path. If a hint is not provided, software speculates that the branch is not taken (that is, instruction execution continues sequentially). If either speculation is incorrect, there is a large penalty (flush and refetch).

### Function-inlining and loop-unrolling

*Function-inlining* and *loop-unrolling* are two techniques often used to increase the size of basic blocks (sequences of consecutive instructions without branches), which increases scheduling opportunities.

Function-inlining eliminates the two branches associated with function-call linkage. These include the branch and set link for function-call entry, and the branch indirect for function-call return.

Loop-unrolling eliminates branches by decreasing the number of loop iterations. Loop unrolling can be manual, compiler directed, or compiler automated. Typically, branches associated with looping are inexpensive because they are highly predictable. However, if a loop can be fully unrolled, then all branches can be eliminated—including the final nonpredicted branch.

Care should be taken when exploiting function inlining and loop unrolling. Over-aggressive use of these techniques can result in code that is too large to fit in the LS.

### Predication using select-bits instruction

The select-bits (*selb*) instruction is the key to eliminating branches for simple control-flow statements (for example, *if* and *if-then-else* constructs). An *if-then-else* statement can be made branchless by computing the results of both the *then* and *else* clauses and using select bits (*selb*) to choose the result as a function of the conditional.

If computing both the results costs less than a mispredicted branch, then there are additional savings.

For example, consider the following simple if-then-else statement:

```
unsigned int a, b, c;
...
if (a > b) d += a;
else      d += 1;
```

This code sequence, when directly converted to an SPU instruction sequence without branch optimizations, would look like:

```
clgt cc, a, b
brz cc, else
then:
a d, d, a
br done
else:
ai d, d, 1
done:
```

Using the select bits instruction, this simple conditional becomes:

```
clgt cc, a, b /* compute the greater-than condition */
a d_plus_a, d, a /* add d + a */
ai d_plus_1, d, 1 /* add d + 1 */
selb d, d_plus_1, d_plus_a, cc /* select proper result */
```

This example shows:

- Both branches were eliminated, and the correct result was placed in d .
- New registers were needed to maintain potential values of d (d\_plus\_a and d\_plus\_1 ). This does not put significant pressure on the register file because the register file is so large and the life of these variables is very short.
- The rewritten code sequence is smaller.
- The latency of the operations permits the scheduler to cover most of the cost of computing both conditions. Further scheduling these instructions with those before and after this code sequence will likely improve performance even further.

Here is another example of using the select bit — this time with C intrinsics. This code fragment shows how to use SPU intrinsics, including spu\_cmpgt , spu\_add , and spu\_sel , to eliminate conditional branches.

The following sequence generates four instructions, assuming a, b, c are already in registers (because we are promoting and extracting to and from the preferred integer element, the spu\_promote and spu\_extract intrinsics produce no additional instructions):

```
unsigned int a,b,c;
vector unsigned int vc1, vab, va, vb, vc;

va = spu_promote(a, 0);
vb = spu_promote(b, 0);
vc = spu_promote(c, 0);
vc1 = spu_add(vc, 1);
vab = spu_add(va, vb);
vc = spu_sel(vab, vc1, spu_cmpgt(va, vb));
c = spu_extract(vc, 0);
```

## Reducing branch mispredicts with branch hint

General-purpose processors have typically addressed branch prediction by supporting hardware look-asides with branch history tables (BHT), branch target address caches (BTAC), or branch target instruction caches (BTIC).

The SPU addresses branch prediction through a set of hint for branch (HBR) instructions that facilitate efficient branch processing by allowing programs to avoid the penalty of taken branches.

- If a branch hint is provided, software speculates that the instruction branches to the target path.
- If a hint is not provided, software speculates that the instruction does not branch to a new location (that is, it stays inline).
- If speculation is incorrect, the speculated branch is flushed and refetched.

It is possible to sequence multiple hints in advance of multiple branches. As with all programmer-provided hints, care must be exercised when using branch hints because, if the information provided is incorrect, performance might degrade.

Branch-hint instructions can provide three kinds of advance knowledge about future branches:

- Address of the branch target (that is, where will the branch take the flow of control)
- Address of the actual branch instruction (known as the *hint-trigger address* )
- Prefetch schedule (when to initiate prefetching instructions at the branch target)

Branch-hint instructions load a branch-target buffer (BTB) in the SPU. When the BTB is loaded with a branch target, the hint-trigger address and branch address are also loaded into the BTB. After loading, the BTB monitors the instruction stream as it goes into the issue stage of the pipeline. When the address of the instruction going into issue matches the hint trigger address, the hint is triggered, and the SPU speculates to the target address in the hint buffer.

Branch-hint instructions have no program-visible effects. They provide a hint to the SPE architecture about a future branch instruction, with the intention that the information be used to improve performance by prefetching the branch target. The SPE branch-hint instructions are shown in Table 35. There are immediate and indirect forms for this instruction class. The location of the branch is always specified by an immediate operand in the instruction.

*Table 35. Branch-Hint Instructions*

Instruction	Description
hbr s11, ra	Hint for branch (r-form). Hint that the instruction addressed by the sum of the address of the current instruction and the signed extended, 11-bit value s11 will branch to the address contained in word element 0 of register ra. This form is used to hint function returns, pointer function calls, and other situations that give rise to indirect branches.
hbra s11, s18	Hint for branch (a-form). Hint that the instruction addressed by the sum of the address of the current instruction and the signed extended, 11-bit value s11 will branch to the address specified by the sign extended, 18-bit value s18.
hbrr s11, s18	Hint for branch relative. Hint that the instruction addressed by the sum of the address of the current instruction and the signed extended, 11-bit value s11 will branch to the address specified by the sum of the address of the current instruction and sign extended, 18-bit value s18.

The following rules apply to the hint for branch (HBR) instructions:

- An HBR instruction should be placed at least 11 cycles followed by four instruction pairs before the branch instructions being hinted by the HBR instruction. In other words, an HBR instruction must be followed by at least 11 cycles of instructions, followed by eight instructions aligned on an even address boundary. More separation between the hint and branch improves the performance of applications on future SPU implementations.
- If an HBR instruction is placed too close to the branch, then a hint stall will result. This results in the branch instruction stalling until the timing requirement of the HBR instruction is satisfied.
- If an HBR instruction is placed closer to the hint-trigger address than four instruction pairs plus one cycle, then the hint stall does not occur and the HBR is not used.
- Only one HBR instruction can be active at a time. Issuing another HBR cancels the current one.
- An HBR instruction can be moved outside of a loop and will be effective on each loop iteration as long as another HBR instruction is not executed.
- The HBR instruction must be placed within 255 instructions of the branch instruction.
- The HBR instruction only affects performance.

The HBR instructions can be used to support multiple strategies of branch prediction. These include:

- *Static Branch Prediction* — Prediction based upon branch type or displacement, and prediction based upon profiling or linguistic hints.
- *Dynamic Branch Prediction* — Software caching of branch-target addresses, and using control flow to record branching history.

A common approach to generating static branch prediction is to use expert knowledge that is obtained either by feedback-directed optimization techniques or using linguistic hints supplied by the programmer.

The document *C/C++ Language Extensions for Cell Broadband Engine Architecture* defines a mechanism for directing branch prediction. The `__builtin_expect` directive allows programmers to predict conditional program statements. The following example demonstrates how a programmer can predict that a conditional statement is false (a is not larger than b).

```
if(__builtin_expect((a>b),0))
    c += a;
else
    d += 1;
```

Not only can the `__builtin_expect` directive be used for static branch prediction, it can be used for dynamic branch prediction.

---

## Porting SIMD code from the PPE to the SPEs

For some, it is easier to write SIMD programs by writing them first for the PPE, and then porting them to the SPEs. This approach postpones some SPE-related considerations of dealing with the local store (LS) size, data movements, and debug until after the port. The approach can also allow partitioning of the work into simpler (perhaps more digestible) steps on the SPEs.

After the Vector/SIMD Multimedia Extension code is working properly on the PPE, a strategy for parallelizing the algorithm across multiple SPEs can be

developed. This is often, but not always, a data-partitioning method. The effort might involve converting from Vector/SIMD Multimedia Extension intrinsics to SPU intrinsics, adding data-transfer and synchronization constructs, and tuning for performance. It might be useful to test the impact of various techniques, such as DMA double buffering, loop unrolling, branch elimination, alternative intrinsics, number of SPEs, and so forth. Debugging tools such as the static timing-analysis tool and the *IBM Full System Simulator* for the Cell Broadband Engine are available to assist this effort, as described in “Performance analysis” on page 112.

Alternatively, experienced Cell Broadband Engine programmers may prefer to skip the Vector/SIMD Multimedia Extension coding phase and go directly to SPU programming. In some cases, SIMD programming can be easier on an SPE than the PPE because of the SPE’s unified register file.

The earlier chapters in this tutorial describe the Vector/SIMD Multimedia Extension and SPU programming environments and some of their differences. Armed with knowledge of these differences, one can devise a strategy for developing code that is portable between the PPE and the SPEs. The strategy one should employ depends upon the type of instructions to be executed, the variety of vector data types, and the performance objectives. Solutions span the range of simple macro translation to full functional mapping.

## Code-mapping considerations

There are several challenges associated with mapping code designed for one instruction set and compiled for another instruction set. These including performance, unmappable constructs, limited size of LS, and equivalent precision, as described in this section.

### Code-mapping performance considerations

Simple remapping of low-level intrinsics can result in less-than-optimal performance, depending upon the intrinsics used.

Understanding the dynamic range of the remapping’s operands can reduce the performance impact of simple remapping.

### Unmappable constructs considerations

Differences in the processing of intrinsics make simple translation of certain intrinsics unmappable.

The unmappable SPU intrinsics include:

- stop and stopd
- conditional halt
- interrupt enable and disable
- move to and from status control and special-purpose registers
- channel instructions
- branch on external data

### Limited size of LS considerations

Vector/SIMD Multimedia Extension programs mapped to SPU programs might not fit within the LS of the SPE, either because the program is initially too big or because mapping expands the code.



## Equivalent precision considerations

The SPU instruction set does not fully implement the IEEE 754 single-precision floating-point standard (default rounding mode is *round to zero*, denormals are treated as zero, and there are no infinities or NaNs).

Therefore, floating-point results on an SPE may differ slightly from floating-point results using the PPE's PowerPC instruction set. In addition, all estimation intrinsics (for example, ceiling, floor, reciprocal estimate, reciprocal square root estimate, exponent estimate, and log estimate) do not have equivalent accuracy on the SPU and PPE PowerPC instruction sets.

However, the instructions in the PPE's Vector/SIMD Multimedia Extension have a *graphics rounding mode* that allows programs written with Vector/SIMD Multimedia Extension instructions to produce floating-point results that are equivalent in precision to those written in the SPU instruction set. In this Vector/SIMD Multimedia Extension mode, as in the SPU environment, the default rounding mode is *round to zero*, denormals are treated as zero, and there are no infinities or NaNs.

Details on the graphics rounding mode can be found in *Cell Broadband Engine, Programming Handbook*.

## Simple macro translation

For many programs, it is possible to use a simple macro translation strategy for developing code that is portable between the Vector/SIMD Multimedia Extension and SPU instruction sets.

The keys to simple macro translation are:

- *Use a Compatible Vector-Literal Construction Format* — The PPE Vector/SIMD Multimedia Extension and the SPE's SPU instruction set specifies two styles of constructing literal vectors: curly brace and parenthesis. Some compilers support both styles. A set of construction macros can be used to insulate programs from any differences in the tools.
- *Use Single-Token Vector Data Types* — The *C/C++ Language Extensions for Cell Broadband Engine Architecture* document specifies a set of single-token vector data types. Because these are single-token, the data types can be easily redefined by a preprocessor to the desired target processor. Additional single-token data types must be standardized for the unique Vector/SIMD Multimedia Extension data types. Table 36 lists the data types. See also:
  - Table 22 on page 73.
  - Table 23 on page 75.

Table 36. Vector/SIMD Multimedia Extension Single-Token Data Types

Vector Data Type	Single-Token Data Type
vector bool char	vec_bchar16
vector bool short	vec_bshort8
vector bool int	vec_bint4
vector pixel	vec_pixel8

- *Use Intrinsics that Map One-to-One* — Regardless of the technique used to provide portability, performance will be optimized if the operations map one-to-one between Vector/SIMD Multimedia Extension intrinsics and SPU intrinsics.



The SPU intrinsics that map one-to-one with Vector/SIMD Multimedia Extension (except for the specific intrinsics described in “Specific intrinsics” on page 66) are shown in Table 37.

The Vector/SIMD Multimedia Extension intrinsics that map one-to-one with SPU are shown in Table 38.

*Table 37. SPU Intrinsics with One-to-One Vector/SIMD Multimedia Extension Mapping*

<b>SPU Intrinsic</b>	<b>Vector/SIMD Multimedia Extension Intrinsic</b>	<b>For Data Types</b>
spu_add	vec_add	vector operands only, no scalar operands
spu_and	vec_and	vector operands only, no scalar operands
spu_andc	vec_andc	all
spu_avg	vec_avg	all
spu_cmpeq	vec_cmpeq	vector operands only, no scalar operands
spu_cmpgt	vec_cmpgt	vector operands only, no scalar operands
spu_convtf	vec_ctf	limited scale range (5 bits)
spu_convts	vec_cts	limited scale range (5 bits)
spu_convtu	vec_ctu	limited scale range (5 bits)
spu_extract	vec_extract	all
spu_genc	vec_addc	all
spu_insert	vec_insert	all
spu_madd	vec_madd	float only
spu_mulhh	vec_mule	all
spu_mulo	vec_mulo	halfword vector operands only, no scalar operands
spu_nmsub	vec_nmsub	float only
spu_nor	vec_nor	all
spu_or	vec_or	vector operands only, no scalar operands
spu_promote	vec_promote	all
spu_re	vec_re	all
spu_rl	vec_rl	vector operands only, no scalar operands
spu_rsqrte	vec_rsqrte	all
spu_sel	vec_sel	all
spu_splats	vec_splats	all
spu_sub	vec_sub	vector operands only, no scalar operands
spu_genb	vec_subc	vector operands only, no scalar operands
spu_xor	vec_xor	vector operands only, no scalar operands

*Table 38. Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping*

<b>Vector/SIMD Multimedia Extension Intrinsic</b>	<b>SPU Intrinsic</b>	<b>For Data Types</b>
vec_add	spu_add	halfwords, words, and floats only (not bytes)

Table 38. Vector/SIMD Multimedia Extension Intrinsic with One-to-One SPU Mapping (continued)

Vector/SIMD Multimedia Extension Intrinsic	SPU Intrinsic	For Data Types
vec_addc	spu_genc	all
vec_and	spu_and	all
vec_andc	spu_andc	all
vec_avg	spu_avg	unsigned chars only
vec_cmpeq	spu_cmpeq	all
vec_cmpgt	spu_cmpgt	all
vec_ctf	spu_convtf	all
vec_cts	spu_convts	all
vec_ctu	spu_convtu	all
vec_extract	spu_extract	all
vec_insert	spu_insert	all
vec_madd	spu_madd	all
vec_mulo	spu_mulo	halfwords only (not bytes)
vec_nmsub	spu_nmsub	all
vec_nor	spu_nor	all
vec_or	spu_or	all
vec_promote	spu_promote	all
vec_re	spu_re	all
vec_rl	spu_rl	halfwords and words only (not bytes)
vec_rsrte	spu_rsrte	all
vec_sel	spu_sel	all
vec_splats	spu_splats	all
vec_sub	spu_sub	halfwords, words, and floats only
vec_subc	spu_genb	all
vec_xor	spu_xor	all

**Note:** The toolchain contains headers files of overloaded C++ functions that can be used to assist in mapping or porting of Vector/SIMD Multimedia Extension intrinsics to SPU intrinsics, and vice-versa.

## Example 1: Euler particle-system simulation

This programming example illustrates many of the concepts discussed earlier in this chapter.

It can be found in the SDK under:

```
/opt/cell/sdk/src/tutorial/euler
```

This example — a simple Euler-based particle-system simulation — illustrates the following steps involved in coding for the Cell Broadband Engine:

1. Transform scalar code to vector code (SIMDize) for execution on the PPE's VXU.

2. Port the code for execution on the SPE's SPU unit.
3. Parallelize the code for execution across multiple SPEs.

A subsequent step — tuning the code for performance on the SPE — is covered in “Performance analysis” on page 112. The above steps are only one example of coding for the Cell Broadband Engine. The steps can be reordered or combined, depending upon the skill and comfort level of the programmer.

This example shows a particle-system simulation using numerical integration techniques to animate a large set of particles. Numerical integration is implemented using Euler's method of integration. It computes the next value of a function of time,  $F(t)$ , by incrementing the current value of the function by the product of the time step and the derivative of the function:

$$F(t + dt) = F(t) + dt * F'(t);$$

Our simple particle system consists of:

- An array of 3-D positions for each particle (`pos []`)
- An array of 3-D velocities for each particle (`vel []`)
- An array of masses for each particle (`mass []`)
- A force vector that varies over time (`force`)

This programming example is intended to illustrate programming concepts for the Cell Broadband Engine, and is not meant to be a physically realistic simulation. For example, it does not consider:

- how the time-variant force function and the time step,  $dt$ , is computed (instead, the example treats them as constants).
- particle collisions.

In addition, we assume that all 3-D vectors ( $x, y, z$ ) are expressed as 4-D homogeneous coordinates ( $x, y, z, 1$ ).

### Initial scalar code

The following code shows a C implementation of the Euler algorithm, implemented for a uniprocessor using scalar data. There are no intrinsics calls in this listing.

```
#define END_OF_TIME      10
#define PARTICLES       100000

typedef struct {
    float x, y, z, w;
} vec4D;

vec4D pos[PARTICLES];      // particle positions
vec4D vel[PARTICLES];     // particle velocities
vec4D force;               // current force being applied to the particles
float inv_mass[PARTICLES]; // inverse mass of the particles
float dt = 1.0f;          // step in time

int main()
{
    int i;
    float time;
    float dt_inv_mass;

    // For each step in time
    for (time=0; time<END_OF_TIME; time += dt) {
        // For each particle
        for (i=0; i<PARTICLES; i++) {
```

```

    // Compute the new position and velocity as acted upon by the force f.
    pos[i].x = vel[i].x * dt + pos[i].x;
    pos[i].y = vel[i].y * dt + pos[i].y;
    pos[i].z = vel[i].z * dt + pos[i].z;

    dt_inv_mass = dt * inv_mass[i];

    vel[i].x = dt_inv_mass * force.x + vel[i].x;
    vel[i].y = dt_inv_mass * force.y + vel[i].y;
    vel[i].z = dt_inv_mass * force.z + vel[i].z;
}
}
return (0);
}

```

### Step 1: SIMDize the code for execution on the PPE

There are multiple strategies for SIMDizing code for execution either on the PPE's VXU or on an SPE's SPU unit. The technique chosen depends upon the type of data being operated on and the interdependencies of the data computations.

There are several strategies to consider:

- *Let the Compiler Do It* — This will work effectively for some code samples (like this simple example), but it tends to be unsuccessful for more complicated code. Results will vary depending upon the algorithm, the language the code is expressed in, coding style, and capabilities of the compiler.
- *Array-of-Structures (AOS) Form* — This is the most common technique when the input data is naturally expressed as a vector (also call vector-across form). 3-D graphic applications express geometry as 3-component or 4-component vectors. These components naturally fit within a 4-component, single-precision floating-point vector. See also Figure 24 on page 63.
- *Structure-of-Arrays (SOA) Form* — In this form, you collect the individual elements of the natural vectors into separate arrays (also called parallel-array form). The code is then written as if it were to execute scalar instructions, but it will be executing SIMD instructions. This results in code that computes four single-precision floats results simultaneously. See also Figure 25 on page 64.
- *Hybrid Forms* — Often it is important that the input vector format remain unchanged. But SOA solutions are easier to code and more efficient than the AOS solutions. In this case, one can:
  - Input the data in its natural, AOS form.
  - Transform each data element on the fly into SOA form, using either the `vec_perm` (Vector/SIMD Multimedia Extension) or the `spu_shuffle` (SPU) intrinsic.
  - Perform computation using the SOA technique.
  - Translate each output back into its natural, AOS form.

Assuming the compiler auto-SIMDization is either unavailable or ineffective, you must adjust the data structures for efficient SIMD access. This decision cannot be made without also considering the SPE data-accessing method and the data-parallelization method. In addition, data should be aligned or padded for efficient quadword accesses, using the `aligned` attribute.

