# Design of Image Processing Algorithms for Multicore Processor: Implementation and Performance Analysis on Cell Broadband Engine

By

**RUCHANDANI KAPIL V.**

**(06MCE018)**


**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**
**INSTITUTE OF TECHNOLOGY**
**NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY**
**AHMEDABAD 382 481**
**MAY 2008**

**Major Project**

**On**

# Design of Image Processing Algorithms for Multicore Processor: Implementation and Performance Analysis on Cell Broadband Engine

Submitted in partial fulfillment of the requirements

For the degree of

**Master of Technology in Computer Science & Engineering**

By

**Ruchandani Kapil V.**
**(06MCE018)**

Under Guidance of

**Dr. S.N. Pradhan**


**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**
**INSTITUTE OF TECHNOLOGY**
**NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY**
**Ahmedabad 382481**
**May 2008**


This is to certify that Dissertation entitled

# Design of Image Processing Algorithms for Multicore Processor: Implementation and Performance Analysis on Cell Broadband Engine

Submitted by

Ruchandani Kapil V

has been accepted towards fulfillment of the requirement

for the degree of

Master of Technology in Computer Science & Engineering

| | |
|---|---|
| Dr. S. N. Pradhan | Prof. D. J. Patel |
| P.G.Coordinator | Head of the Department |

Prof. A. B. Patel

Director, Institute of Technology

# *CERTIFICATE*

This is to certify that the Major Project entitled **"Design of Image Processing Algorithms for Multicore Processor: Implementation and Performance Analysis on Cell Broadband Engine"** submitted by **Mr. Ruchandani Kapil (06MCE018)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science & Engineering, Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any Master degree.

**Dr. S.N.Pradhan**
Project Guide,
P. G. Coordinator,
Department of Computer Science & Engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Date:-

# ACKNOWLEDGEMENT

It gives me immense pleasure in expressing thanks and profound gratitude to **Dr**. **S.N Pradhan**, P.G. Coordinator, Computer Science & Engineering Department, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout my Major project. I am heartily thankful to him for his precious time, suggestions and sorting out the difficulties of my topic that helped me a lot during this study.

I would like to give my special thanks to **Prof. D.J Patel**, Head, Computer Science & Engineering Department, Nirma University, Ahmedabad for his encouragement and motivation throughout the Major Project. I am also thankful to **Prof. A. B. Patel**, Director, Institute of Technology, Nirma University, Ahmedabad for his kind support in all respect during my study.

I am thankful to all faculty members of Computer Science & Engineering Department, Nirma University, Ahmedabad for their special attention and suggestions towards the project work.

**Ruchandani Kapil**
Roll No. 06MCE018

# ABSTRACT

Enhancement of CPU processing power and speed has been a priority issue in the past several years. Conventional approaches to satisfy this demand were to improve the performance of the core singularly. However the CPU design became much more complex, causing problems as increases in power consumption, over-heating and manufacturing cost. That's why world is moving towards multi-core configurations. Multi-core architecture offers higher performance at same power consumption and manufacturing cost.

The Cell Broadband Engine Architecture (CBEA) is a multi-core architecture, including a 64-bit PowerPC Processor Element and eight Synergistic Processor Elements which offers a raw compute power of up to 200 GFlops per 3.2 GHz chip. The Cell bears a huge potential for compute intensive applications like Image processing applications, scientific applications etc.

Image processing applications such as filtering, edge detection, correlation etc. require large computation power. Processing large images on a single core system consumes a lot of time. Such applications can be ported on CBEA and performance can be improved. However it requires addressing many challenges for implementing such applications for CBEA.

The goal of this dissertation is to implement image processing algorithms for CELL Broadband Engine, address the challenges occurring because of multiple cores and achieve higher performance than conventional systems.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| BEI | Cell Broadband Engine Interface |
| CBE | Cell Broadband Engine |
| CMP | Chip Multiprocessors |
| DCT | Discrete Cosine Transform |
| DIP | Digital Image Processing |
| DMA | Direct Memory Access |
| EIB | Elementary Interconnect Bus |
| FXU | Fixed-Point Unit |
| IU | Instruction Unit |
| LS | Local Store |
| LSU | Load and Store Unit |
| MFC | Memory Flow Controller |
| MIC | Memory Interface Controller |
| MMU | Memory Management Unit |
| P-4 | Pentium 4 |
| PPE | PowerPC Processor Element |
| PPSS | PowerPC Processor Storage Subsystem |
| PPU | PowerPC Processor Unit |
| PS3 | Playstation 3 |
| RISC | Reduced Instruction Set Computer |
| SCN | SPU Control Unit |
| SDK | Software Development Kit |
| SFP | SPU Floating-Point Unit |
| SFS | SPU Odd Fixed-Point Unit |
| SFX | SPU Even Fixed-Point Unit |
| SIMD | Single Instruction Multiple Data |
| SLS | SPU Load and Store Unit |