#### Step 1a: SIMDize in Array-of-Structures Form for Vector/SIMD Multimedia Extension

The following example shows how to SIMDize in the AOS form. Vector/SIMD Multimedia Extension intrinsics are used, and they can be identified by their prefix

vec\_. The algorithm assumes that the number of particles is a multiple of four. Special code must be included to handle the last number of particles that is not a multiple of four.

```
#define END_OF_TIME    10
#define PARTICLES     100000

typedef struct {
    float x, y, z, w;
} vec4D;
vec4D pos[PARTICLES] __attribute__((aligned(16)));
vec4D vel[PARTICLES] __attribute__((aligned(16)));
vec4D force __attribute__((aligned(16)));
float inv_mass[PARTICLES] __attribute__((aligned(16)));
float dt __attribute__((aligned(16))) = 1.0f;

int main()
{
    int i;
    float time;
    float dt_inv_mass __attribute__((aligned(16)));
    vector float dt_v, dt_inv_mass_v;
    vector float *pos_v, *vel_v, force_v;
    vector float zero = (vector float){0.0f, 0.0f, 0.0f, 0.0f};

    pos_v = (vector float *)pos;
    vel_v = (vector float *)vel;
    force_v = *((vector float *)&force);

    // Replicate the variable time step across elements 0-2 of
    // a floating point vector. Force the last element (3) to zero.
    dt_v = vec_sld(vec_splat(vec_lde(0, &dt), 0), zero, 4);

    // For each step in time
    for (time=0; time<END_OF_TIME; time += dt) {
        // For each particle
        for (i=0; i<PARTICLES; i++) {
            // Compute the new position and velocity as acted upon by the force f.
            pos_v[i] = vec_madd(vel_v[i], dt_v, pos_v[i]);

            dt_inv_mass = dt * inv_mass[i];
            dt_inv_mass_v = vec_splat(vec_lde(0, &dt_inv_mass), 0);

            vel_v[i] = vec_madd(dt_inv_mass_v, force_v, vel_v[i]);
        }
    }
    return (0);
}
```

### Step 1b: : SIMDize in Structure-of-Arrays Form for Vector/SIMD Multimedia Extension

The following example shows how to SIMDize in the SOA form. As in Step 1a, the algorithm assumes that the number of particles is a multiple of 4.

```
#define END_OF_TIME    10
#define PARTICLES     100000

typedef struct {
    float x, y, z, w;
} vec4D;

// Separate arrays for each component of the vector.
vector float pos_x[PARTICLES/4], pos_y[PARTICLES/4], pos_z[PARTICLES/4];
vector float vel_x[PARTICLES/4], vel_y[PARTICLES/4], vel_z[PARTICLES/4];
vec4D force __attribute__((aligned(16)));
float inv_mass[PARTICLES] __attribute__((aligned(16)));
```

```

float dt = 1.0f;

int main()
{
    int i;
    float time;
    float dt_inv_mass __attribute__((aligned (16)));
    vector float force_v, force_x, force_y, force_z;
    vector float dt_v, dt_inv_mass_v;

    // Create a replicated vector for each component of the force vector.
    force_v = *(vector float *)&force;
    force_x = vec_splat(force_v, 0);
    force_y = vec_splat(force_v, 1);
    force_z = vec_splat(force_v, 2);

    // Replicate the variable time step across all elements.
    dt_v = vec_splat(vec_ldc(0, &dt), 0);

    // For each step in time
    for (time=0; time<END_OF_TIME; time += dt) {
        // For each particle
        for (i=0; i<PARTICLES/4; i++) {
            // Compute the new position and velocity as acted upon by the force f.
            pos_x[i] = vec_madd(vel_x[i], dt_v, pos_x[i]);
            pos_y[i] = vec_madd(vel_y[i], dt_v, pos_y[i]);
            pos_z[i] = vec_madd(vel_z[i], dt_v, pos_z[i]);

            dt_inv_mass = dt * inv_mass[i];
            dt_inv_mass_v = vec_splat(vec_ldc(0, &dt_inv_mass), 0);

            vel_x[i] = vec_madd(dt_inv_mass_v, force_x, vel_x[i]);
            vel_y[i] = vec_madd(dt_inv_mass_v, force_y, vel_y[i]);
            vel_z[i] = vec_madd(dt_inv_mass_v, force_z, vel_z[i]);
        }
    }
    return (0);
}

```

## Step 2: Port the PPE code for execution on the SPE

This step entails: (1) creating an SPE thread of execution on the PPE, (2) migrating the computation loops from Vector/SIMD Multimedia Extension intrinsics to SPU intrinsic, and finally (3) adding DMA transfers to move data in and out of the SPE's local store (LS).

We assume that the particle data structures cannot be restructured into SOA form. Therefore, we use Step 1a from the previous section (the AOS form). SPU intrinsics are used, and they can be identified by their prefix `spu_`.

Moving the code from the PPE to the SPE requires:

- Creating a control-structure, called parameter context, that defines the parameters to be computed on the SPE. This includes pointers to the particle array data, current force information, and so forth. The pointer to the context control-structure defined in the PPE is passed to the SPE thread by using the parameter passing mechanism in `spe_create_thread`. Alternatively, this information could have been passed via the mailbox.
- Porting the computation for execution on the SPE. The complexity of this operation depends upon the types of data and types of intrinsics used. For this case, some of the intrinsics only require a simple name translation (for example, `vec_madd` to `spu_madd`). The translation of the scalar values is a little more extensive.

- Adding an additional looping construct to partition the data arrays into smaller blocks. This is required because all the data does not fit within the SPE's local store.
- Adding DMA transfers to move data in and out of the SPE's local store.

#### particle.h:

```
#define END_OF_TIME    10
#define PARTICLES     100000

typedef struct {
    float x, y, z, w;
} vec4D;

typedef struct {
    int particles;           // number of particles to process
    vector float *pos_v;    // pointer to array of position vectors
    vector float *vel_v;    // pointer to array of velocity vectors
    float *inv_mass;        // pointer to array of mass vectors
    vector float force_v;   // force vector
    float dt;               // current step in time
} context;
```

#### PPE Makefile:

```
#####
#                               Subdirectories
#####

DIRS := spu

#####
#                               Target
#####

PROGRAM_ppu := euler_spe

#####
#                               Local Defines
#####

IMPORTS      := spu/lib_particle_spu.a -lspe2 -lpthread

#####
#                               make.footer
#####

ifdef CELL_TOP
    include $(CELL_TOP)/builddutils/make.footer
else
    include ../../../../builddutils/make.footer
endif
```

#### PPE Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
#include <pthread.h>
#include "particle.h"

vec4D pos[PARTICLES] __attribute__((aligned (128)));
vec4D vel[PARTICLES] __attribute__((aligned (128)));
vec4D force __attribute__((aligned (16)));
float inv_mass[PARTICLES] __attribute__((aligned (128)));
float dt = 1.0f;
```

```

extern spe_program_handle_t particle;

typedef struct ppu_thread_data {
    spe_context_ptr_t spe_ctx;
    pthread_t pthread;
    unsigned int entry;
    void *argp;
} ppu_thread_data_t;

void *ppu_thread_function(void *arg) {
    ppu_thread_data_t *datap = (ppu_thread_data_t *)arg;

    if (spe_context_run(datap->spe_ctx, &datap->entry, 0,
        datap->argp, NULL, NULL) < 0) {
        perror ("Failed running context\n");
        exit (1);
    }
    pthread_exit(NULL);
}

int main()
{
    ppu_thread_data_t data;
    parm_context ctx __attribute__ ((aligned (16)));

    ctx.particles = PARTICLES;
    ctx.pos_v = (vector float *)pos;
    ctx.vel_v = (vector float *)vel;
    ctx.force_v = *((vector float *)&force);
    ctx.inv_mass = inv_mass;
    ctx.dt = dt;

    /* Create a SPE context */
    if ((data.spe_ctx = spe_context_create (0, NULL)) == NULL) {
        perror ("Failed creating context");
        exit (1);
    }
    /* Load SPE program into the SPE context*/
    if (spe_program_load (data.spe_ctx, &particle)) {
        perror ("Failed loading program");
        exit (1);
    }
    /* Initialize context run data */
    data.entry = SPE_DEFAULT_ENTRY;
    data.argp = &ctx;
    /* Create pthread for each of the SPE contexts */
    if (pthread_create (&data.pthread, NULL, &ppu_thread_function, &data)) {
        perror ("Failed creating thread");
        exit (1);
    }
    /* Wait for the threads to complete */
    if (pthread_join (data.pthread, NULL)) {
        perror ("Failed joining thread\n");
        exit (1);
    }
    return (0);
}

```

### SPE Makefile:

```

#####
# Target
#####

PROGRAM_spu      := particle
LIBRARY_embed    := lib_particle_spu.a

```



```
#####
# Local Defines
#####

INCLUDE := -I ..

#####
# make.footer
#####

ifdef CELL_TOP
    include $(CELL_TOP)/buildutils/make.footer
else
    include ../../../../buildutils/make.footer
endif
```

### SPE Code:

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"

#define PARTICLES_PER_BLOCK          1024

// Local store structures and buffers.
volatile context ctx;
volatile vector float pos[PARTICLES_PER_BLOCK] __attribute__((aligned (128)));
volatile vector float vel[PARTICLES_PER_BLOCK] __attribute__((aligned (128)));
volatile float inv_mass[PARTICLES_PER_BLOCK] __attribute__((aligned (128)));

int main(unsigned long long spe_id, unsigned long long parm)
{
    int i, j;
    int left, cnt;
    float time;
    unsigned int tag_id;
    vector float dt_v, dt_inv_mass_v;

    /* Reserve a tag ID */
    tag_id = mfc_tag_reserve();

    spu_writtech(MFC_WrTagMask, -1);

    // Input parameter parm is a pointer to the particle context.
    // Fetch the context, waiting for it to complete.
    spu_mfcdma32((void *)&ctx, (unsigned int)parm, sizeof(context),
        tag_id, MFC_GET_CMD);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

    dt_v = spu_splats(ctx.dt);

    // For each step in time
    for (time=0; time<END_OF_TIME; time += ctx.dt) {
        // For each block of particles
        for (i=0; i<ctx.particles; i+=PARTICLES_PER_BLOCK) {
            // Determine the number of particles in this block.
            left = ctx.particles - i;
            cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;

            // Fetch the data - position, velocity, inverse_mass. Wait for DMA to
            // complete before performing computation.
            spu_mfcdma32((void *)(pos), (unsigned int)(ctx.pos_v+i), cnt *
                sizeof(vector float), tag_id, MFC_GET_CMD);
            spu_mfcdma32((void *)(vel), (unsigned int)(ctx.vel_v+i), cnt *
                sizeof(vector float), tag_id, MFC_GET_CMD);
            spu_mfcdma32((void *)(inv_mass), (unsigned int)(ctx.inv_mass+i), cnt *
```

```

        sizeof(float), tag_id, MFC_GET_CMD);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

    // Compute the step in time for the block of particles
    for (j=0; j<cnt; j++) {
        pos[j] = spu_madd(vel[j], dt_v, pos[j]);
        dt_inv_mass_v = spu_mul(dt_v, spu_splats(inv_mass[j]));
        vel[j] = spu_madd(dt_inv_mass_v, ctx.force_v, vel[j]);
    }

    // Put the position and velocity data back into main storage
    spu_mfcdma32((void *)pos), (unsigned int)(ctx.pos_v+i), cnt *
        sizeof(vector float), tag_id, MFC_PUT_CMD);
    spu_mfcdma32((void *)vel), (unsigned int)(ctx.vel_v+i), cnt *
        sizeof(vector float), tag_id, MFC_PUT_CMD);
    }
}
// Wait for final DMAs to complete before terminating SPE thread.
(void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
return (0);
}

```

### Step 3: Parallelize code for execution across multiple SPEs

The most common and practical method of parallelizing computation across multiple SPEs is to partition the data. This works well for applications with little or no data dependency.

In our example, we can partition the Euler integration of the particle equally among the available SPEs. If there are four available SPEs, then the first quarter of the particles is processed by the first SPE, the second quarter of the particles is processed by the second SPE, and so forth.

The SPE code for this step is the same as that in Step 2, so only the PPE code is shown below.

#### PPE Code:

```

#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
#include <pthread.h>
#include "particle.h"

#define MAX_SPE_THREADS 16

vec4D pos[PARTICLES] __attribute__((aligned (128)));
vec4D vel[PARTICLES] __attribute__((aligned (128)));
vec4D force __attribute__((aligned (16)));
float inv_mass[PARTICLES] __attribute__((aligned (128)));
float dt = 1.0f;

extern spe_program_handle_t particle;

typedef struct ppu_thread_data {
    spe_context_ptr_t spe_ctx;
    pthread_t pthread;
    unsigned int entry;
    void *argp;
} ppu_thread_data_t;

void *ppu_thread_function(void *arg) {
    ppu_thread_data_t *datap = (ppu_thread_data_t *)arg;

    if (spe_context_run(datap->spe_ctx, &datap->entry, 0, datap->argp, NULL,
        NULL) < 0)
        {

```

```

        perror ("Failed running context\n");
        exit (1);
    }
    pthread_exit(NULL);
}

int main()
{
    int i, offset, count, spe_threads;
    ppu_thread_data_t datas[MAX_SPE_THREADS];
    parm_context ctxs[MAX_SPE_THREADS] __attribute__((aligned (16)));

    /* Determine the number of SPE threads to create */
    spe_threads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
    if (spe_threads > MAX_SPE_THREADS) spe_threads = MAX_SPE_THREADS;

    /* Create multiple SPE threads */
    for (i=0, offset=0; i<spe_threads; i++, offset+=count) {
        /* Construct a parameter context for each SPE. Make sure
         * that each SPEs (excluding the last) particle count is a multiple
         * of 4 so that inv_mass context pointer is always quadword aligned.
         */
        count = (PARTICLES / spe_threads + 3) & ~3;
        ctxs[i].particles = (i==(SPE_THREADS-1)) ? PARTICLES - offset : count;
        ctxs[i].pos_v = (vector float *)&pos[offset];
        ctxs[i].vel_v = (vector float *)&vel[offset];
        ctxs[i].force_v = *((vector float *)&force);
        ctxs[i].inv_mass = &inv_mass[offset];
        ctxs[i].dt = dt;

        /* Create SPE context */
        if ((datas[i].spe_ctx = spe_context_create (0, NULL)) == NULL) {
            perror ("Failed creating context");
            exit (1);
        }
        /* Load SPE program into the SPE context */
        if (spe_program_load (datas[i].spe_ctx, &particle)) {
            perror ("Failed loading program");
            exit (1);
        }
        /* Initialize context run data */
        datas[i].entry = SPE_DEFAULT_ENTRY;
        datas[i].argp = &ctxs[i];

        /* Create pthread for each of the SPE contexts */
        if (pthread_create (&datas[i].pthread, NULL, &ppu_thread_function,
            &datas[i])){
            perror ("Failed creating thread");
        }
    }

    /* Wait for all the SPE threads to complete.*/
    for (i=0; i<spe_threads; i++) {
        if (pthread_join (datas[i].pthread, NULL)) {
            perror ("Failed joining thread");
            exit (1);
        }
    }

    return (0);
}

```

Now that the program has been migrated to the SPEs, you can analyze and tune its performance. This is discussed in “Performance analysis” on page 112.

---

## Performance analysis

After a Cell Broadband Engine program executes without errors on the PPE and the SPEs, optimization through parameter-tuning can begin.

Programmers typically tune for performance using algorithmic methods. This is important for SPE programming also. But equally important for SPE programming is performance tuning through the elimination of stalls. There are two forms of stalls to consider:

- instruction dependency stalls, and
- data stalls.

Instruction stalls can be analyzed statically or dynamically.

## Performance issues

Two software tools are available in the SDK to assist in measuring the performance of programs: the *spu-timing* static timing analyzer, and the *IBM Full System Simulator for the Cell Broadband Engine*.

The *spu-timing* analyzer performs a static timing analysis of a program by annotating its assembly instructions with the instruction-pipeline state. This analysis is useful for coarsely spotting dual-issue rates (odd and even pipeline use) and assessing what program sections may be experiencing instruction-dependency and data-dependency stalls. It is useful, for example, for determining whether or not dependencies might be mitigated by unrolling, or whether reordering of instructions or better placement of no-ops will improve the dual-issue behavior in a loop. However, static analysis outputs typically do not provide numerical performance information about program execution. Thus, it cannot report anything definitive about cycle counts, branches taken or not taken, branches hinted or not hinted, DMA transfers, and so forth.

The *IBM Full System Simulator for the Cell Broadband Engine* performs a dynamic analysis of program execution. It is available in the SDK. Any part of a program, from a single line to the entire program, can be studied. Performance numbers are provided for:

- Instruction histograms (for example, branch, hint, and prefetch)
- Cycles per instruction (CPI)
- Single-issue and dual-issue rates
- Stall statistics
- Register use

The output of the IBM Full System Simulator for the Cell Broadband Engine can be a text listing or a graphic plot.

## Example 1: Tuning SPE performance with static and dynamic timing analysis

### Static analysis of SPE threads

The listing below shows an *spu-timing* static timing analysis for the inner loop of the SPE code.

The SPE code is shown in “Step 2: Port the PPE code for execution on the SPE” on page 106, the Euler Particle-System Simulation example. This listing shows significant dependency stalls (indicated by the “-”) and poor dual-issue rates. The

inner loop has an instruction mix of eight even-pipeline (pipe 0) instructions and ten odd-pipeline (pipe 1) instructions. Therefore, any program changes that minimize data dependencies will improve dual-issue rates and lower the cycle per instruction (CPI).

```

.L19:
0D                                     78      a      $49,$8,$10
1D 012                                789     lqx    $51,$6,$9
0D                                     89     ila    $47,66051
1D 0123                               89     lqx    $52,$6,$11
0 0                                   9      ai     $7,$7,-1
0 ----456789                          fma    $50,$51,$12,$52
1      -----012345                    stqx   $50,$6,$11
1          123456                        lqx    $48,$8,$10
0D              23                       ai     $8,$8,4
1D              234567                    lqa    $44,ctx+16
1              345678                    lqx    $43,$6,$9
1              ---7890                    rotqby $46,$48,$49
1              ---1234                    shufb  $45,$46,$46,$47
0              ---567890                  fm     $42,$12,$45
0d              -----123456            fma    $41,$42,$44,$43
1d              -----789012            stqx   $41,$6,$9
0D              89                       ai     $6,$6,16
.L39:
1D              8901                      brnz   $7,.L19

```

The character columns in the above static-analysis listing have the following meanings:

- Column 1 — The first column shows the pipeline that issued an instruction. Pipeline 0 is represented by 0 in the first column and pipeline 1 is represented by 1.
- Column 2 — The second column can contain a D, d, or “nothing”. A D signifies a successful dual-issue was accomplished by the two instructions listed in row-pairs. A d signifies a dual-issue was possible, but did not occur due to dependencies; for example, operands being in flight. If there is no entry in the second column, dual-issue could not be performed because the issue rules were not satisfied (for example, an even-pipeline instruction was fetched from an odd LS address or an odd-pipeline instruction was fetched from an even LS address). See “Pipelines and dual-issue rules” on page 53.
- Column 3 — The third column is always blank.
- Columns 4 through 53 — The next 50 columns represent clock cycles and are repeated as 0123456789 five times. A digit is displayed in these columns whenever the instruction executes during that clock cycle. Therefore, an <n>-cycle instruction will display <n> digits. Dependency stalls are flagged by a dash (“-”).
- Columns 54 and beyond — The remaining entries on the row are the assembly-language instructions or assembler-line addresses (for example, .L19) of the program’s assembly code.

Static-analysis timing files can be quickly interpreted by:

- Scanning the columns of digits. Small slopes (more horizontal) are bad. Large slopes (more vertical) are good.
- Looking for instructions with dependencies (those with dashes in the listing).
- Looking for instructions with poor dual-issue rates — either a d or “nothing” in column 2.

This information can be used to understand what areas of code are scheduled well and which are poorly scheduled.

## About SPU\_TIMING:

If you are using a Bash shell, you can set SPU\_TIMING as a shell variable by using the command `export SPU_TIMING=1`. You can also set SPU\_TIMING in the makefile and build the `.s` file by using the following statement:

```
SPU_TIMING=1 make foo.s
```

This creates the timing file for file `foo.c`. It sets the SPU\_TIMING variable only in the sub-shell of the makefile. It generates `foo.s` and then invokes `spu-timing` on `foo.s` to produce a `foo.s.timing` file.

Another way to invoke the performance tool is by entering one of the following statements in the command prompt:

```
SPU_TIMING=1 make foo.s
```

## Dynamic analysis of SPE threads

The listing below shows a dynamic timing analysis on the same SPE inner loop using the *IBM Full System Simulator for the Cell Broadband Engine*.

The results confirm the view of program execution from the static timing analysis:

- It shows poor dual-issue rates (7%) and large dependency stalls (65%), resulting in a overall CPI of 2.39.
- Most workloads should be capable of achieving a CPI of 0.7 to 0.9, roughly 3 times better than this.
- The number of used registers is 73, a 57.03% utilization of the full 128 register set.

```
SPU DD1.0
```

```
***
```

```
Total Cycle count          43120454
Total Instruction count     18068949
Total CPI                   2.39
***
Performance Cycle count    43120454
Performance Instruction count 18068949 (18062968)
Performance CPI             2.39 (2.39)
```

```
Branch instructions        1001990
Branch taken                1000007
Branch not taken           1983
```

```
Hint instructions          1973
Hint hit                   1000001
```

```
Contention at LS between Load/Store and Prefetch 2000986
```

```
Single cycle                12049144 ( 27.9%)
Dual cycle                   3006912 (  7.0%)
Nop cycle                    4003 (  0.0%)
Stall due to branch miss    17977 (  0.0%)
Stall due to prefetch miss   0 (  0.0%)
Stall due to dependency     28042299 (65.0%)
Stall due to fp resource conflict 0 (  0.0%)
Stall due to waiting for hint target 110 (  0.0%)
Stall due to dp pipeline    0 (  0.0%)
Channel stall cycle         0 (  0.0%)
SPU Initialization cycle    9 (  0.0%)
-----
Total cycle                  43120454 (100.0%)
```

Stall cycles due to dependency on each pipelines

FX2	5909
SHUF	6011772
FX3	1960
LS	7022608
BR	0
SPR	0
LNOP	0
NOP	0
FXB	0
FP6	15000050
FP7	0
FPD	0

The number of used registers are 73; the used ratio is 57.03

## Optimizations

To eliminate stalls and improve the CPI — and ultimately the performance — the compiler needs more instructions to schedule, so that the program does not stall. The SPE’s large register file allows the compiler or the programmer to unroll loops.

In our example program, there are no inter-loop dependencies (loop-carried dependencies), and our dynamic analysis shows that the register usage is fairly small, so moderately aggressive unrolling will not produce register spilling (that is, registers having to be written into temporary stack storage).

Most compilers can automatically unroll loops. Sometimes this is effective. But because automatic loop unrolling is not always effective, or because the programmer wants explicit control to manage the limited local store, this example shows how to manually unroll the loop.

The first pass of optimizations include:

- Unroll the loop to provide additional instructions for interleaving.
- Load DMA-buffer contents into local nonvolatile registers to eliminate volatile migration constraints.
- Eliminate scalar loads (the `inv_mass` variable).
- Eliminate extra multiplies of `dt*inv_mass` and `splat` the products after the SIMD multiply, instead of before the multiply.
- Interleave DMA transfers with computation by multibuffering the inputs and outputs to eliminate (or reduce) DMA stalls. These stalls are not reflected in the static and dynamic analyses. In the process of adding double buffering, the inner loop is moved into a function, so that the code need not be repeated.

The following SPE code results from these optimizations. Among the changes are the addition of a GET instruction with a barrier suffix (B), accomplished by the `spu_mfcdma32()` intrinsic with the `MFC_GETB_CMD` parameter. This GET is the barrier form of `MFC_GET_CMD`. The barrier form is used to ensure that previously computed results are put before the get for the next buffer’s data.

```

#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"

#define PARTICLES_PER_BLOCK          1024

// Local store structures and buffers.
volatile context ctx;
volatile vector float pos[2][PARTICLES_PER_BLOCK] __attribute__((aligned (128)));
volatile vector float vel[2][PARTICLES_PER_BLOCK] __attribute__((aligned (128)));
volatile vector float inv_mass[2][PARTICLES_PER_BLOCK/4] __attribute__((aligned (128)));

```

```

void process_buffer(int buffer, int cnt, vector float dt_v)
{
    int i;
    volatile vector float *p_inv_mass_v;
    vector float force_v, inv_mass_v;
    vector float pos0, pos1, pos2, pos3;
    vector float vel0, vel1, vel2, vel3;
    vector float dt_inv_mass_v, dt_inv_mass_v_0, dt_inv_mass_v_1,
        dt_inv_mass_v_2, dt_inv_mass_v_3;
    vector unsigned char splat_word_0 =
(vector unsigned char){0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
    vector unsigned char splat_word_1 =
(vector unsigned char){4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7};
    vector unsigned char splat_word_2 =
(vector unsigned char){8, 9,10,11, 8, 9,10,11, 8, 9,10,11, 8, 9,10,11};
    vector unsigned char splat_word_3 =
(vector unsigned char){12,13,14,15,12,13,14,15,12,13,14,15,12,13,14,15};

    p_inv_mass_v = (volatile vector float *)&inv_mass[buffer][0];
    force_v = ctx.force_v;

    // Compute the step in time for the block of particles, four
    // particle at a time.
    for (i=0; i<cnt; i+=4) {
        inv_mass_v = *p_inv_mass_v++;

        pos0 = pos[buffer][i+0];
        pos1 = pos[buffer][i+1];
        pos2 = pos[buffer][i+2];
        pos3 = pos[buffer][i+3];

        vel0 = vel[buffer][i+0];
        vel1 = vel[buffer][i+1];
        vel2 = vel[buffer][i+2];
        vel3 = vel[buffer][i+3];

        dt_inv_mass_v = spu_mul(dt_v, inv_mass_v);

        pos0 = spu_madd(vel0, dt_v, pos0);
        pos1 = spu_madd(vel1, dt_v, pos1);
        pos2 = spu_madd(vel2, dt_v, pos2);
        pos3 = spu_madd(vel3, dt_v, pos3);

        dt_inv_mass_v_0 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_0);
        dt_inv_mass_v_1 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_1);
        dt_inv_mass_v_2 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_2);
        dt_inv_mass_v_3 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_3);

        vel0 = spu_madd(dt_inv_mass_v_0, force_v, vel0);
        vel1 = spu_madd(dt_inv_mass_v_1, force_v, vel1);
        vel2 = spu_madd(dt_inv_mass_v_2, force_v, vel2);
        vel3 = spu_madd(dt_inv_mass_v_3, force_v, vel3);

        pos[buffer][i+0] = pos0;
        pos[buffer][i+1] = pos1;
        pos[buffer][i+2] = pos2;
        pos[buffer][i+3] = pos3;

        vel[buffer][i+0] = vel0;
        vel[buffer][i+1] = vel1;
        vel[buffer][i+2] = vel2;
        vel[buffer][i+3] = vel3;
    }
}

```



```

int main(unsigned long long spe_id, unsigned long long argv)
{
    int buffer, next_buffer;
    int cnt, next_cnt, left;
    float time, dt;
    vector float dt_v;
    volatile vector float *ctx_pos_v, *ctx_vel_v;
    volatile vector float *next_ctx_pos_v, *next_ctx_vel_v;
    volatile float *ctx_inv_mass, *next_ctx_inv_mass;
    unsigned int tags[2];

    // Reserve a pair of DMA tag IDs
    tags[0] = mfc_tag_reserve();
    tags[1] = mfc_tag_reserve();

    // Input parameter argv is a pointer to the particle context.
    // Fetch the context, waiting for it to complete.
    spu_writecth(MFC_WrTagMask, 1 << tags[0]);
    spu_mfcdma32((void *)&ctx, (unsigned int)argv, sizeof(context), tags[0],
        MFC_GET_CMD);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

    dt = ctx.dt;
    dt_v = spu_splats(dt);

    // For each step in time
    for (time=0; time<END_OF_TIME; time += dt) {
        // For each double buffered block of particles
        left = ctx.particles;

        cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;

        ctx_pos_v = ctx.pos_v;
        ctx_vel_v = ctx.vel_v;
        ctx_inv_mass = ctx.inv_mass;

        // Prefetch first buffer of input data
        buffer = 0;
        spu_mfcdma32((void *)(&pos), (unsigned int)(ctx_pos_v), cnt *
            sizeof(vector float), tags[0], MFC_GETB_CMD);
        spu_mfcdma32((void *)(&vel), (unsigned int)(ctx_vel_v), cnt *
            sizeof(vector float), tags[0], MFC_GET_CMD);
        spu_mfcdma32((void *)(&inv_mass), (unsigned int)(ctx_inv_mass), cnt *
            sizeof(float), tags[0], MFC_GET_CMD);

        while (cnt < left) {
            left -= cnt;

            next_ctx_pos_v = ctx_pos_v + cnt;
            next_ctx_vel_v = ctx_vel_v + cnt;
            next_ctx_inv_mass = ctx_inv_mass + cnt;
            next_cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;

            // Prefetch next buffer so the data is available for computation on next
            // loop iteration.
            // The first DMA is barriered so that we don't GET data before the
            // previous iteration's data is PUT.
            next_buffer = buffer^1;

            spu_mfcdma32((void *)(&pos[next_buffer][0]), (unsigned int)(next_ctx_pos_v),
                next_cnt * sizeof(vector float), tags[next_buffer], MFC_GETB_CMD);
            spu_mfcdma32((void *)(&vel[next_buffer][0]), (unsigned int)(next_ctx_vel_v),
                next_cnt * sizeof(vector float), tags[next_buffer], MFC_GET_CMD);
            spu_mfcdma32((void *)(&inv_mass[next_buffer][0]), (unsigned int)
                (next_ctx_inv_mass), next_cnt * sizeof(float), tags[next_buffer],
                MFC_GET_CMD);
        }
    }
}

```

```

// Wait for previously prefetched data
spu_writetech(MFC_WrTagMask, 1 << tags[buffer]);
(void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

process_buffer(buffer, cnt, dt_v);

// Put the buffer's position and velocity data back into main storage
spu_mfcdma32((void *)&pos[buffer][0]), (unsigned int)(ctx_pos_v), cnt *
sizeof(vector float), tags[buffer], MFC_PUT_CMD);
spu_mfcdma32((void *)&vel[buffer][0]), (unsigned int)(ctx_vel_v), cnt *
sizeof(vector float), tags[buffer], MFC_PUT_CMD);

ctx_pos_v = next_ctx_pos_v;
ctx_vel_v = next_ctx_vel_v;
ctx_inv_mass = next_ctx_inv_mass;

buffer = next_buffer;
cnt = next_cnt;
}

// Wait for previously prefetched data
spu_writetech(MFC_WrTagMask, 1 << tags[buffer]);
(void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

process_buffer(buffer, cnt, dt_v);

// Put the buffer's position and velocity data back into main storage
spu_mfcdma32((void *)&pos[buffer][0]), (unsigned int)(ctx_pos_v), cnt *
sizeof(vector float), tags[buffer], MFC_PUT_CMD);
spu_mfcdma32((void *)&vel[buffer][0]), (unsigned int)(ctx_vel_v), cnt *
sizeof(vector float), tags[buffer], MFC_PUT_CMD);

// Wait for DMAs to complete before starting the next step in time.
spu_writetech(MFC_WrTagMask, 1 << tags[buffer]);
(void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
}

return (0);
}

```

## Static analysis of optimization

The listing below shows a *spu\_timing* static timing analysis for the optimized SPE thread (process\_buffer subroutine only).

```

.type process_buffer, @function

0D 0123
1D 012345
0D 12
1D 1234
0D 23
1D 2345
0D 34
1D 3456
0 45
0 56
0 67
0 78
0 89
0 90
0D 01
1D 0
0D 12
1D 1234
0 2345
0 34
0D 45

process_buffer:
shli $2,$3,10
lqa $19,ctx+16
ori $6,$3,0
shlqbyi $24,$4,0
cgti $3,$4,0
shlqbyi $18,$5,0
ila $4,inv_mass
fsmbi $21,0
ilhu $27,1029
ilhu $26,2057
ilhu $25,3085
ila $28,66051
a $20,$2,$4
iohl $27,1543
iohl $26,2571
lnop
iohl $25,3599
brz $3,.L7
shli $17,$6,14
ila $23,pos
ila $22,vel

```

1D	456789	hbra	.L10,.L5
1	5	lnop	
0	6	nop	\$127
		.L5:	
0D	78	ila	\$43,pos
1D	789012	lqd	\$41,0(\$20)
0D	89	ila	\$42,vel
1D	890123	lqx	\$40,\$17,\$23
0	90	a	\$6,\$17,\$43
0	01	a	\$7,\$17,\$42
0D	12	ai	\$21,\$21,4
1D	123456	lqd	\$39,16(\$6)
0D	23	ai	\$20,\$20,16
1D	234567	lqd	\$38,32(\$6)
0D	345678	fm	\$36,\$18,\$41
1D	345678	lqd	\$37,48(\$6)
0D	45	cgt	\$16,\$24,\$21
1D	456789	lqx	\$13,\$17,\$22
1	567890	lqd	\$34,16(\$7)
1	678901	lqd	\$14,32(\$7)
1	789012	lqd	\$15,48(\$7)
1	-9012	shufb	\$35,\$36,\$36,\$28
0D	012345	fma	\$32,\$13,\$18,\$40
1D	0123	shufb	\$33,\$36,\$36,\$27
0D	123456	fma	\$10,\$34,\$18,\$39
1D	1234	shufb	\$31,\$36,\$36,\$26
0D	234567	fma	\$11,\$14,\$18,\$38
1D	2345	shufb	\$30,\$36,\$36,\$25
0	345678	fma	\$8,\$15,\$18,\$37
0	456789	fma	\$29,\$35,\$19,\$13
0D	567890	fma	\$5,\$33,\$19,\$34
1D	5	lnop	
0D	678901	fma	\$12,\$31,\$19,\$14
1D	678901	stqx	\$32,\$17,\$23
0D	789012	fma	\$9,\$30,\$19,\$15
1D	789012	stqd	\$10,16(\$6)
1	890123	stqd	\$11,32(\$6)
1	901234	stqd	\$8,48(\$6)
0D	0	nop	\$127
1D	012345	stqx	\$29,\$17,\$22
0D	12	ai	\$17,\$17,64
1D	123456	stqd	\$5,16(\$7)
1	234567	stqd	\$12,32(\$7)
1	345678	stqd	\$9,48(\$7)
0D	4	nop	\$127
		.L10:	
1D	4567	brnz	\$16,.L5
		.L7:	
0D	5	nop	\$127
1D	5678	bi	\$1r

## Dynamic analysis of optimizations

The listing below shows a dynamic timing analysis on the *IBM Full System Simulator for the Cell Broadband Engine* for the optimized SPE thread (process buffer only). It shows that 78 registers are used, so the used percentage is 60.94.

```

SPU DD1.0
***
Total Cycle count          7134843
Total Instruction count    10602009
Total CPI                   0.67
***
Performance Cycle count    7134843
Performance Instruction count 10602009 (9839265)
Performance CPI             0.67 (0.73)

Branch instructions        253940

```

Branch taken	251967
Branch not taken	1973
Hint instructions	2952
Hint hit	250980

Contention at LS between Load/Store and Prefetch 6871

Single cycle	3815689 ( 53.5%)
Dual cycle	3011788 ( 42.2%)
Nop cycle	5898 ( 0.1%)
Stall due to branch miss	34655 ( 0.5%)
Stall due to prefetch miss	0 ( 0.0%)
Stall due to dependency	266732 ( 3.7%)
Stall due to fp resource conflict	0 ( 0.0%)
Stall due to waiting for hint target	72 ( 0.0%)
Stall due to dp pipeline	0 ( 0.0%)
Channel stall cycle	0 ( 0.0%)
SPU Initialization cycle	9 ( 0.0%)
-----	
Total cycle	7134843 (100.0%)

Stall cycles due to dependency on each pipelines

FX2	8808
SHUF	1971
FX3	5870
LS	32
BR	0
SPR	1
LNOP	0
NOP	0
FXB	0
FP6	250050
FP7	0
FPD	0

The number of used registers are 78, the used ratio is 60.94

The above static and dynamic timing analysis of the optimized SPE code reveals:

- Significant increase in dual-issue rate and reduction in dependency stalls. The static analysis shows that the `process_buffer` inner loop still contains a single-cycle stall and some instructions that are not dual-issued. Further performance improvements could likely be achieved by either more loop unrolling or software loop-pipelining.
- The number of instructions has decreased by 41% from the initial instruction count.
- The CPI has dropped from 2.39 to a more typical 0.73.
- The performance of the SPE code, measured in total cycle count, has gone from approximately 43 M cycles to 7 M cycles, an improvement of more than 6x. This improvement does not take into account the DMA latency-hiding (stall elimination) provided by double buffering.

For details about performance simulation, including examples of coding for simulations, see Chapter 5, "The simulator," on page 129. The *IBM Full System Simulator for the Cell Broadband Engine* described in that chapter supports performance simulation for a full system, including the MFCs, caches, bus, and memory controller. )

---

## General SPE programming tips

This section contains a short summary of general tips for optimizing the performance of SPE programs.

- *Local Store*
  - Design for the LS size. The LS holds up to 256 KB for the program, stack, local data structures, and DMA buffers. One can do a lot with 256 KB, but be aware of this size.
  - Use overlays (runtime download program kernels) to build complex function servers in the LS (see “SPE overlays” on page 127).
- *DMA Transfers*
  - Use SPE-initiated DMA transfers rather than PPE-initiated DMA transfers. There are more SPEs than the one PPE, and the PPE can enqueue only eight DMA requests whereas each SPE can enqueue 16.
  - Overlap DMA with computation by double buffering or multibuffering (see “DMA transfers” on page 83). Multibuffer code or (typically) data.
  - Use double buffering to hide memory latency.
  - Use fence command options to order DMA transfers within a tag group.
  - Use barrier command options to order DMA transfers within the queue.
- *Loops*
  - Unroll loops to reduce dependencies and increase dual-issue rates. This exploits the large SPU register file.
  - Compiler auto-unrolling is not perfect, but pretty good.
- *SIMD Strategy*
  - Choose an SIMD strategy appropriate for your algorithm. For example:
  - Evaluate array-of-structure (AOS) organization. For graphics vertices, this organization (also called or vector-across) can have more-efficient code size and simpler DMA needs, but less-efficient computation unless the code is unrolled.
  - Evaluate structure-of-arrays (SOA) organization. For graphics vertices, this organization (also called parallel-array) can be easier to SIMDize, but the data must be maintained in separate arrays or the SPU must shuffle AOS data into an SOA form.
  - Consider the effects of unrolling when choosing an SIMD strategy.
- *Load/Store*
  - Scalar loads and stores are slow, with long latency.
  - SPU only support quadword loads and stores.
  - Consider making scalars into quadword integer vectors.
  - Load or store scalar arrays as quadwords, and perform your own extraction and insertion to eliminate load and store instructions.
- *Branches*
  - Eliminate nonpredicted branches.
  - Use feedback-directed optimization.
  - Use the `__builtin_expect` language directive when you can explicitly direct branch prediction.
- *Multiplies*
  - Avoid integer multiplies on operands greater than 16 bits in size. The SPU supports only a “16-bit x16-bit multiply”. A “32-bit multiply” requires five instructions (three 16-bit multiplies and two adds).

- Keep array elements sized to a power-of-2 to avoid multiplies when indexing.
- Cast operands to unsigned short prior to multiplying. Constants are of type `int` and also require casting. Use a macro to explicitly perform 16-bit multiplies. This can avoid inadvertent introduction of signed extends and masks due to casting.
- *Pointers*
  - Use the PPE's load/store with update instructions. These allow sequential indexing through an array without the need of additional instructions to increment the array pointer.
  - For the SPEs (which do not support load/store with update instructions), use the d-form instructions to specify an immediate offset from a base array pointer.
- *Dual-Issue*
  - Choose intrinsics carefully to maximize dual-issue rates or reduce latencies.
  - Dual issue will occur if a `pipe-0` instruction is even-addressed, a `pipe-1` instruction is odd-addressed, and there are no dependencies (operands are available).
  - Code generators use nops to align instructions for dual-issue.
  - Use software pipeline loops to improve dual-issue rates.

---

## Chapter 4. Programming models

On any processor, coding optimizations are achieved by exploiting the unique features of the hardware. In the case of the Cell Broadband Engine, the large number of SPEs, their large register file, and their ability to hide main-storage latency with concurrent computation and DMA transfers support many interesting programming models.

With the computational efficiency of the SPEs, software developers can create programs that manage dataflow as opposed to leaving dataflow to a compiler or to later optimizations.

Many of the unique features of the SPE are handled by the compiler, although programmers looking for the best performance can take advantage of the features independently of the compiler. It is almost never necessary to program the SPE in assembly language. C intrinsics provide a convenient way to program the efficient movement and buffering of data.

“Application partitioning” on page 13 introduced some concepts for application programming.

This chapter introduces these seven types of *programming models*:

- Function-Offload Model,
- Device-Extension Model,
- Computation-Acceleration Model,
- Streaming Model,
- Shared-Memory Multiprocessor Model,
- Asymmetric-Thread Runtime Model,
- User-Mode Thread Model.

---

### Function-Offload Model

In the Function-Offload Model, the SPEs are used as accelerators for performance-critical procedures.

This model is the quickest way to effectively use the Cell Broadband Engine with an existing application. In this model, the main application runs on the PPE and calls selected procedures to run on one or more SPEs.

The Function-Offload Model is sometimes called the Remote Procedure Call (RPC) Model. The model allows a PPE program to call a procedure located on an SPE as if it were calling a local procedure on the PPE. This provides an easy way for programmers to use the asynchronous parallelism of the SPEs without having to understand the low-level workings of the MFC DMA layer.

In this model, you identify which procedures should execute on the PPE and which should execute on the SPEs. The PPE and SPE source modules must be compiled separately, by different compilers.

## Remote procedure call

The Function Offload or Remote Procedure Call (RPC) Model is implemented using stubs as proxies.

A *method stub*, or simply *stub*, is a small piece of code used to stand in for some other code. The stub or proxy acts as a local surrogate for the remote procedure, hiding the details of server communication. The main code on the PPE contains a stub for each remote procedure on the SPEs. Each procedure on an SPE has a stub that takes care of running the procedure and communicating with the PPE.

When the program on the PPE calls a remote procedure, it actually calls that procedure's stub located on the PPE. The stub code initializes the SPE with the necessary data and code, packs the procedure's parameters, and sends a mailbox message to the SPE to start its stub procedure.

The SPE stub retrieves the parameters and executes the procedure locally on the SPE. The PPE program then retrieves the output parameters. Figure 27 shows an example of a program using this method.

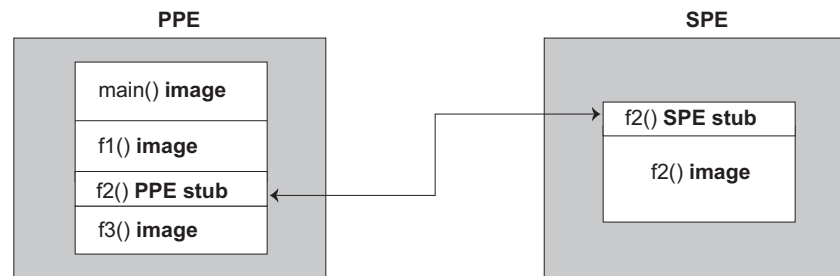


Figure 27. Example of the Function-Offload (or RPC) Model

---

## Device-Extension Model

The Device Extension Model is a special case of the Function-Offload Model in which the SPEs act like I/O devices.

SPEs can also act as intelligent front ends to an I/O device. Mailboxes can be used as command and response FIFOs between the PPE and SPEs.

The SPEs can interact with I/O devices because:

- all I/O devices are memory-mapped, and
- the SPEs DMA transfers support transfer sizes of a single byte.

I/O devices can use an SPE's signal-notification facility (described in "Signal notification" on page 59) to tell the SPE when commands complete.