| | |
|---|---|
| SNR | Signal Notification Registers |
| SPE | Synergistic Processor Element |
| SPU | Synergistic Processor Unit |
| SSC | SPU Channel and DMA Unit |
| VSU | Vector/Scalar Unit |
| YDL | Yellow Dog Linux |

# 1 INTRODUCTION

## 1.1 General

Image processing applications such as filtering, edge detection, correlation etc. require large computation power. Processing large images on a system with single processors consumes a lot of time. Using DSPs we can improve the performance of such applications. However performance gain is limited. We can use multiprocessor or multicore system to improve performance of such applications.

Cell Broadband Engine (Cell BE) which is a multicore system can be used for this purpose. Cell BE is the result of collaboration between Sony, Toshiba, and IBM. Although the Cell BE was initially intended for application in game consoles and media-rich consumer-electronics devices such as high-definition televisions, the architecture can be used for compute intensive applications such as satellite image processing, medical imaging, digital media, communications, and some scientific applications to improve performance. The Cell BE performs significantly faster than existing CPUs for many applications. IBM provides the Cell SDK for programming and simulating the cell architecture.

As exciting as it may sound, using the PS3 for scientific computing is a bumpy ride. Parallel programming models for multi-core processors are in their infancy, and standardized APIs are not even on the horizon. As a result, presently, only hand-written code fully exploits the hardware capabilities of the CELL processor. Ultimately, the suitability of the PS3 platform for scientific computing is most heavily impaired by the devastating disproportion between the processing power of the processor and the crippling slowness of the interconnect. Nevertheless, the CELL processor is a revolutionary chip, delivering ground-breaking performance and now available in an affordable package. Programming such a multicore system is

a challenging task. To make full use of the architecture potential the algorithm design must be done carefully. All the cores of Cell BE are SIMD based architecture cores. Hence Cell BE provides two levels of parallelism. This needs to be taken care of while designing the algorithm.

This report shows the details of the Cell architecture and design of Image Processing algorithms for Cell BE.

### 1.2 Motivation

Enhancement of CPU processing power and speed has been a priority issue in the past several years due to the widespread use of multimedia applications, medical imaging, scientific applications as well as the advent of 3D games and other advanced video applications that demand higher-speed processing of massive audio/video data to offer higher-quality entertainment.

Conventional approaches to satisfy this demand counted on the improvement of the performance of the core singularly used in each CPU. However, they soon proved to have limitations – the CPU design became much more complex, causing problems as increases in power consumption, over-heating and manufacturing cost. That's why CPU manufacturers set about developing multi-core configurations that allow multiple cores to run in parallel on a single chip to realize further performance upgrades.

The Cell employs the heterogeneous multi-core processor configuration. Instead of conventional multi-application cores, the Cell uses two types of cores optimized for different applications: a control-intensive processor core that excels in handling frequent thread switching and thus is suited for use with the operating system, and a simple compute-intensive processor core that addresses multimedia processing. With this configuration, each core can maintain its processing performance. The core structure can also be simplified drastically.

Initially the Cell project was intended for Playstation3 so the main concentration of the developers of Cell processor was on the better performance on gaming consoles. But later it was realized that the immense computing power of Cell Broadband engine can be utilized for other compute intensive scientific fields. Though the programmability of the this processor architecture is a bit complex but its worth taking pains to utilize its computing power. A lot of work is being done in this direction and a lot of success has been achieved using this processor architecture for diverse application areas.

Image processing applications such as filtering, edge detection, correlation etc. require large computation power. Processing large images on a system with single processors consumes a lot of time. In this thesis work some Image processing algorithms, which consumes a lot of processing power, are implemented on CELL processor so as to take full advantage of the high processing power provided by its novice architecture. The reason for selecting image processing as the area of application is that image processing algorithms provide a lot of scope for parallelism, which is the hallmark of Cell processor architecture. Cell architecture provides two levels of parallelism, one at the inter-SPU level which is provided by the 8 SPU's and other at the intra-SPU level which is provided by the vector support. Image processing algorithms are very well suited to exploit these two levels of parallelism provided by Cell BE. Because the image can be easily broken into smaller images and also in image processing algorithms we generally have to perform the same operation over each and every pixel which gives us a scope for utilizing vector operations.