When SPEs are used in the Device-Extension Model, they usually run privileged software that is part of the operating system. As such, this code is trusted and may be given access to privileged registers for a physical device. For example, a secure file system may be treated as a device. The operating system's device driver can be written to use the SPE for encryption and decryption and for responding to disk-controller requests on all file reads and writes to this virtual device.



---

## Computation-Acceleration Model

The *Computation-Acceleration Model* is an SPE-centric model that provides a smaller-grained and more integrated use of SPEs.

The model speeds up applications that use computation-intensive mathematical functions without requiring significant rewrite of the applications. Most computation-intensive sections of the application run on SPEs. The PPE acts as a control and system-service facility.

Multiple SPEs work in parallel. The work is partitioned manually by the programmer, or automatically by the compilers. The SPEs must efficiently schedule MFC DMA commands that move instructions and data.

This model either uses shared memory to communicate among SPEs, or it uses a message-passing model.

---

## Streaming model

In the *Streaming Model*, each SPE, in either a serial or parallel pipeline, computes data that streams through.

The PPE acts as a stream controller, and the SPEs act as stream-data processors. For the SPEs, on-chip load and store bandwidth exceeds off-chip DMA-transfer bandwidth by an order of magnitude. If each SPE has an equivalent amount of work, this model can be an efficient way to use the Cell Broadband Engine because data remains inside the Cell Broadband Engine as long as possible.

The PPE and SPEs support message-passing between the PPE, the processing SPE, and other SPEs.

Although the SDK does not include a formal streaming language, most of the programs written for the Cell Broadband Engine are likely to use the streaming model to some extent. For example, the Euler particle-system simulation (described in “Example 1: Euler particle-system simulation” on page 102) implements the streaming model. This particle-system simulation contains a computational kernel that streams packets of data through the kernel for each step in time.

---

## Shared-Memory Multiprocessor Model

The Cell Broadband Engine can be programmed as a shared-memory multiprocessor, using two different instruction sets. The SPEs and the PPE fully interoperate in a cache-coherent *Shared-Memory Multiprocessor Model*.

All DMA operations in the SPEs are cache-coherent. Shared-memory load instructions are replaced by DMA operations from shared memory to local store (LS), followed by a load from LS to the register file. The DMA operations use an effective address that is common to the PPE and all the SPEs. Shared-memory store instructions are replaced by a store from the register file to the LS, followed by a DMA operation from LS to shared memory.

The SPE's DMA lock-line commands provide the equivalent of the PowerPC Architecture atomic-update primitives (load with reservation and store conditional).

A compiler or interpreter could manage part of the LS as a local cache for instructions and data obtained from shared memory.

---

## Asymmetric-Thread Runtime Model

Threads can be scheduled to run on either the PPE or on the SPEs, and threads interact with one another in the same way they do in a conventional symmetric multiprocessor. The *Asymmetric-Thread Runtime Model* extends thread task models and lightweight task models to include the different instruction sets supported by the PPE and SPE.

Scheduling policies are applied to the PPE and SPE threads to optimize performance. Although preemptive task-switching is supported on SPEs for debugging purposes, there is a runtime performance and resource-allocation cost. FIFO run-to-completion models, or lightweight cooperatively-yielding models, can be used for efficient task-scheduling.

A single SPE can run only one thread at a time; it cannot support multiple simultaneous threads.

The Asymmetric-Thread Runtime Model is flexible and supports all of the other programming models described in this chapter. Any program that explicitly calls `spe_context_create` and `spe_context_run` is an example of the Asymmetric-Thread Runtime Model. For an example of invoking SPE threads, see “Creating threads for the SPEs” on page 38.

This is the fundamental model provided by the SDK’s SPU Runtime Management Library, and it is identified by user threads (both PPE and SPE) running on the Cell Broadband Engine’s heterogeneous processing complex.

---

## User-mode thread model

The *User-Mode Thread Model* refers to one SPE thread managing a set of user-level functions running in parallel.

The user-level functions are called *microthreads* (and also *user threads* and *user-level tasks*). The SPE thread is supported by the operating system. The microthreads are created and supported by user software; the operating system is not involved. However, the set of microthreads can run across a set of SPUs.

The SPU application schedules tasks in shared memory, and the tasks are processed by available SPUs. For example, in game programming, the tasks can refer to scene objects that need updating. Microthreads can complete at any time, and new microthreads can be spawned at any time.

One advantage of this programming model is that the microthreads, running on a set of SPUs under the control of an SPE thread, have predictable overhead. A single SPE cannot save and restore the MFC commands queues without assistance from the PPE.

---

## Cell application frameworks

The complexity of implementing many of these programming models is significantly reduced by using *application frameworks*.

One such framework is the *Accelerated Library Framework (ALF)*. This framework provides a set of functions to help programmers solve data parallel computation problems on hybrid systems using a single-program-multiple-data (SPMD) programming style. Features include architecturally independent data transfer management, parallel task management, multi-buffering, and data partitioning.

For further details about the ALF, refer to the *Accelerated Library Framework, User's Guide*.

---

## SPE overlays

When code does not fit in an SPE's local store, *overlays* can be useful.

An overlay is SPU code that is dynamically loaded and executed by a running SPU program. It cannot be independently loaded or run on an SPE. SPE overlays allow the programmer to manage SPU code in a modular fashion.

The specific SPU code that is needed at runtime is *dynamically* loaded.

Additional information on developing code with overlays is provided in the *Software Development Kit, Programmer's Guide*.



## Chapter 5. The simulator

The IBM Full System Simulator for the Cell Broadband Engine is a generalized simulator that can be configured to simulate a broad range of full-system configurations.

The Full-System Simulator is withdrawn from the SDK. You can download it from the alphaWorks Web site: <http://www.alphaworks.ibm.com/tech/cellsystemsim>

The simulator supports full *functional simulation*, including the:

- PPE
- SPEs
- MFCs
- PPE caches
- bus
- memory controller

It can simulate and capture many levels of operational details on instruction execution, cache and memory subsystems, interrupt subsystems, communications, and other important system functions. It also supports some *cycle-accurate simulation* (performance or timing simulation).

Figure 28 shows the simulation stack. The simulator is part of the software development kit (SDK).

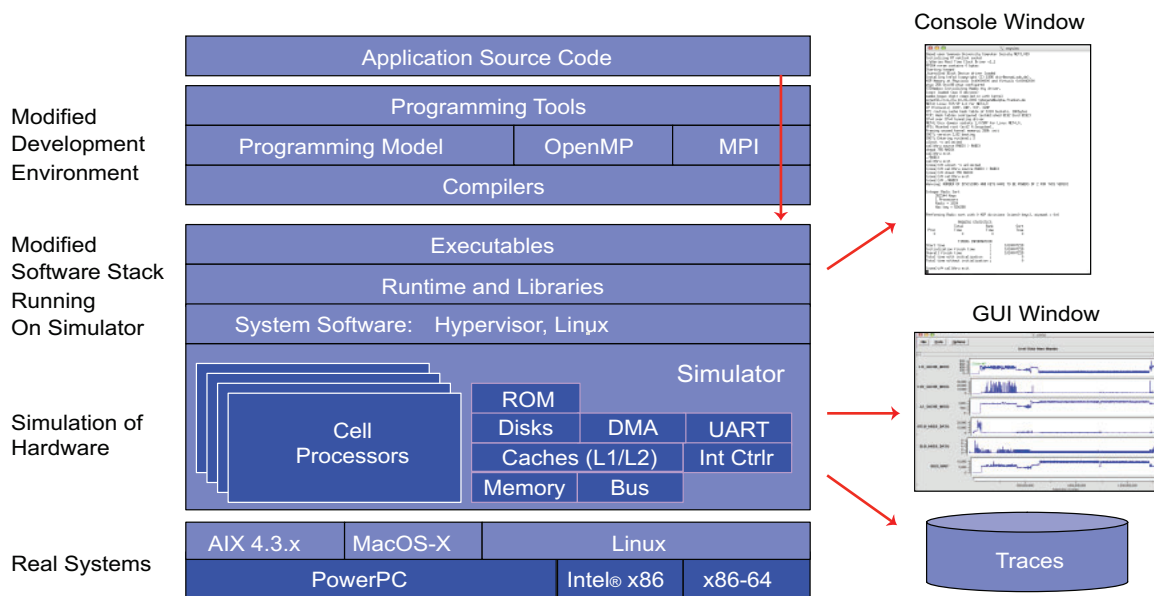


Figure 28. Simulation stack

If accurate timing and cycle-level simulation are not required, the simulator can be used in its *functional-only mode*, running as a debugger to test the functions and features of a program.

If cycle-level analysis is required, it can be used in *performance simulation* (or *timing simulation*) mode, to get accurate performance analyses.

Simulator configurations are extensible and can be modified using Tool Command Language (Tcl) commands to produce the type and level of analysis required.

The simulator itself is a general tool that can be configured for a broad range of microprocessors and hardware simulations. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development and analysis.

---

## Simulator basics

This section provided as overview of *IBM Full System Simulator for the Cell Broadband Engine*.

Additional details can be found in the simulator's documentation installed in `/opt/ibm/systemsim-cell/doc`.

### Operating-system modes

The simulator has two modes of operation, with regard to operating systems: *Linux mode*, and *standalone mode*.

#### Linux mode

In Linux mode, after the simulator is configured and loaded, the simulator boots the Linux operating system on the simulated system.

At runtime, the operating system is simulated along with the running programs. The simulated operating system takes care of all the system calls, just as it would in a non-simulation (real) environment.

#### Standalone mode

In *standalone mode*, the application is loaded without an operating system.

Standalone applications are user-mode applications that are normally run on an operating system. On a real system, these applications rely on the operating system to perform certain tasks, including loading the program, address translation, and system-call support. In standalone mode, the simulator provides some of this support, allowing applications to run without having to first boot an operating system on the simulator.

There are, however, limitations that apply when building an application to be loaded and run by the simulator without an operating system. Typically, the operating system provides address-translation support.

- Since an operating system is not present in this mode, the simulator loads executables without address translation, so that the *effective address* is the same as the *real address*. Therefore, all addresses referenced in the executable must be *valid real addresses*.
- If the simulator has been configured with 64 MB of memory, all addresses must fit in the range of `X'0'` to `X'3FFFFFF'`.

### Interacting with the simulator

There are two ways to interact with the simulator. Firstly, by issuing commands to the *simulated system*. Secondly, by issuing commands to the *simulator*.

The simulated system is the Linux environment on top of the simulated Cell Broadband Engine, where you run and debug programs. You interact with it by entering commands at the Linux command prompt, in the *console window*. The console window is a Linux shell of the simulated Linux operating system.

You can also control the simulator itself, configuring it to do such tasks as collect and display performance statistics on particular SPEs, or set breakpoints in code. These commands are entered at the simulator command line in the *simulator command window*, or using the equivalent actions in the graphical user interface (GUI).

The GUI is a graphical means of interacting with the simulator. The GUI is described in “Graphical User Interface” on page 132.

Figure 29 shows the simulator windows, and the layers with which they communicate.

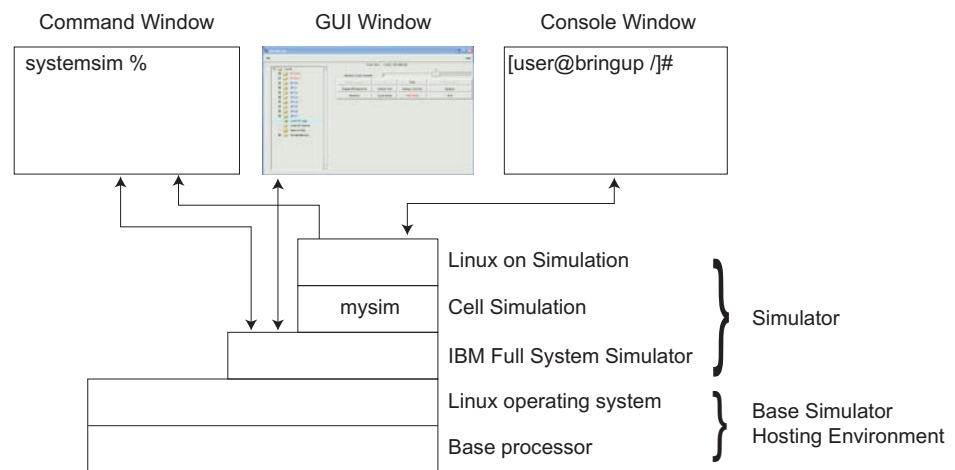


Figure 29. Simulator structures and screens

## Command-line interface

To start the simulator in *command-line mode*, enter the following command:  
`PATH=/opt/ibm/systemsim-cell/bin:$PATH; systemsim.`

This command starts the simulator, which initializes the simulation and displays the prompt:

```
systemsim %
```

The window displaying the simulator prompt is the command window. While starting the simulation, the simulator creates the console window, which is initially labeled `UART0` in the window's title bar.

All commands must be entered at the prompt in the command window (that is, the window in which the simulator was started). Some of the important commands are shown in Table 39 on page 132.

Table 39. Important Commands for the IBM Full System Simulator for the Cell Broadband Engine

Simulator Command	Meaning
quit	Closes the simulation and exits the simulator.
help	Displays a list of the available simulator commands.
mysim go	Starts or continues the simulation. The first time it is issued, the simulator boots the Linux operating system on the simulation.
mysim spu <i>n</i> set model <i>mode</i>	Sets SPEn into model mode, where <i>n</i> is a value from 0 to 7 and <i>mode</i> is either pipeline or instruction.
mysim spu <i>n</i> display statistics	Displays to the simulator command window, the performance analysis statistics collected on SPEn, where <i>n</i> is a value from 0 to 7. Statistics are only collected when the SPE is executing in pipeline mode.

The simulator prompt is displayed in the command window when the simulation is stopped, or paused. When the simulation is running, the command window, instead, displays a copy of the output to the console window and simulation-cycle information every few seconds, and the prompt is not available.

To stop the simulation and get back the prompt — use the Ctrl-c key sequence. This will stop the simulation, and the prompt will reappear.

---

## Graphical User Interface

The simulator's GUI offers a visual display of the state of the simulated system, including the PPE and the eight SPEs.

You can view the values of the registers, memory, and channels, as well as viewing performance statistics. The GUI also offers an alternate method of interacting with the simulator. Figure 30 on page 133 shows the main GUI window that appears when the GUI is launched.



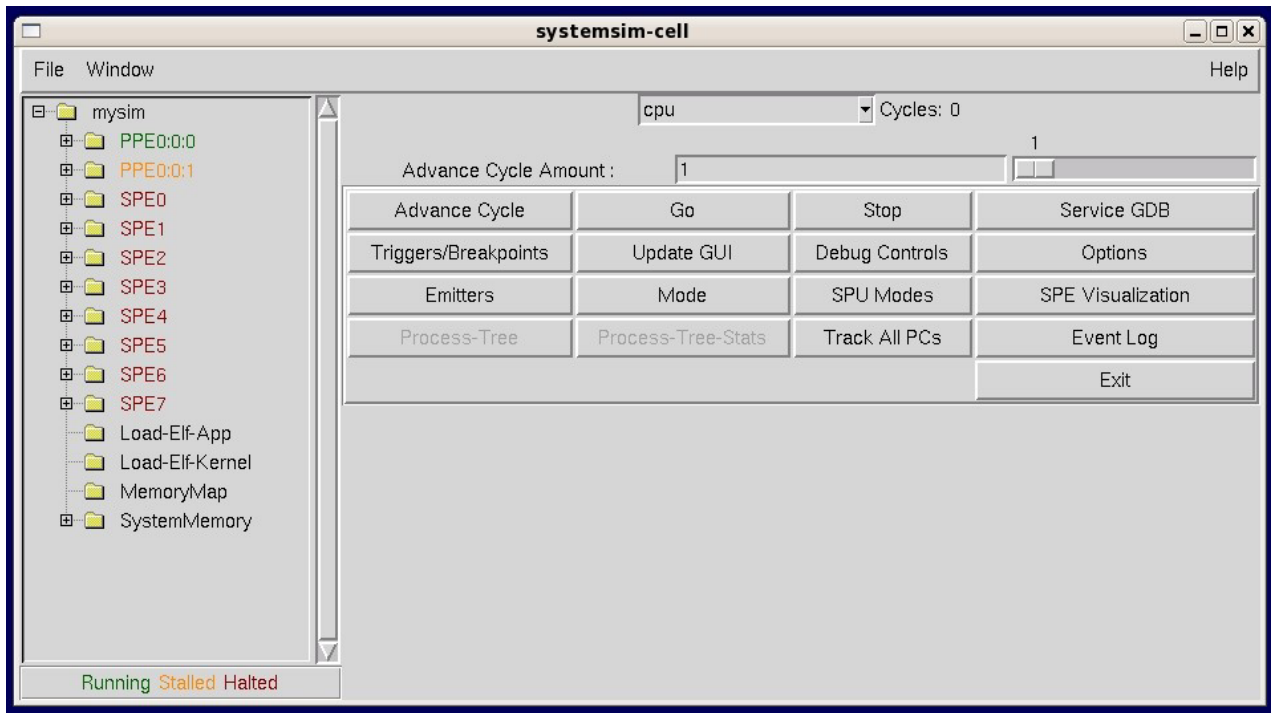


Figure 30. Main Graphical User Interface for the simulator

The main GUI window has two basic areas:

- The vertical panel on the left.
- The rows of buttons on the right.

The vertical panel represents the simulated system and its components. The rows of buttons are used to control the simulator.

To start the GUI from the Linux run directory, enter:

```
PATH=/opt/ibm/systemsin-cell/bin:$PATH; systemsim -g
```

The simulator will then configure the simulator as a Cell Broadband Engine and display the main GUI window, labeled with the name of the application program. When the GUI window first appears, click the Go button to boot the Linux operating system.

For a detailed description of starting the simulator and running a program see “Running the program in the simulator” on page 44.

## The simulation panel

When the main GUI window first appears, the vertical panel contains a single folder labeled `mysim`.

To see the contents of `mysim`, click on the plus sign (+) in front of the folder icon. When the folder is expanded, you can see its contents. These include

- a PPE (labelled `PPE0:0:0` and `PPE0:0:1`,
- the two threads of the PPE),
- eight SPEs (`SPE0... SPE7`).

The folders representing the processors can be further expanded to show the viewable objects and the options and actions available. Figure 31 shows the vertical panel with several of the processor folders expanded.

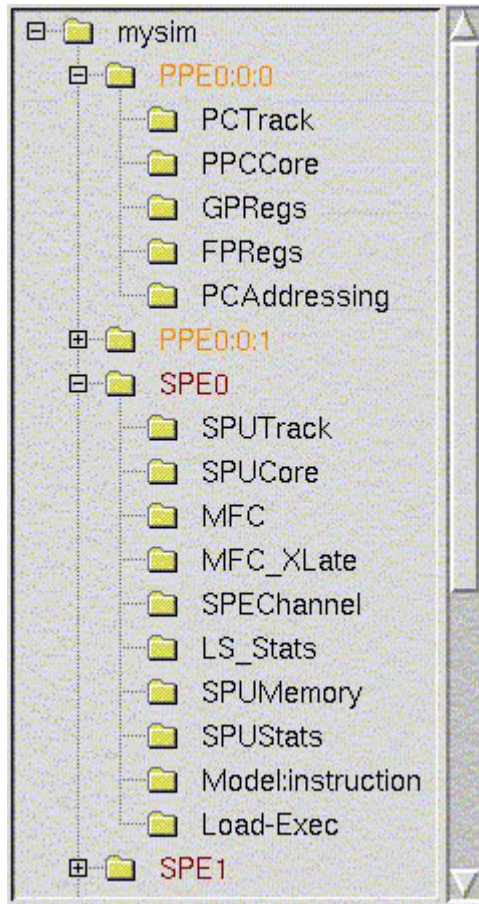


Figure 31. Project and processor folders

## PPE components

There are five PPE components visible in the expanded PPE folder.

The five visible PPE components are:

- PCTrack
- PCCore
- GPRs
- FPRs
- PCAddressing

Double-clicking a folder icon brings up a window displaying the program-state data. Several of the available windows are shown in the figures provided here.

The general-purpose registers (GPRs) and the floating-point registers (FPRs) can be viewed separately by double-clicking on the GPRs and the FPRs folders respectively.

Figure 32 shows the GPR window, and Figure 33 shows the FPR window. As data changes in the simulated registers, the data in the windows is updated and registers that have changed state are highlighted.

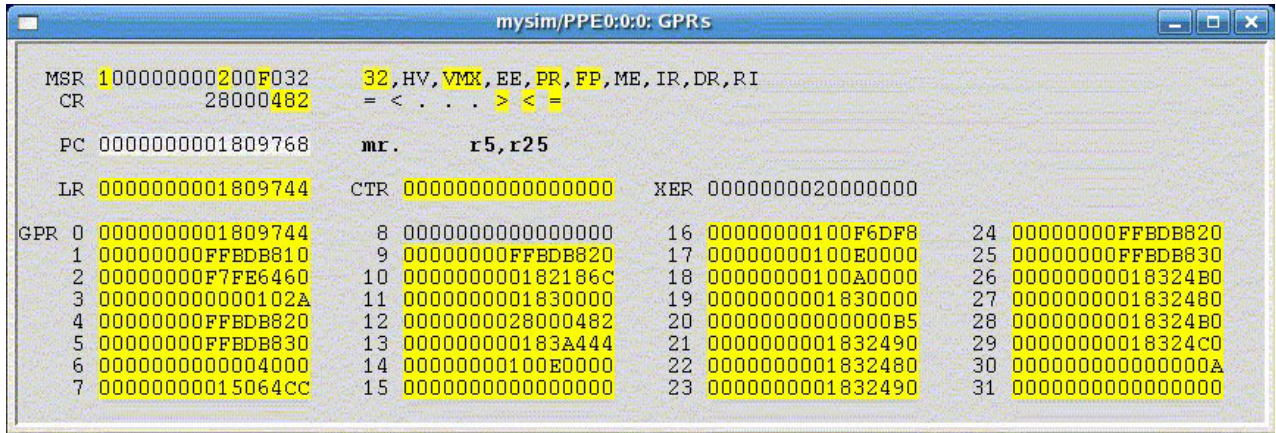


Figure 32. PPE General-Purpose Registers window

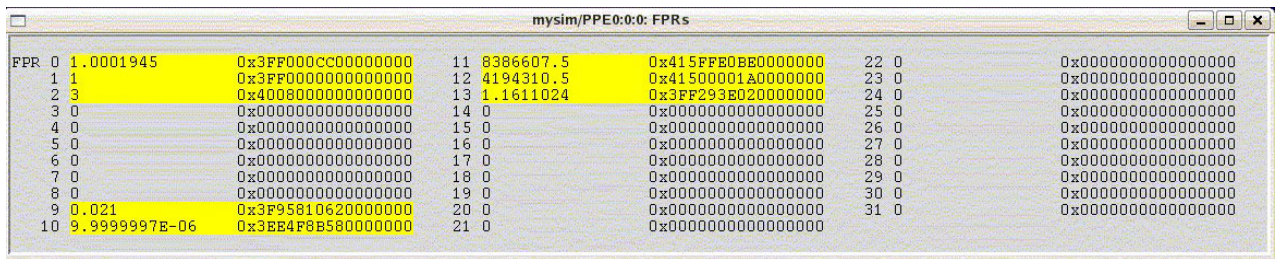


Figure 33. PPE Floating-Point Registers window

The PPE Core window (PCCore) shows the contents of all the registers of the PPE, including the Vector/SIMD Multimedia Extension registers. Figure 34 on page 136 shows the PPE Core window.



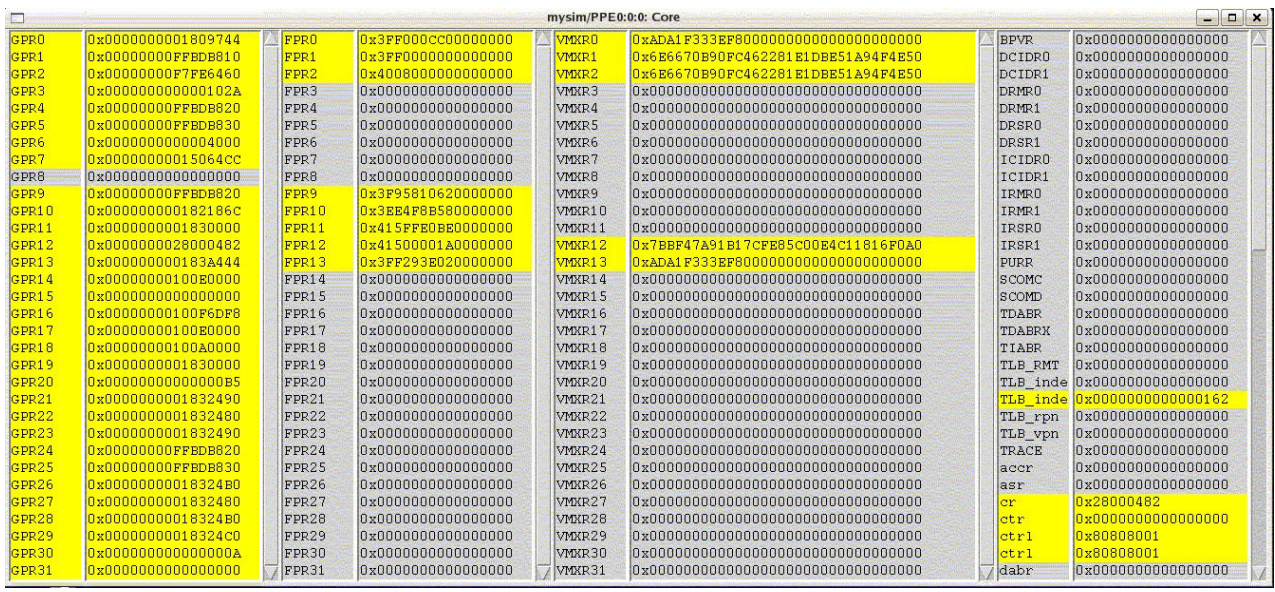


Figure 34. PPE Core window

## SPE components

The SPE folders (SPE0 ... SPE7) each have ten sub-items.

Five of the sub-items represent windows that show data in the registers, channels, and memory:

- SPUTrack
- SPUCore
- SPEChannel
- LS\_Stats
- SPUMemory

Two of the sub-items, and , represent windows that show state information on the MFC:

- MFC
- MFC\_XLate

The last three sub-items represent actions to perform on the SPE:

- SPUStats
- Model
- Load-Exec

Several interesting SPE data windows are shown in this section's figures. Figure 35 on page 137 shows the MFC window, which provides internal MFC state information. Figure 36 on page 138 shows the MFC\_XLate window, which provides translation structure state information. Figure 37 on page 139 shows the SPEChannel window, which provides information about the SPE's channels.

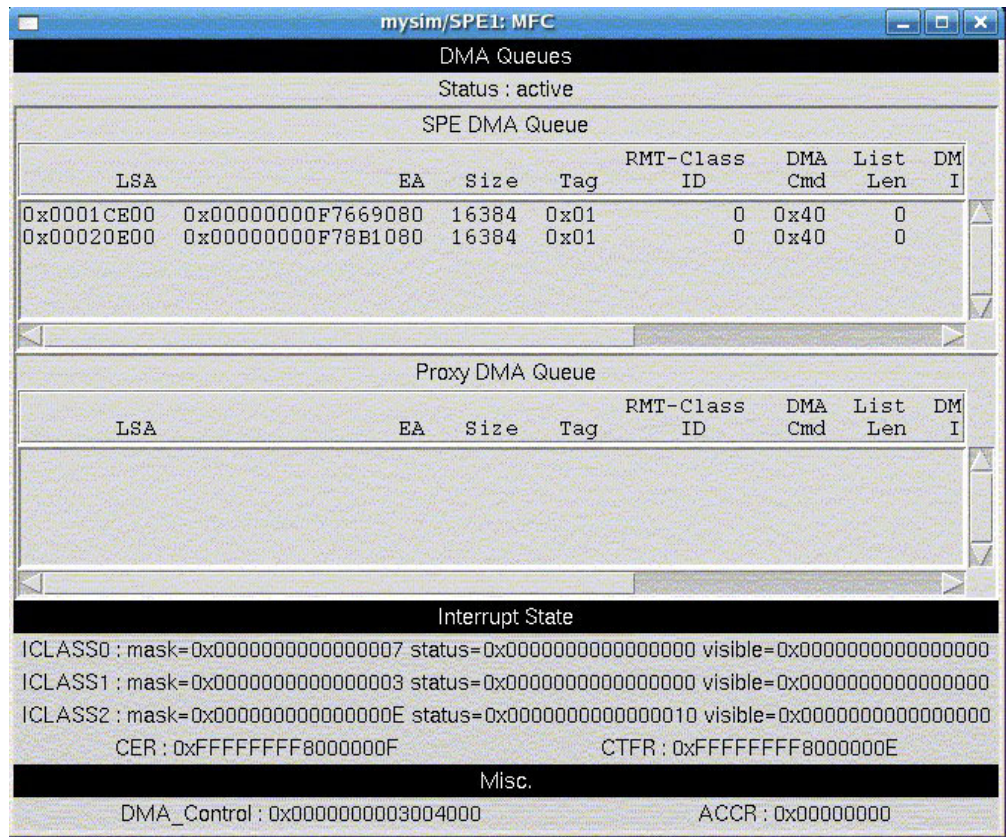


Figure 35. SPE MFC window



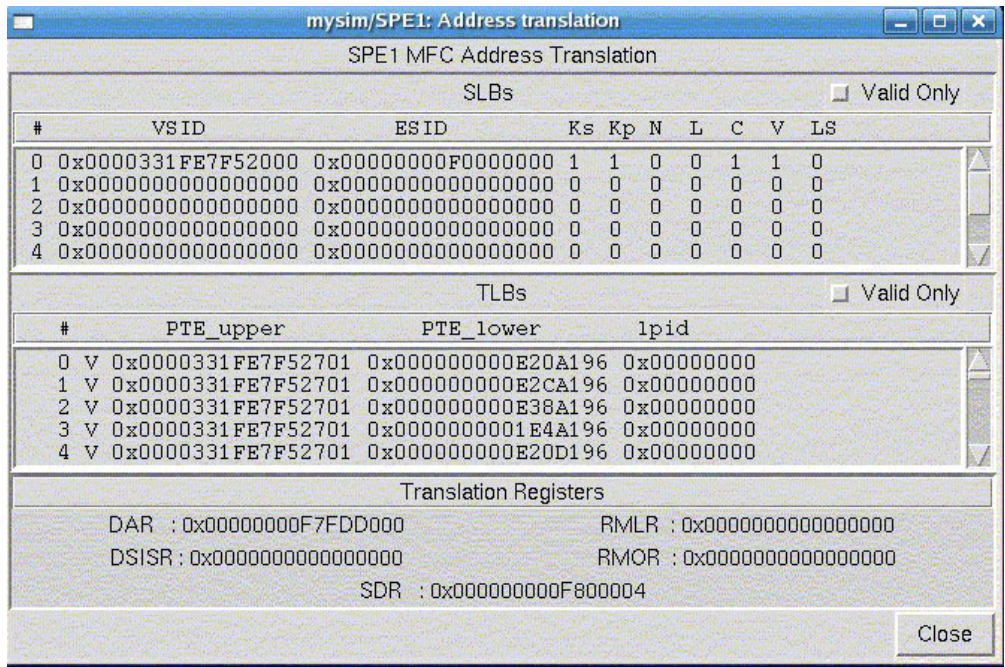


Figure 36. SPE MFC Address Translation window

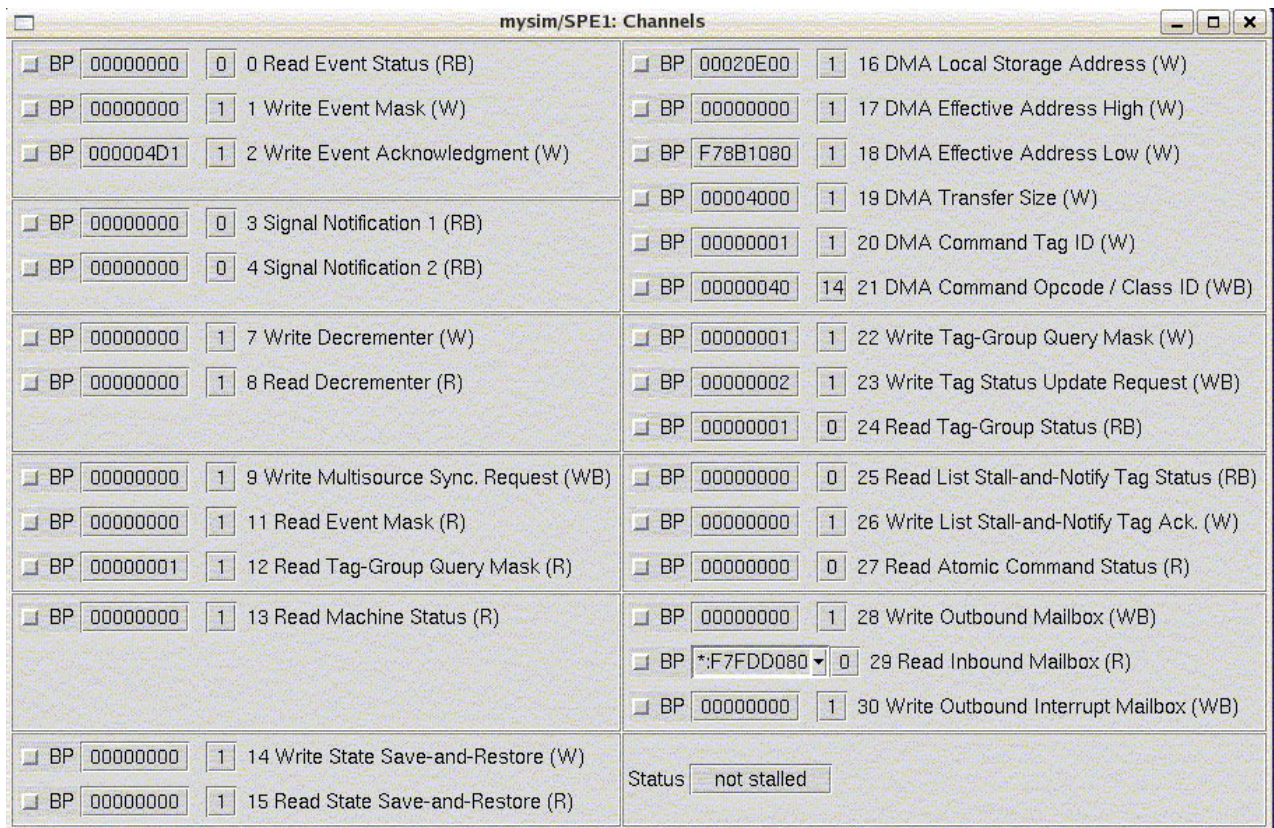


Figure 37. SPE Channels window

The last three items in an SPE folder represent actions to perform, with respect to the associated SPE. The first of these is SPUStats. When the system is stopped and you double-click on this item, the simulator displays program performance statistics in its own pop-up window.

Figure 38 on page 140 shows an example of a statistics dump. These statistics are only collected when the Model is set to *pipeline mode*.





Figure 38. SPE statistics

The next item in the SPE folder is labelled either:

- Model: instruction,
- Model: pipeline, or
- Model: fast.

The label indicates whether the simulation is in:

- *instruction mode* for checking and debugging the functionality of a program,
- *pipeline mode* for collecting performance statistics on the program, or
- *fast mode* for fast functional simulation only.

The model can be toggled by double-clicking the item. The Perf Models button on the GUI can also be used to display a menu for setting the simulator model modes of all of the SPEs simultaneously.



The last item in the SPE folder, Load-Exec, is used for loading an executable onto an SPE. When you double-click the item, a file-browsing window is displayed, allowing you to find and select the executable file to load.

## GUI buttons

On the right side of the GUI screen are five rows of buttons. These are used to manipulate the simulation process.

The five rows of buttons, shown in Figure 30 on page 133, do the following:

- *Advance Cycle* — Advances the simulation by a set number of cycles. The default value is 1 cycle, but it can be changed by entering an integer value in the textbox above the buttons, or by moving the slider next to the textbox. The drop-down menu at the top of the GUI allows the user to select the time domain for cycle stepping. The time units to use for cycles are expressed in terms of various system components. The simulation must be stopped for this button to work; if the simulation is not stopped, the button is inactive.
- *Go* — Starts or continues the simulation. In the SDK's simulator, the first time the Go button is clicked it initiates the Linux boot process. (In general, the action of the Go button is determined by the startup `tc1` file located in the directory from which the simulator is started.)
- *Stop* — Pauses the simulation.
- *Service GDB* — Allows the external gdb debugger to attach to the running program. This button is also inactive while the simulation is running.
- *Triggers/Breakpoints* — Displays a window showing the current triggers and breakpoints.
- *Update GUI* — Refreshes all of the GUI screens. By default, the GUI screens are updated automatically every four seconds. Click this button to force an update.
- *Debug Controls* — Displays a window of the available debug controls and allows you to select which ones should be active. Once enabled, corresponding information messages will be displayed. Figure 39 on page 143 shows the *Debug Controls* window.
- *Options* — Displays a window allowing you to select fonts for the GUI display. On a separate tab, you can enter the gdb debugger port.
- *Emitters* — Displays a window with the defined emitters, with separate tabs for writers and readers. Figure 46 on page 154 shows the *Emitters* window. For more on emitters, see “Emitters” on page 153.
- *Fast Mode* — Toggles fast mode on and off. Fast mode accelerates the execution of the PPE at the expense of disabling certain system-analysis features. It is useful for quickly advancing the simulation to a point of interest. When fast mode is on, the button appears depressed; otherwise it appears normal. Fast mode can also be enabled with the `mysim fast on` command and disabled with the `mysim fast off` command.
- *Perf Models* — Displays a window in which various performance models can be selected for the various system simulator components. Provides a convenient means to set each SPU's simulation mode to either cycle accurate pipeline mode or instruction mode or fast functional-only mode. The same capabilities are available using the `Model:instruction`, `Model:pipeline`, `Model:fast` toggle menu sub-item under each SPE in the tree menu at the left of the main control panel. Figure 42 on page 146 shows the *SPU Performance Model Modes* window.
- *SPE Visualization* — Plots histograms of SPU and DMA event counts. The counts are sampled at user defined intervals, and are continuously displayed. Two modes of display are provided: a “scroll” view, which tracks only the most

recent time segment, and a “compress” view, which accumulates samples to provide an overview of the event counts during the time elapsed. Users can view collected data in either detail or summary panels.

- The detailed, single-SPE panel tracks SPU pipeline phenomena (such as stalls, instructions executed by type, and issue events), and DMA transaction counts by type (gets, puts, atomics, and so forth).
- The summary panel tracks all eight SPEs for the CBE, with each plot showing a subset of the detailed event count data available.

Figure 40 on page 144 shows the *SPE Visualization* window.

- *Process-Tree and Process-Tree-Stats* — This feature requires OS kernel hooks that allow the simulator to display process information. This feature is currently not provided in the SDK kernel.
- *Track All PCs* — Figure 41 on page 145 shows the *Track All PCs* window.
- *SPU Modes* — Provides a convenient means to set each SPU’s simulation mode to either cycle accurate pipeline mode or fast functional-only mode. The same capabilities are available using the `Model:instruction` or `Model:pipeline` toggle menu sub-item under each SPE in the tree menu at the left of the main control panel. Figure 42 on page 146 shows the *SPU Modes* window.
- *Event Log* — Enables a set of predefined triggers to start collecting the log information. The window provides a set of buttons that can be used to set the marker cycle to a point in the process.
- *Exit* — Exits the simulator and closes the GUI window.

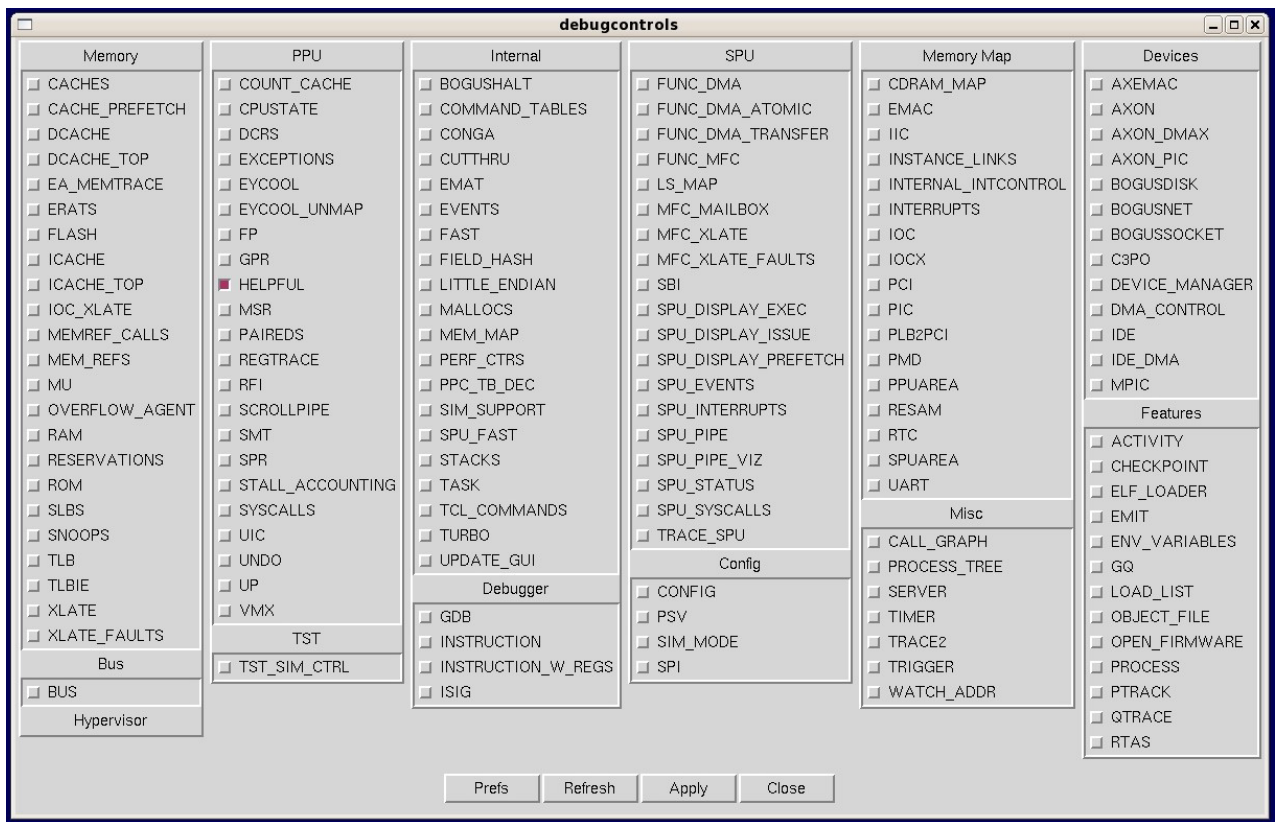


Figure 39. Debug Controls window

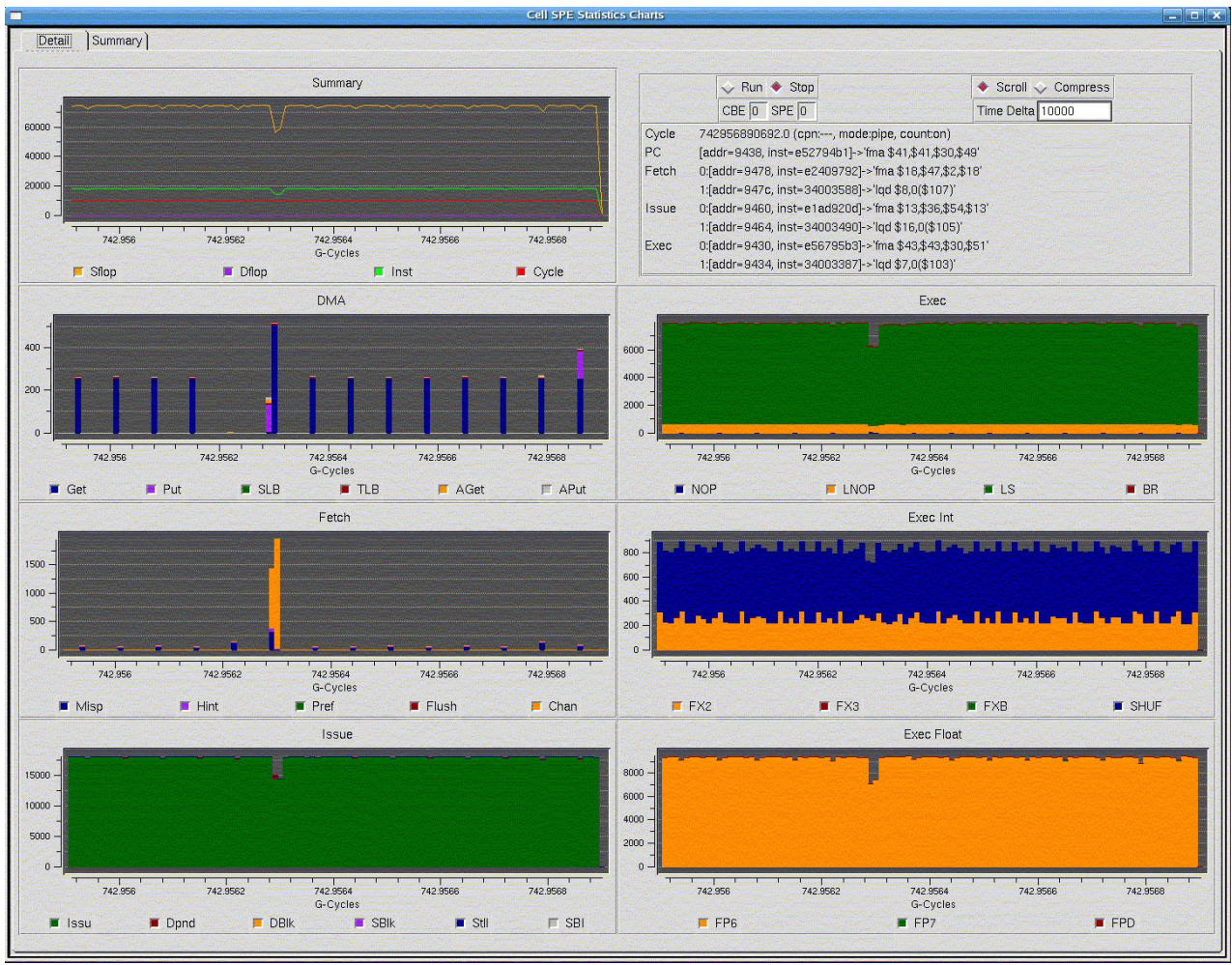


Figure 40. SPE Visualization window



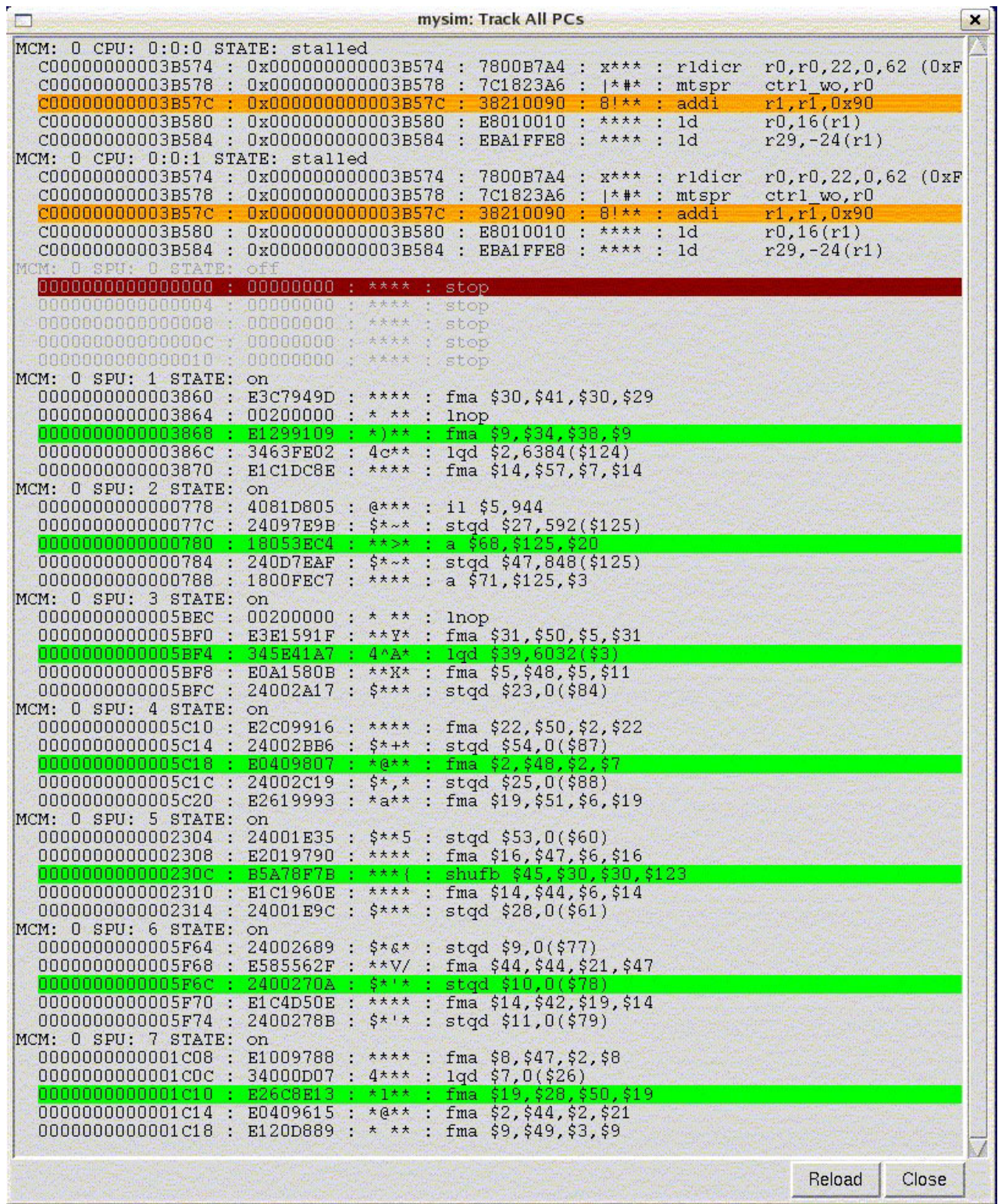


Figure 41. Track All PCs window



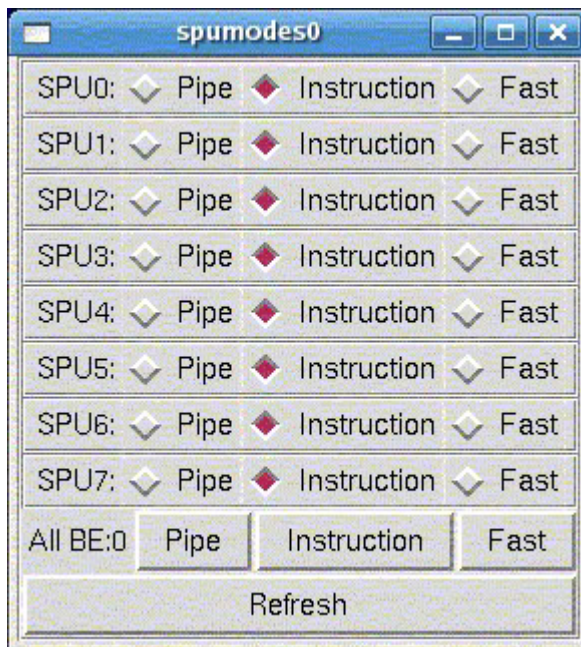


Figure 42. SPU Modes window

## Performance monitoring

The simulator provides both functional-only and cycle-accurate simulation modes.

The *functional-only mode* models the effects of instructions, without accurately modeling the time required to execute the instructions. In functional-only mode, a fixed latency is assigned to each instruction; the latency can be arbitrarily altered by the user. Since latency is fixed, it does not account for processor implementation and resource conflict effects that cause instruction latencies to vary. Functional-only mode assumes that memory accesses are synchronous and instantaneous. This mode is useful for software development and debugging, when a precise measure of execution time is not required.

The *cycle-accurate mode* models not only functional accuracy but also timing. It considers internal execution and timing policies as well as the mechanisms of system components, such as arbiters, queues, and pipelines. Operations may take several cycles to complete, accounting for both processing time and resource constraints.

The cycle-accurate mode allows you to:

- Gather and compare performance statistics on full systems, including the PPE, SPEs, MFCs, PPE caches, bus, and memory controller.
- Determine precise values for system validation and tuning parameters, such as cache latency.
- Characterize the system workload.
- Forecast performance at future loads, and fine-tune performance benchmarks for future validation.

In the cycle-accurate mode, the simulator automatically collects many performance statistics. Some of the more important SPE statistics are:

- Total cycle count
- Count of branch instructions
- Count of branches taken
- Count of branches not taken
- Count of branch-hint instructions
- Count of branch-hints taken
- Contention for an SPE's local store
- Stall cycles due to dependencies on various pipelines

## Displaying performance statistics

You can collect and display simple performance statistics on a program without performing any instrumentation of the program code. Collection of more complex statistics requires program instrumentation.

The following steps demonstrate how to collect and display simple performance statistics. The example PPE program starts (“spawns”) the same thread on three SPEs.

- When an SPE thread is spawned, its SPE number (any number between 0 and 7) is passed in a data structure as a parameter to the `main` function.
- The SPE program contains a for-loop that is executed zero or more times.
- The number of times it is executed is equal to three times the value passed to its `main` function.

The names of the PPE and SPE programs are `tpa1` and `tpa1_spu`, respectively. Part of the most important sections of the programs are shown in “Example program: `tpa1`” on page 152.

The following steps are marked as to whether they are performed in the simulator's command window or its console window. To collect and display simple performance statistics, do the following:

1. *Start the simulator.* Start the simulator by entering the following command:

```
PATH=/opt/ibm/systemsin-cell/bin:$PATH; systemsim
```

This command starts the simulator in command-line mode, and displays the *simulator prompt*.

```
systemsim %
```

2. *In the command window, set the SPUs to pipeline mode.* An SPU must be in pipeline mode to collect performance statistics from that SPU. If, instead, the SPU is in instruction mode, it will only report the total instruction count. Use the `mysim spu` command to set those processors to pipeline mode, as follows:

```
mysim spu 0 set model pipeline
mysim spu 1 set model pipeline
mysim spu 2 set model pipeline
```

**Note:** The specific SPU numbers are *examples only*. The operating system may assign the SPU programs to execute on a different set of SPUs. You can also use the SPU Modes button or the folder under each SPE labeled Model to set the model to pipeline mode.

3. *In the command window, boot Linux.* Boot the Linux operating system on the simulated PPE by entering:

```
mysim go
```

4. *In the console window, load the executables.* Load the PPE and SPE executables from the base environment into the simulated environment, and set their file permissions to executable, as follows:

```
callthru source tpa1 > tpa1  
callthru source tpa1_spu > tpa1_spu  
chmod +x tpa1  
chmod +x tpa1_spu
```

5. *In the console window, run the PPE program.* Run the PPE program in the simulation by entering the name of the executable file, as follows:

```
tpa1
```

6. *In the command window, pause the simulation and display statistics.* When the program finishes execution, select the simulator control window. Pause the simulator by entering the Ctrl-c key sequence. To display the performance statistics for the three SPEs, enter the following commands:

```
mysim spu 0 display statistics  
mysim spu 1 display statistics  
mysim spu 2 display statistics
```

As each command is entered, the simulator displays the performance statistics in the simulator command window. Figure 43 on page 149 shows a screen image of the SPE 0 performance statistics.



---

```

SPU DD3.0
***
Total Cycle count          35185
Total Instruction count    643
Total CPI                  54.72
***
Performance Cycle count   35185
Performance Instruction count 1701 (1502)
Performance CPI           20.68 (23.43)

Branch instructions       135
Branch taken              120
Branch not taken          15

Hint instructions         9
Hint hit                  31

Contention at LS between Load/Store and Prefetch 49

Single cycle              1108 ( 3.1%)
Dual cycle                197 ( 0.6%)
Nop cycle                 137 ( 0.4%)
Stall due to branch miss 1655 ( 4.7%)
Stall due to prefetch miss 0 ( 0.0%)
Stall due to dependency   826 ( 2.3%)
Stall due to fp resource conflict 0 ( 0.0%)
Stall due to waiting for hint target 11 ( 0.0%)
Issue stalls due to pipe hazards 6 ( 0.0%)
Channel stall cycle      31236 ( 88.8%)
SPU Initialization cycle 9 ( 0.0%)
-----
Total cycle                35185 (100.0%)

Stall cycles due to dependency on each pipelines
FX2      62 ( 7.5% of all dependency stalls)
SHUF     322 ( 39.0% of all dependency stalls)
FX3       2 ( 0.2% of all dependency stalls)
LS       413 ( 50.0% of all dependency stalls)
BR        0 ( 0.0% of all dependency stalls)
SPR      21 ( 2.5% of all dependency stalls)
LNOP     0 ( 0.0% of all dependency stalls)
NOP       0 ( 0.0% of all dependency stalls)
FXB       0 ( 0.0% of all dependency stalls)
FP6       0 ( 0.0% of all dependency stalls)
FP7       0 ( 0.0% of all dependency stalls)
FPD       6 ( 0.7% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00
dumped pipeline stats

```

---

Figure 43. *tpa1* statistics for SPE 0

Although the programs on SPE 0 and SPE 2 are the same, the program on SPE 0 executed the loop zero times, but the program on SPE 2 executed the loop six times.

You can compare the performance statistics of SPE 0 (Figure 43) with those of SPE 2, which are shown in Figure 44 on page 150.

**Note:** The statistics collected in this manner include the SPU cycles required to load the SPE thread, start the SPE thread, and cleanup the SPE thread upon

completion.

---

```
SPU DD3.0
***
Total Cycle count          35537
Total Instruction count    643
Total CPI                  55.27
***
Performance Cycle count   35537
Performance Instruction count 1802 (1590)
Performance CPI           19.72 (22.35)

Branch instructions       153
Branch taken              136
Branch not taken          17

Hint instructions         15
Hint hit                  37

Contention at LS between Load/Store and Prefetch 49

Single cycle              1170 ( 3.3%)
Dual cycle                210 ( 0.6%)
Nop cycle                 150 ( 0.4%)
Stall due to branch miss 1854 ( 5.2%)
Stall due to prefetch miss 0 ( 0.0%)
Stall due to dependency   879 ( 2.5%)
Stall due to fp resource conflict 0 ( 0.0%)
Stall due to waiting for hint target 23 ( 0.1%)
Issue stalls due to pipe hazards 6 ( 0.0%)
Channel stall cycle       31236 ( 87.9%)
SPU Initialization cycle  9 ( 0.0%)
-----
Total cycle                35537 (100.0%)

Stall cycles due to dependency on each pipelines
FX2      86 ( 9.8% of all dependency stalls)
SHUF     348 ( 39.6% of all dependency stalls)
FX3       2 ( 0.2% of all dependency stalls)
LS       413 ( 47.0% of all dependency stalls)
BR        3 ( 0.3% of all dependency stalls)
SPR      21 ( 2.4% of all dependency stalls)
LNOP     0 ( 0.0% of all dependency stalls)
NOP       0 ( 0.0% of all dependency stalls)
FXB       0 ( 0.0% of all dependency stalls)
FP6       0 ( 0.0% of all dependency stalls)
FP7       0 ( 0.0% of all dependency stalls)
FPD       6 ( 0.7% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00
dumped pipeline stats
```

---

Figure 44. *tpa1* statistics for SPE 2

## SPE performance profile checkpoints

The simulator can automatically capture system-wide performance statistics that are useful in determining the sources of performance degradation, such as channel stalls and instruction-scheduling problems.

You can also use SPE performance *profile checkpoints* to delimit a specific region of code over which performance statistics are to be gathered.

Performance profile checkpoints (such as `prof_clear` , `prof_start` and `prof_stop` in the code samples below) can be used to capture higher-level statistics such as the total number of instructions, the number of instructions other than no-op instructions, and the total number of cycles executed by the profiled code segment.

The checkpoints are special no-op instructions that indicate to the simulator that some special action should be performed. No-op instructions are used because they allow the same program to be executed on real hardware. A SPE header file, `profile.h` , provides a convenient function-call-like interface to invoke these instructions.

In addition to displaying performance information, certain performance profile checkpoints can control the statistics-gathering functions of the SPU.

For example, profile checkpoints can be used to capture the total cycle count on a specific SPE. The resulting statistic can then be used to further guide the tuning of an algorithm or structure of the SPE. The following example illustrates the profile-checkpoint code that can be added to an SPE program in order to clear, start, and stop a performance counter:

```
#include <profile.h>
...
prof_clear();    // clear performance counter
prof_start();    // start recording performance statistics
...
<code_to_be_profiled>
...
prof_stop();     // stop recording performance statistics
```

When a profile checkpoint is encountered in the code, an instruction is issued to the simulator, causing the simulator to print data identifying the calling SPE and the associated timing event. The data is displayed on the simulator control window in the following format:

```
SPUn: Cm, xxxxx(yyyyy), zzzzzz
```

where:

- *n* is the number of the SPE on which the profile checkpoint has been issued,
- *m* is the checkpoint number,
- *xxxxx* is the instruction counter,
- *yyyyy* is the instruction count excluding no-ops,
- *zzzzzz* is the cycle counter.

The following example uses the `tpa1_spu` program and instruments the loop with the `prof_clear` , `prof_start` and `prof_stop` profile checkpoints. The relevant code is shown here.

```
// file tpa2_spu.c

#include <sim_printf.h>
#include <profile.h>

...

prof_clear();
prof_start();
for( i=0; i<spe_num*3; i++ )
    sim_printf("SPE#: %lld, Count: %d\n", spe_num, i);
prof_stop();
```

Figure 45 shows the output produced by the program.

---

```
SPU2: CP0, 863(740), 17800
clear performance info.
SPU2: CP30, 0(0), 1
start recording performance info.
SPE#: 25296904, Count: 0
SPE#: 25296904, Count: 1
SPE#: 25296904, Count: 2
SPE#: 25296904, Count: 3
SPE#: 25296904, Count: 4
SPE#: 25296904, Count: 5
SPU2: CP31, 118(103), 400
stop recording performance info.
```

---

Figure 45. Profile checkpoint output for SPE 2

## Example program: tpa1

The following example program `tpa1` is used in the sections above to show the basic performance statistics that can be collected and displayed without instrumentation of the code.

`tpa1.c` is the source code for the PPE, which spawns three copies of program `tpa1_spu` on SPEs 0, 1 and 2. The code in `tpa1_spu` executes the for-loop a different number of times in each of the SPEs. For each SPE, the loop is executed three times the number passed in as the parameter.

```
// file tpa1.c - error checking removed to improve readability

...

// the value of nr_spus is 3
for (i = 0; i < nr_spus; i++) {
    spe_context_ptr_t spe;
    spe_program_handle_t *tpa1_spu;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    union {
        void *ptr;
        unsigned int spe_num;
    } t_info;

    spe = spe_context_create(0, NULL);
    tpa1_spu = spe_image_open("tpa1_spu")
    (void)spe_program_load(spe, &tpa1_spu);
    printf("Spawning thread: %d\n", i);
    t_info.spe_num = i;
    (void)spe_context_run(spe, &entry, 0, t_info.ptr, NULL, NULL);
    (void)spe_context_destroy(spe);
}

// file tpa1_spu.c

main(unsigned long long id, unsigned long long spe_num)
{
    int i;

    for( i=0; i<spe_num*3; i++ )
        sim_printf("SPE#: %lld, Count: %d\n", spe_num, i);
}
```

## Emitters

In addition to the basic cycle-count and summary statistics provided by its profile checkpoints and triggers, the simulator also supports a user-extensible event-analysis system, called *emitters*.

The emitters, selected on the GUI screen (Figure 30 on page 133), de-couple performance event-collection from performance analysis tools. The emitter event-analysis system has two primary functions:

- *Event Data Production* — During simulation, the simulator can identify a wide variety of architectural and programmatic events that influence system and software performance. Using configuration commands, you can request the simulator to emit records for a specific set of events into a circular, shared memory buffer. Reader programs attach to the shared memory buffer to consume these event records. Examples of emitter events include instruction execution, memory-reference addresses, and cache hits and misses.
- *Event Processing* — There are one or more readers that analyze event records from this buffer. The readers typically compute performance measurements and statistics, visualize system and application behavior, and capture traces for post-processing. The simulator is prepackaged with a set of prebuilt sample emitter readers, and users can develop and customize their own emitter readers.

Figure 46 on page 154 shows the emitter selections available by clicking the Emitters button on the GUI screen.

Figure 47 on page 154 shows the emitter architecture. Emitters can be used in any simulator mode. The writer toggle buttons in the GUI are used to enable or disable production of the associated event to the circular buffer. An emitter reader program is needed to receive the events from the circular buffer using the emitter reader API.

The emitter framework is meant for programmers who wish to conduct performance analyses or capture traces by developing custom reader programs.

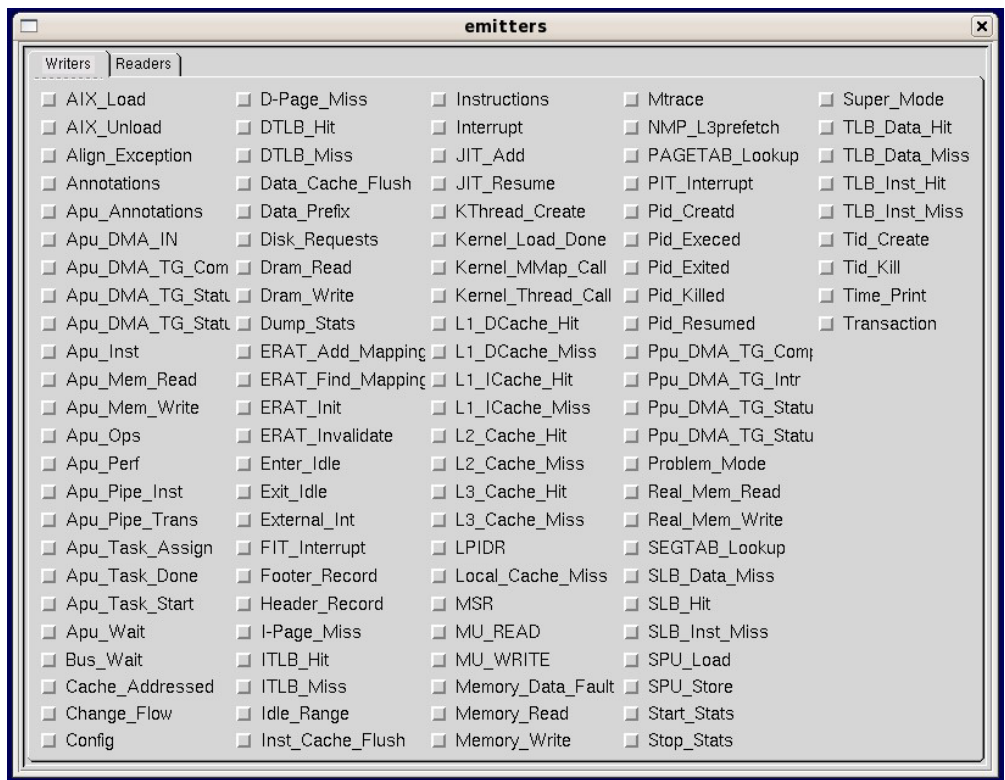


Figure 46. Emitters

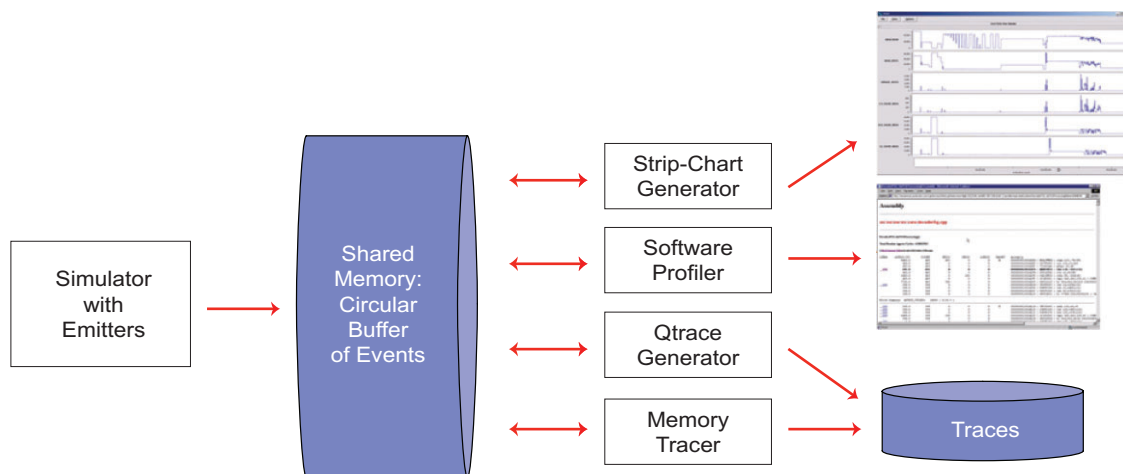


Figure 47. Emitter architecture

The types of events that can be tracked are described in:

`/opt/ibm/systemsim-cell/include/emitter/emitter_data_t.h`

The categories of events are:

- Begin/end markers (Header, Footer)
- PPU and SPU instructions
- Cache hits or misses
- Process/thread state (create, resume, kill, and so forth)
- Translation Lookaside Buffer (TLB), Segment Lookaside Buffer (SLB), Effective-to-Real Address Translation (ERAT) operations
- Device operations (disk)
- Annotations
- Transactions

---

## SPU performance and semantics

The simulator collects several statistics related to SPU performance.

Table 40 lists the performance statistics that are available in the *public SDK*.

*Table 40. Simulator Performance Statistics for the SPU*

Statistic Name	Meaning
performance_inst_count	Instruction count (profile checkpoint sensitive), including and not including no-ops.
performance_cycle_count	Cycle count (profile checkpoint sensitive).
branch_taken	Count of branch instructions taken.
branch_not_taken	Count of branch instructions not taken.
hint_instructions	Count of branch hint instructions.
hint_instruction_hits	Number of times a hint instruction predicted correctly.
ls_contention	Number of cycles in which local store load/store instructions prevented prefetch.
sbi_contention	Number of cycles in which the Synergistic Bus Interface (SBI) DMA operations prevented SPU local store access.
single_cycle	Number of cycles in which only one pipeline executed an instruction.
dual_cycle	Number of cycles in which both pipelines executed an instruction.
sp_issue_block	Number of cycles in which dual-issue was prevented, due to an SP-class instruction not being available to issue.
dp_issue_block	Number of cycles in which dual-issue was prevented, due to a DP-class instruction not being available to issue.
cross_issue_cycle	Number of cycles in which issue pipe{0,1} sent an instruction to the opposite issue pipe{1, 0}.
nop_inst_count	Number of NOP instructions executed (NOP, LNOP, HBR, and HBC).
src0_dep_cycle	Number of cycles in which dual-issue was prevented, due to operand dependencies between the two instructions that were ready to issue simultaneously.



Table 40. Simulator Performance Statistics for the SPU (continued)

Statistic Name	Meaning
nop_cycle	Number of cycles in which a NOP was executed in either pipeline.
branch_stall_cycles	Number of cycles stalled due to branch miss.
prefetch_miss_stall_cycles	Number of cycles instruction issue stalled due to prefetch miss.
pipe_dep_stall_cycles	Number of cycles instruction issue stalled, due to source operand dependencies on target operands in any execution pipeline.
pipe_busy_cycles	Number of cycles all execution pipelines were expected to be busy processing in-flight instructions (unaffected by flush).
fp_resource_conflict_stall_cycles	Number of cycles stalled due to floating-point unit resource conflict.
hint_stall_cycles	Number of cycles stalled due to waiting for hint target.
siss_stall_cycles	Number of cycles stalled due to structural execution pipe dependencies.
channel_stall_cycles	Number of cycles stalled waiting for a channel operation to complete.
XXX_inst_count (see below)	Number of XXX instructions executed.
XXX_dep_stall_cycles (see below)	Number of cycles stalled due to a source operand dependency on a target operand of an in-flight instruction in the XXX execution pipeline.
XXX_iss_stall_cycles (see below)	Number of cycles stalled due to a structural dependency on an XXX class instruction.
XXX_busy_cycle (see below)	Total cycles the XXX execution pipeline was expected to be busy processing in-flight instructions (unaffected by flush).
Where XXX (above) is one of:	
FX2	SPX fixed-point unit (fixed [FX] class) instructions.
SHUF	SFS shuffle and quad-rotate fixed-point unit (shuffle [SH] class) instructions.
FX3	SFX 4-cycle fixed-point unit (word rotate and shift [WS] class) instructions.
LS	SLS load and store unit (load and store [LS] class) instructions.
BR	SCN branch and control unit and sequencer (branch resolution [BR] class) instructions.
SPR	SSC Channel and DMA unit (channel interface [CH] class) instructions.
LNOP	Odd pipeline (load no operation [LNOP] class) no-ops.
NOP	Even pipeline (NOP class) no-ops.
FXB	SFP byte operations (byte operations [BO] class) instructions.
FP6	SFP FPU single-precision (single-precision floating-point [SP] class) instructions.

Table 40. Simulator Performance Statistics for the SPU (continued)

Statistic Name	Meaning
FP7	SFP integer (floating-point integer [FI] class) instructions.
FPD	SFP FPU double-precision (double-precision floating-point [DP] class) instructions.



---

## Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing 2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licenses of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Software Interoperability Coordinator, Department 49XA  
3605 Highway 52 N  
Rochester, MN 55901  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## **Edition notices**

**© Copyright International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation 2006, 2007. All rights reserved.**

U.S. Government Users Restricted Rights — Use, duplication, or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.





---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

BladeCenter  
IBM  
The IBM logo  
ibm.com  
POWER  
Power PC  
PowerPC  
PowerPC Architecture

Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom

Intel, Intel Inside<sup>®</sup> (logos), MMX, and Pentium<sup>®</sup> are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft<sup>®</sup>, Windows<sup>®</sup>, Windows NT<sup>®</sup>, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java<sup>™</sup> and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Red Hat, the Red Hat “Shadow Man” logo, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc., in the United States and other countries.

UNIX<sup>®</sup> is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.



---

## Glossary

**ABI.** Application Binary Interface. This is the standard that a program follows to ensure that code generated by different compilers (and perhaps linking with various, third-party libraries) will run correctly on the Cell Broadband Engine. The ABI defines data types, register use, calling conventions, object formats.

**AOS.** Array of structures. A method of organizing related data values. Also called vector-across form. See also *SOA*.

**API.** Application Program Interface.

**ATO.** Atomic Unit. Part of an SPE's MFC. It is used to synchronize with other processor units.

**atomic access.** A bus access that attempts to be part of an atomic operation.

**atomic operation.** A set of operations, such as read-write, that are performed as an uninterrupted unit.

**BIC.** Bus Interface Controller. Part of the Cell Broadband Engine Interface (BEI) to I/O.

**BIF.** Cell Broadband Engine Interface. The EIB's internal communication protocol. It supports coherent interconnection to other Cell Broadband Engines and BIF-compliant I/O devices, such as memory subsystems, switches, and bridge chips. See also *IOIF*.

**BIU.** Bus Interface Unit. Part of the PPE's interface to the EIB.

**branch hint.** A type of branch instruction that provides a hint of the address of the branch instruction and the address of the target instruction. Hints are coded by the programmer or inserted by the compiler. The branch is assumed taken to the target. Hints are used in place of branch prediction in the SPU.

**built-ins.** A type of C and C++ programming language intrinsic that "built in" to the compiler.

**B.** Byte.

**cache.** High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.

**caching-inhibited.** A memory update policy in which the cache is bypassed, and the load or store is performed to or from main memory.

**CBEA.** Cell Broadband Engine Architecture. The Cell Broadband Engine is one implementation of the Cell Broadband Engine Architecture.

**Cell Broadband Engine Linux task.** A task running on the PPE and SPE. Each such task has one or more Linux threads and some number of SPE threads. All the Linux threads within the task share the task's resources, including access to the SPE threads.

**Cell Broadband Engine program.** A PPE program with one or more embedded SPE programs.

**channel.** Channels are unidirectional, function-specific registers or queues. They are the primary means of communication between an SPE's SPU and its MFC, which in turn mediates communication with the PPE, other SPEs, and other devices. These other devices use MMIO registers in the destination SPE to transfer information on the channel interface of that destination SPE.

Specific channels have read or write properties, and blocking or nonblocking properties. Software on the SPU uses channel commands to enqueue DMA commands, query DMA and processor status, perform MFC synchronization, access auxiliary resources such as the decremter (timer), and perform interprocessor-communication via mailboxes and signal-notification.

**CL.** The class-ID parameter in an MFC command.

**coherence.** Memory and cache coherence. The correct ordering of stores to a memory address, and the enforcement of any required cache write-backs during accesses to that memory address. Cache coherence is implemented by a hardware snoop (or inquire) method, which compares the memory addresses of a load request with all cached copies of the data at that address. If a cache contains a modified copy of the requested data, the modified data is written back to memory before the pending load request is serviced.

**control plane.** Software or hardware that manages the operation of data-plane software or hardware, by allocating resources, updating tables, handling errors, and so forth. See also *data-plane*.

**cycle.** Unless otherwise specified, one tick of the PPE clock.

**data plane.** Software or hardware that operates on a stream or other large body of data and is managed by control-plane software or hardware. See also *control-plane*.

**decrementer.** A register that counts down each time an event occurs. Each SPU contains dedicated 32-bit decrementers for scheduling or performance monitoring, by the program or by the SPU itself.

**D-ERAT.** Data ERAT.

**DMA.** Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

**DMAC.** Direct Memory Access Controller. A controller that performs DMA transfers.

**DMA command.** A type of MFC command that transfers or controls the transfer of a memory location containing data or instructions. See also *MFC command*.

**DMA list.** A sequence of *transfer elements* (or list entries) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers between a single area of LS and discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA-list command such as *getl* or *putl*. DMA-list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

**DMA-list command.** A type of MFC command that initiates a sequence of DMA transfers specified by a DMA list stored in an SPE's LS. See also *DMA list*.

**DMA queue.** A set of two queues for holding DMA-transfer commands. The SPE's queue has 16 entries. The PPE's queue has four entries (two plus an additional two for the L2 cache) for SPE-requested DMA commands, and eight entries for PPE-requested DMA commands.

**dual-issue.** Issuing two instructions at once, under certain conditions. See also *fetch group*.

**EA.** Effective address.

**ECC.** Error-Correcting Code.

**effective address.** An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective-address space is  $2^{64}$  bytes.

**EIB.** Element Interconnect Bus. The on-chip coherent bus that handles communication between the PPE, SPEs, memory, and I/O devices (or a second Cell Broadband Engine). The EIB is organized as four unidirectional data rings (two clockwise and two counterclockwise).

**ELF.** Executable and Linking Format. The standard object format for many UNIX operating systems, including Linux. Originally defined by AT&T and placed in public domain. Compilers generate ELF files. Linkers link to files with ELF files in libraries. Systems run ELF files.

**ERAT.** Effective-to-Real Address Translation, or a buffer or table that contains such translations, or a table entry that contains such a translation.

**even pipeline.** Part of an SPE's dual-issue execution pipeline. Also referred to as pipeline 0.

**exception.** An error, unusual condition, or external signal that may alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled. See also *interrupt*.

**fence.** An option for a barrier ordering command that causes the processor to wait for completion of all MFC commands before starting any commands queued after the fence command. It does not apply to these immediate commands: *getllar*, *putllc*, and *putlluc*.

**fetch group.** A doubleword-aligned instruction pair. Dual-issue occurs when a fetch group has two instructions that are ready to issue, and when the first instruction can be issued on the even pipeline and the second instruction can be issued on the odd pipeline.

**FIFO.** First In First Out. Refers to one way elements in a queue are processed. It is analogous to "people standing in line."

**flat register .** An architecture with only one register file, in which all types of operands are stored. Also called a unified register file. By contrast, conventional register architectures have separate sets of special-purpose registers for such things as scalar operands, floating-point operands, vectors, branch-and-link values, conditions, and so forth. The SPEs have a flat register architecture. The PPE has a conventional register architecture.

**FlexIO.** Rambus FlexIO bus, a high performance I/O bus.

**FPU.** Floating-point unit.

**FXU.** In the PPE, the fixed-point integer unit. In the SPU, the fixed-point exception unit.

**gdb.** GNU debugger. A modified version of *gdb*, *ppu-gdb*, starts a Cell Broadband Engine program. The PPE component runs first and uses system calls, hidden by the SPU programming library, to move the SPU component of the Cell Broadband Engine program into the local store of the SPU and start it running.

**generic intrinsics.** C and C++ language extensions that map to one or more specific intrinsics. (See also *intrinsic*.) All generic SPU intrinsics are prefaced by the

string `spu_`. For example, the generic intrinsic that implements the stop assembly instruction is named `spu_stop`.

**guarded.** Prevented from responding to speculative loads and instruction fetches. The operating system typically implements guarding, for example, on all I/O devices.

**hypervisor.** A control (or virtualization) layer between hardware and the operating system. It allocates resources, reserves resources, and protects resources among (for example) sets of SPEs that may be running under different operating systems.

The Cell Broadband Engine has three operating modes: user, supervisor and hypervisor. The hypervisor performs a meta-supervisor role that allows multiple independent supervisors' software to run on the same hardware platform.

For example, the hypervisor allows both a real-time operating system and a traditional operating system to run on a single PPE. The PPE can then operate a subset of the SPEs in the Cell Broadband Engine with the real-time operating system, while the other SPEs run under the traditional operating system.

**IEEE 754.** The IEEE 754 floating-point standard. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

**I-ERAT.** Instruction ERAT.

**imprecise exception.** A synchronous exception that does not adhere to the precise exception model. In the Cell Broadband Engine, single-precision floating-point operations generate imprecise exceptions. See also *precise exception*.

**instruction latency.** The total number of clock cycles necessary to execute an instruction and produce the results of that instruction.

**in-order.** In program order. The PPE and SPEs execute instructions in-order; that is, they do not rearrange them (out-of-order).

**interrupt.** A change in machine state in response to an exception. See also *exception*.

**intrinsic.** A C-language command, in the form of a function call, that is a convenient substitute for one or more inline assembly-language instructions. Intrinsic make the underlying ISA accessible from the C and C++ programming languages.

**IOC.** I/O Interface Controller.

**I/O device.** Input/output device. From software's viewpoint, I/O devices exist as memory-mapped registers that are accessed in main-storage space by

load/store instructions. The operating system typically configures access to I/O devices as caching-inhibited and guarded.

**IOIF.** Cell Broadband Engine I/O Interface. The EIB's noncoherent protocol for interconnection to I/O devices. See also *BIF*.

**JSRE.** Joint Software Reference Environment. An organization of the Cell Broadband Engine developers pursuing the development of reference software and standards for the Cell Broadband Engine.

**JTAG.** Joint Test Action Group. A test-access port defined by the IEEE 1149 standard.

**KB.** Kilobyte.

**L1.** Level-1 cache memory. The closest cache to a processor, measured in access time.

**L2.** Level-2 cache memory. The second-closest cache to a processor, measured in access time. An L2 cache is typically larger than an L1 cache.

**LA.** An LS address of a DMA list. It is used as a parameter in an MFC command.

**latency.** The time between when a function (or instruction) is called and when it returns. Programmers often optimize code so that functions return as quickly as possible; this is referred to as the low-latency approach to optimization. Low-latency designs often leave the processor data-starved, and performance can suffer.

**libspe.a.** An SPU-thread runtime management library.

**Inop.** A NOP in an SPU's odd pipeline. It can be inserted in code to align for dual issue of subsequent instructions.

**local store.** The 256-KB local store (LS) associated with each SPE. It holds both instructions and data.

**loop unrolling.** A programming optimization that increases the step of a loop, and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and memory access time.

**LS.** See *local store*.

**LSA.** Local Store Address. An address in the LS of an SPU, by which programs running in the SPU and DMA transfers managed by the MFC access the LS.

**list element.** See *transfer element* and *DMA list*.

**Linux thread.** A thread running on the PPE in the Linux operating-system environment.

**mailbox.** A queue in an SPE's MFC for exchanging 32-bit messages between the SPE and the PPE or other

devices. Two mailboxes (the SPU Write Outbound Mailbox and SPU Write Outbound Interrupt Mailbox) are provided for sending messages from the SPE. One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to the SPE.

**main memory.** See *main storage*.

**main storage.** The effective-address (EA) space. It consists physically of real memory (whatever is external to the memory-interface controller, including both volatile and nonvolatile memory), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices (all I/O is memory-mapped), and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. See also *local store*.

**makefile.** A descriptive file used by the make command in which the user specifies: (a) target program or library, (b) rules about how the target is to be built, (c) dependencies which, if updated, require that the target be rebuilt.

**MB.** Megabyte.

**memory channel.** An interface to external memory chips. The Cell Broadband Engine supports two Rambus Extreme Data Rate (XDR) memory channels.

**memory-mapped.** Mapped into the Cell Broadband Engine's addressable-memory space. Registers, SPE local stores (LSs), I/O devices, and other readable or writable storage can be memory-mapped. Privileged software does the mapping.

**method stub.** A small piece of code used to stand in for some other code.

**MFC.** Memory Flow Controller. It is part of an SPE and provides two main functions: moves data via DMA between the SPE's local store (LS) and main storage, and synchronizes the SPU with the rest of the processing units in the system.

**MFC proxy commands.** MFC commands issued using the MMIO interface.

**MIC.** Memory Interface Controller. The Cell Broadband Engine's MIC supports two memory channels.

**MMIO.** Memory-Mapped Input/Output. See also *memory-mapped*.

**MMU.** Memory Management Unit. A functional unit that translates between effective addresses (EAs) used by programs and real addresses (RAs) used by physical memory. The MMU also provides protection mechanisms and other functions.

**M:N thread model.** A programming model in which M threads are distributed over N processor elements.

**MPI.** Message Passing Interface.

**MSR.** Machine State Register.

**MT.** Multithreading. See *multithreading*.

**multithreading.** Simultaneous execution of more than one program thread. It is implemented by sharing one software process and set of execution resources but duplicating the architectural state (registers, program counter, flags, and so forth) of each thread.

**NaN.** Not-a-Number. A special string of bits encoded according to the IEEE 754 Floating-Point Standard. A NaN is the proper result for certain arithmetic operations; for example,  $0/0 = \text{NaN}$ . There are two types of NaNs, quiet NaNs and signaling NaNs. Only the signaling NaN raises a floating-point exception when it is generated.

**NCU.** Non-Cacheable Unit.

**odd pipeline.** Part of an SPE's dual-issue execution pipeline. Also referred to as pipeline 1.

**OpenMP.** An API that supports multiplatform, shared-memory parallel programming.

**overlay.** SPU code that is dynamically loaded and executed by a running SPU program.

**page table.** A table that maps virtual addresses (VAs) to real addresses (RA) and contains related protection parameters and other information about memory locations.

**PC.** Personal Computer.

**performance simulation.** Simulation by the IBM Full System Simulator for the Cell Broadband Engine in which both the functional behavior of operations and the time required to perform the operations is simulated. Also called cycle-accurate simulation.

**pervasive logic.** Logic that provides power management, thermal management, clock control, software-performance monitoring, trace analysis, and so forth.

**pipelining.** A technique that breaks operations, such as instruction processing or bus transactions, into smaller stages so that a subsequent stage in the pipeline can begin before the previous stage has completed.

**PMD.** Power Management and Debug.

**POSIX.** Portable Operating System Interface.

**PowerPC 970.** A 64-bit microprocessor from IBM in the PowerPC family. It supports both the PowerPC and Vector/SIMD Multimedia Extension instruction sets.



**PowerPC Architecture.** A computer architecture that is based on the third generation of RISC processors. The PowerPC architecture was developed jointly by Apple, Motorola, and IBM.

**PPE.** PowerPC Processor Element. The general-purpose processor in the Cell Broadband Engine.

**PPSS.** PowerPC Processor Storage Subsystem. Part of the PPE. It operates at half the frequency of the PPU and includes an L2 cache and Bus Interface Unit (BIU).

**PPU.** PowerPC Processor Unit. The part of the PPE that includes the execution units, memory-management unit, and L1 cache.

**precise exception.** An exception for which the pipeline can be stopped, so instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispached after exception handling has completed.

**preferred slot.** The left-most word (bytes 0, 1, 2, and 3) of a 128-bit register in an SPE. The SIMD element in which scalar values are naturally maintained.

**privileged mode.** Also known as supervisor mode. The permission level of operating system instructions. The instructions are described in *PowerPC Architecture, Book III* and are required of software that accesses system-critical resources.

**problem state.** The permission level of user instructions. The instructions are described in *PowerPC Architecture, Books I and II* and are required of software that implements application programs.

**PTE.** Page Table Entry. See *page table*.

**QoS.** Quality of Service. It usually relates to a guarantee of minimum bandwidth for streaming applications.

**RA.** Real Address. See *real address*.

**real address.** An address for physical storage, which includes physical memory, the PPE's L1 and L2 caches, and the SPE's local stores (LSs) if the operating system has mapped the LSs to the real-address space. The maximum size of the real-address space is  $2^{42}$  bytes.

**scalar.** An instruction operand characterized by a single value.

**scarf hint.** A performance hint for DMA put operations. The hint is intended to allow another processor or device, such as the PPE, to capture the data into its cache as the data is transferred to storage.

**SCN.** SPU Control Unit. A unit in the SPU that handles branches and program control.

**SDK.** Software Development Kit. Sample software for the Cell Broadband Engine that includes the Linux operating system.

**semi-pipelined.** A processor is semi-pipelined if it fetches the next instruction while decoding and executing the current instruction.

**SFP.** SPU Floating-Point Unit. It handles single-precision and double-precision floating-point operations.

**SFS.** SPU Odd Fixed-Point Unit. It handles shuffle operations.

**SFX.** SPU Even Fixed-Point Unit. It handles arithmetic, logical, and shift operations.

**signal.** Information sent on a signal-notification channel. These channels are inbound (to an SPE) registers. They can be used by the PPE or other processor to send information to an SPE. Each SPE has two 32-bit signal-notification registers, each of which has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor. Unlike mailboxes, they can be configured for either one-to-one or many-to-one signalling.

These signals are unrelated to UNIX signals. See also *channel and mailbox*.

**signal notification.** See *signal*.

**SIMD.** Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

**SIMDize.** Transform scalar code to vector code.

**single-ported.** Single-ported memory allows only one access at a time.

**SLB.** Segment Lookaside Buffer. It is used to map an effective address (EA) to a virtual address (VA).

**SLS.** SPU Load and Store Unit. It handles loads, stores, and branch hints, and it includes the SPE's local store (LS).

**SMM.** Synergistic Memory Management Unit. It translates EAs to RAs in an SPU.

**snoop.** To compare an address on a bus with a tag in a cache, in order to detect operations that violate memory coherency. Also called *inquire*.

**SOA.** Structure of arrays. A method of organizing related data values. Also called parallel-array form. See also *AOS*.



**SPE.** Synergistic Processor Element. It includes an SPU, an MFC, and an LS.

**specific intrinsic.** A type of C and C++ language extension that maps one-to-one with a single SPU assembly instruction. All SPU specific intrinsics are named by prefacing the SPU assembly instruction with `si_`.

**SPE thread.** (a) A thread running on an SPE. Each such thread has its own 128 × 128-bit register file, program counter, and MFC Command Queues, and it can communicate with other execution units (or with effective-address memory through the MFC channel interface). (b) A thread scheduled and run on an SPE. A program has one or more SPE threads. Each thread has its own SPU local store (LS), register file, program counter, and MFC command queues.

**SPI.** Serial Peripheral Interface. Connects to pervasive logic elements.

**spat.** To replicate, as when a single scalar value is replicated across all elements of an SIMD vector.

**SPR.** Special-Purpose Register.

**SPU.** Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

**SPU ISA.** SPU Instruction Set Architecture. An SIMD instruction set executed in SPEs that is similar to the Vector/SIMD Multimedia Extension instruction set executed by the PPE.

**spulet.** A standalone SPU program that is managed by a PPE executive.

**SSC.** SPU Channel and DMA Unit. It handles all input and output functions for an SPU.

**SSE.** Single SIMD Extensions. An Intel instruction set.

**sticky bit.** A bit that is set by hardware and remains set until cleared by software.

**stub.** See *method stub*.

**supervisor mode.** See *privileged mode*.

**software-managed memory.** An SPE's local store (LS), which is filled from main memory using software-initiated DMA transfers. Although most processors reduce latency to memory by using caches, an SPE uses its DMA-filled LS. This approach provides a high degree of control for real-time programming. However, this approach is advantageous only if the DMA transfer-size is sufficiently large and the DMA command is issued well before the data is needed, because the latency and instruction overhead associated with DMA transfers exceeds the latency of servicing a cache miss.

**synchronization.** The order in which storage accesses are performed.

**system storage.** All program-addressable memory in a system, including main storage (main memory), the PPE's L1 and L2 caches, and the SPE's local store (LS).

**tag group.** A group of DMA commands. Each DMA command is tagged with a 5-bit tag group identifier. Software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups. All DMA commands except **getllar**, **putllc**, and **putlluc** are associated with a Tag Group.

**Tcl.** Tool Command Language. An interpreted script language used to develop GUIs, application prototypes, Common Gateway Interface (CGI) scripts, and other scripts.

**TG.** A tag-group ID parameter in an MFC command.

**thread.** A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously, if each thread has its own architectural state (registers, program counter, flags, and other program-visible state).

Each SPE can support only a single thread at any one time. The multiple SPEs can simultaneously support multiple threads. The PPE supports two threads at any one time, without the need for software to create the threads. The PPE does this by duplicating architectural state.

**throughput.** The number of instructions completed per cycle. A high-throughput application design seeks to keep pipelines full. To improve throughput, functions may need to do nontrivial amounts of work and operate with good locality of data reference.

**TKM.** Token Management Unit. Part of the Element Interconnect Bus (EIB) that software can program to regulate the rate at which particular devices are allowed to make EIB command requests.

**TLB.** Translation Lookaside Buffer. An on-chip cache that translates virtual addresses (VAs) to real addresses (RAs). A TLB caches page-table entries for the most recently accessed pages, thereby eliminating the necessity to access the page table from memory during load/store operations.

**transfer element.** See *DMA list*.

**TS.** The transfer-size parameter in an MFC command.

**unified register file.** A register file in which all data types—integer, single-precision and double-floating-point, logicals, bytes, and others—use the same register file. The SPEs (but not the PPE) have unified register files.

**user mode.** The mode in which problem state software runs. See also *problem state*.

**VA.** Virtual Address.

**vector.** An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called *SIMD operands* or *packed operands*.

**Vector/SIMD .** The SIMD instruction set of the PowerPC Architecture, supported on the PPE.

**virtual address.** An address to the virtual-memory space, which is much larger than the physical address space and includes pages stored on disk. It is translated from an effective address (EA) by a segmentation mechanism and used by the paging mechanism to obtain the real address (RA). The maximum size of the virtual-address space is  $2^{65}$  bytes.

**virtual memory.** The address space created using the memory management facilities of a processor.

**virtual mode.** The mode in which virtual-address translation is enabled.

**VPN.** Virtual Page Number. The number of the page in virtual memory.

**VXU.** Vector/SIMD Multimedia Extension unit.

**word.** Four bytes.

**workload.** A set of code samples in the SDK that characterizes the performance of the architecture, algorithms, libraries, tools, and compilers.

**writeback flag.** A flag written by an SPE to main storage that notifies the PPE of a specific event.

**XDR.** Rambus XDR DRAM memory technology.

**XIO.** A Rambus XDR Extreme Data Rate I/O (XIO) memory channel.

**xlC.** The IBM optimizing C compiler.



---

# Index

## Special characters

`__builtin_expect` 75  
`_align_hint` directive 75

## A

ABI (Application Binary Interface) 65, 84  
Accelerated Library Framework 127  
addressing modes 22, 25  
aligned attribute 75  
AOS (array of structures) 62, 104  
Application Binary Interface (ABI) 65, 84  
architecture of the PPE 6  
array of structures (AOS) 62, 104  
array-summing example 34  
asymmetric-thread runtime model 126  
auto-vectorizing compiler 94

## B

barriers and fences 79  
basic blocks 95  
BHT (branch history table) 97  
big-endian byte and bit ordering 9  
big-endian ordering 60  
blocking channel 57  
branch hints 95, 97  
branch history table (BHT) 97  
branch mispredicts 97  
branch target instruction cache (BTIC) 97  
BTIC (branch target instruction cache) 97  
built-Ins 66  
byte ordering 9

## C

C/C++ language extensions 25  
CBEA (Cell Broadband Engine Architecture) 1  
Cell Application Frameworks 127  
Cell Broadband Engine Architecture (CBEA) 1, 4, 19  
Cell Broadband Engine Architecture, overview 1  
Cell Broadband Engine Linux task 12  
Cell Broadband Engine program, performance optimization 112  
channel domains 35  
checkpoints 150  
clamping 25  
command-line interface 131  
command-line mode 147  
commands, DMA 76  
commands, synchronization 76

compatibility with existing PowerPC code 23  
compiler directives 75  
composite intrinsics 66  
computation-acceleration model 125  
condition register (CR) 19  
console window 131  
control plane 4, 24  
count register (CTR) 19  
CR (condition register) 19  
CTR (count register) 19  
cycle-accurate simulation 129, 146

## D

data plane 3, 4, 24  
data types 28  
debugging 48  
decrementer (timer) 54  
denormals 51  
dependencies 115  
device-extension model 124  
Direct Memory Access Controller (DMAC) 54  
directives 75  
directory structure 41  
DMA command, tag groups 79  
DMA commands 37, 76  
DMA commands, get 76  
DMA commands, put 76  
DMA list, creating 91  
DMA transfers 35, 49, 80, 115  
DMA-list transfers 91  
DMA-list transfers, programming example 91  
DMAC (Direct Memory Access Controller) 54  
dual-issue 53  
dynamic branch prediction 97  
dynamic timing analysis 114  
dynamic timing analysis, on the IBM Full System Simulator 119

## E

EA (effective address) 19, 22, 35, 65, 84, 130  
ECC (error-correcting code) 53  
Eclipse IDE 41  
effective address (EA) 19, 22, 35, 65, 84, 130  
Effective-to-Real Address Translation 153  
Element Interconnect Bus (EIB) 4, 19  
emitters 153  
error-correcting code (ECC) 53  
Euler, particle-system simulation 102  
example program, tpa1 152  
example, array-summing 34

example, using Vector instructions in PPE program 33  
executables 41

## F

fast mode 141  
fetch group 53  
fixed-point exception register (XER) 19  
floating-point registers (FPRs) 19  
floating-point status and control register (FPSCR) 19  
Floating-Point Status and Control Register (FPSCR) 51  
FPRs 134  
FPRs (floating-point registers) 19  
FPSCR (floating-point status and control register) 19  
FPSCR (Floating-Point Status and Control Register) 51  
fscrrd instruction 51  
fscrwr instruction 51  
Full System Simulator, for the Cell Broadband Engine 112  
function offload model 123  
function-inlining 95  
functional simulation 129  
functions, running on the SPEs 124

## G

general-purpose registers (GPRs) 19  
General-Purpose Registers (GPRs) 18  
generic intrinsics 29, 66  
get commands 76  
glossary 165  
GPRs 134  
GPRs (general-purpose registers) 19  
GPRs (General-Purpose Registers) 18  
Graphical User Interface 132  
graphics rounding mode 100  
GUI buttons 141

## H

HBR (hint for branch) 97  
hint for branch (HBR) 97  
hint-trigger address 97

## I

I/O devices 4, 19  
IBM Full System Simulator for the Cell Broadband Engine 45, 129  
IBM Full System Simulator, dynamic timing analysis 119  
IDL (Interface Definition Language) 124  
IEEE 754 51, 100  
in-order 6, 23, 53, 55

- instruction types 23, 25
- inter-loop dependencies 115
- Interface Definition Language (IDL) 124
- intrinsics 11, 25, 26, 33, 64
- intrinsics, casting 66
- intrinsics, not available as generic intrinsics 66

## J

- Joint Software Reference Environment (JSRE) 66
- JSRE (Joint Software Reference Environment) 66

## L

- link register (LR) 19
- Linux command prompt 131
- Linux mode 130
- Linux task 12
- Linux threads 12
- list element 85, 90, 92
- Load-Exec 136
- local store domains 35
- loop unrolling 115
- loop-carried dependencies 115
- loop-unrolling 95
- LR (Link Register) 19
- LS considerations 100

## M

- M:N thread model 13
- macro translation 100
- macros (MFC), input and output 80
- mailboxes (queues) 58
- main storage domain 35
- many-to-one signaling 59
- mapping PPE to SPEs 99
- Memory Flow Controller (MFC) 35, 49
- memory use (limitation of) 3
- method stub 124
- MFC (Memory Flow Controller) 35, 49
- MFC command-parameter registers 35
- microthreads 126
- model instruction 136
- model pipeline 136
- monitoring performance 146
- multi-stage pipeline model 13
- multibuffering 115

## N

- NaN (not-a-number) 51
- non-blocking channel 57
- not-a-number (NaN) 51

## O

- one-to-one signaling 59
- overlays 127

## P

- packed operands 10
- parallel-array form 104
- parallel-stages model 13
- partitioning 13
- PCAddressing 134
- PCC Core window 134
- PCCCore 134
- PCTrack 134
- performance of microprocessors 3, 42
- performance simulation 129
- performance statistics, displaying 147
- performance, SPU 155
- pipeline mode 136
- porting SIMD code from PPE to SPEs 98
- power use (limitation of) 3
- PowerPC Architecture Vector/SIMD Multimedia Extension 1
- PowerPC instructions 22
- PowerPC Processor Element (PPE) 1, 4, 6, 19
- PPE (PowerPC Processor Element) 6
- PPE and SPE, architectural differences 72
- PPE code, parallelize for execution across multiple SPEs 110
- PPE code, porting for execution on SPE 106
- PPE instruction set 21
- PPE registers 19
- PPE thread 12
- PPE-centric models 13
- PPE, and the SPEs 35
- ppu-gdb 48
- precise trap 51
- precision 100
- predicate intrinsics 29
- predication 95
- preferred slot 60, 72
- problem-state registers 19
- procedures, running on the SPEs 124
- processor frequency (limitation of) 3
- profile checkpoints 150
- programming 9
- programming Models 123
- programming tips 121
- put commands 76

## R

- registers 19
- remote procedure call (RPC) model 123, 124
- rounding result 51
- RPC (remote procedure call) 123, 124
- runtime environment 13

## S

- saturation 25
- scalar code, programming example 103
- scalar intrinsics (PPE-specific) 26
- scalar loads 115
- scalar operands 72
- scatter-gather 85, 90, 92
- SDK (software development kit) 16

- Segment Lookaside Buffer 153
- select-bits (selb) instruction 95
- select-bits intrinsic 95
- service model 13
- SFP (SPU Floating-Point Unit) 51
- shared-memory multiprocessor model 125
- signal notification 40
- signal-notification channels 59
- signal-notification facility (SPE) 124
- signals 59, 124
- SIMD (single-instruction, multiple-data vectorization) 10
- SIMD code, porting from PPE to SPEs 98
- SIMDization 10
- SIMDize 102
- simulation 129
- simulation panel 133
- simulator basics 130
- simulator command window 131
- simulator for the Cell Broadband Engine 45
- simulator prompt 147
- single-instruction, multiple-data vectorization (SIMD) 10
- SOA (structure of arrays) 62, 104
- software development kit (SDK) 16
- Sony, Toshiba, and IBM (STI) 1
- SPE (Synergistic Processor Element) 7, 49
- SPE overlays 127
- SPE programming 38
- SPE thread 12, 126
- spe\_context\_create 38
- spe\_context\_run 38
- spe\_program\_load 38
- SPE-centric model 13
- specific intrinsics 29, 66
- SPU (Synergistic Processor Unit) 49
- SPU Floating-Point Unit (SFP) 51
- SPU Instruction Set Architecture (SPU ISA) 60
- SPU intrinsics 64
- SPU ISA (SPU Instruction Set Architecture) 60
- SPU performance 155
- spu\_mffpscr intrinsic 51
- spu\_mtfpscr intrinsic 51
- spu\_timing, static timing analysis 118
- spu-gdb 48
- spu-timing, static timing analyzer 112
- SPUChannel 136
- SPUCore 136
- SPUMemory 136
- SPUStats 136
- SPUTrack 136
- standalone mode 130
- static branch prediction 97
- static timing analysis, spu\_timing 118
- STI (Sony, Toshiba, and IBM) 1
- sticky bit 51
- storage barriers 40
- storage domains 35
- streaming model 125
- structure of arrays (SOA) 62, 104
- stub 124

- suffixes 76
- synchronization commands 76
- Synergistic Processor Element (SPE) 1, 7, 49
- Synergistic Processor Elements (SPEs) 4, 19
- Synergistic Processor Unit (SPU) 49

## T

- tag groups (DMA command) 79
- tag manager 80
- task 12
- Tcl (Tool Command Language)
  - commands 129
- thread 12, 126
- thread model 12
- thread model (M:N) 13
- timing analysis 114
- timing simulation 129
- Tool Command Language (Tcl) 129
- transfer elements 76, 85, 90, 92
- truncation 51

## U

- unified register file 50
- unmappable Constructs 99
- user threads 126
- user-mode thread model 126

## V

- vector 10
- vector data types 60
- Vector instructions in PPE program,
  - example of 33
- vector intrinsics 29
- vector multimedia registers (VMRs) 19
- vector operands 10
- vector save register (VRSAGE) 19
- vector status and control register (VSCR) 19
- vector types 28
- vector-across form 104
- vector/SIMD multimedia extension
  - intrinsics 33
- Vector/SIMD Multimedia Extension
  - vector types 28
- vector/SIMD multimedia extensions 4, 6, 19
- vectorization 10
- VMRs (vector multimedia registers) 19
- VRSAGE (vector save register) 19
- VSCR (vector status and control register) 19

## X

- XER (fixed-point exception register) 19

## Z

- zero result 51





---

## Readers' Comments — We'd Like to Hear from You

Software Development Kit for Multicore Acceleration Version 3.1  
Programming Tutorial  
Version 3.1

Publication No. SC33-8410-01

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:

- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +49-7031-16-3456
- Send your comments via e-mail to: [eservdoc@de.ibm.com](mailto:eservdoc@de.ibm.com)
- Send a note from the web page:

If you would like a response from IBM, please fill in the following information:

\_\_\_\_\_

Name

\_\_\_\_\_

Address

\_\_\_\_\_

Company or Organization

\_\_\_\_\_

Phone No.

\_\_\_\_\_

E-mail address



Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

# BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Entwicklung GmbH  
Department 3248  
Schoenaicherstrasse 220  
D -71032 Boeblingen  
Germany



Fold and Tape

Please do not staple

Fold and Tape





Printed in USA

SC33-8410-01