### 1.3 Scope of the Work

The main objective of this thesis is to modify the existing scalar image processing algorithms to utilize the Cell processor architecture and to claim the performance improvement over the traditional scalar versions of it. But since the architecture of CELL processor is quite new and different, the

existing Image processing algorithms are designed for singlecore processors and would not be able to exploit the computing power provided by CELL B.E. So these algorithms have to be redesigned and restructured in such a way so that all nine cores can be fully utilized. Also the SIMD feature provided by all these cores has to be taken care of while designing the algorithms. This makes the task of programming for CELL processor quite complex and the onus is on the programmer to restructure the algorithms carefully.

The scope of this thesis work encompasses the detailed architectural study and programming methodology of the Cell architecture. Based on this knowledge two image processing algorithms are selected: Spatial Domain Filter and various stages of Baseline JPEG compression. These algorithms are then worked upon and redesigned to execute on Cell B.E so as to give a performance Speedup over the scalar versions of these algorithms executed on conventional single core processors.

### 1.4 Outline of Report

- Chapter 2 presents the details of the Cell architecture like detail architecture of the processor cores used in it, details of memory interface controller and bus connecting all the elements. It also presents some advantages of Cell architecture and gives specifications of some systems based on Cell architecture.
- Chapter 3 presents contents of the SDK used for programming cell and lists SDK installation steps.
- Chapter 4 describes the programming methods for Cell. It also describes some points to consider while partitioning the application and data between various cores and how the program control and data flows between processor cores. In the end it gives some advanced techniques for programming the cell.
- Chapter 5 explains various Image Processing algorithms and proposed algorithms which are optimized for CELL Broadband Engine. It also shows implementation details and performance results.

# 2 THE CELL ARCHITECTURE

## 2.1 History of the Cell Project

Cell represents a revolutionary extension of conventional microprocessor architecture and organization. This report discusses the history of the project, the program objectives and challenges, the design concept, the architecture and programming models, and the implementation.

Initial discussion on the collaborative effort to develop Cell began with support from CEOs from the Sony and IBM companies: Sony as a content provider and IBM as a leading-edge technology and server company. Collaboration was initiated among SCEI (Sony Computer Entertainment Incorporated), IBM, for microprocessor development, and Toshiba, as a development and high-volume manufacturing technology partner. This led to high-level architectural discussions among the three companies during the summer of 2000. During a critical meeting in Tokyo, it was determined that traditional architectural organizations would not deliver the computational power that SCEI sought for their future interactive needs. SCEI brought to the discussions a vision to achieve 1,000 times the performance of PlayStation2. At this stage of the interaction, the IBM Research Division became involved for the purpose of exploring new organizational approaches to the design  IBM process technology was also involved, contributing state-of-the-art 90-nm process with silicon-on insulator (SOI), low-k dielectrics, and copper interconnects. During this interaction, a wide variety of multi core proposals were discussed, ranging from conventional chip multiprocessors (CMPs) to dataflow oriented multiprocessors. By the end of 2000 an architectural concept had been agreed on, that combined the 64-bit Power Architecture with memory flow control and ‘‘synergistic’’ processors in order to provide the required computational density and power efficiency. After several months of architectural discussion and contract negotiations, the STI (SCEI–Toshiba–IBM) Design Center was formally opened in Austin, Texas, on

March 9, 2001. The STI Design Center represented a joint investment in design of about $400,000,000.Separate joint collaborations were also set in place for process technology development. A number of key elements were employed to drive the success of the Cell multiprocessor design. First, a holistic design approach was used, encompassing processor architecture, hardware implementation, system structures, and software programming models. Second, the design center staffed key leadership positions from various IBM sites. Third, the design incorporated many flexible elements ranging from reprogrammable synergistic processors to reconfigurable I/O interfaces in order to support many systems configurations with one high-volume chip. Although the STI design center for this ambitious, large-scale project was based in Austin (with IBM, the Sony Group, and Toshiba as partners), many other IBM sites were also involved and were critical to the project.

## 2.2 Architectural Overview

The Cell Broadband Engine is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. There are two types of processors in it: the PowerPC Processor Element (PPE), and the Synergistic Processor Element (SPE). The Cell Broadband Engine has one PPE and eight SPEs [1], [2], [4]. The first type of processor element, the PPE, 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 second type of processor element, the SPE, 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 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.

Figure 2-1 shows a block diagram of the Cell Broadband Engine.

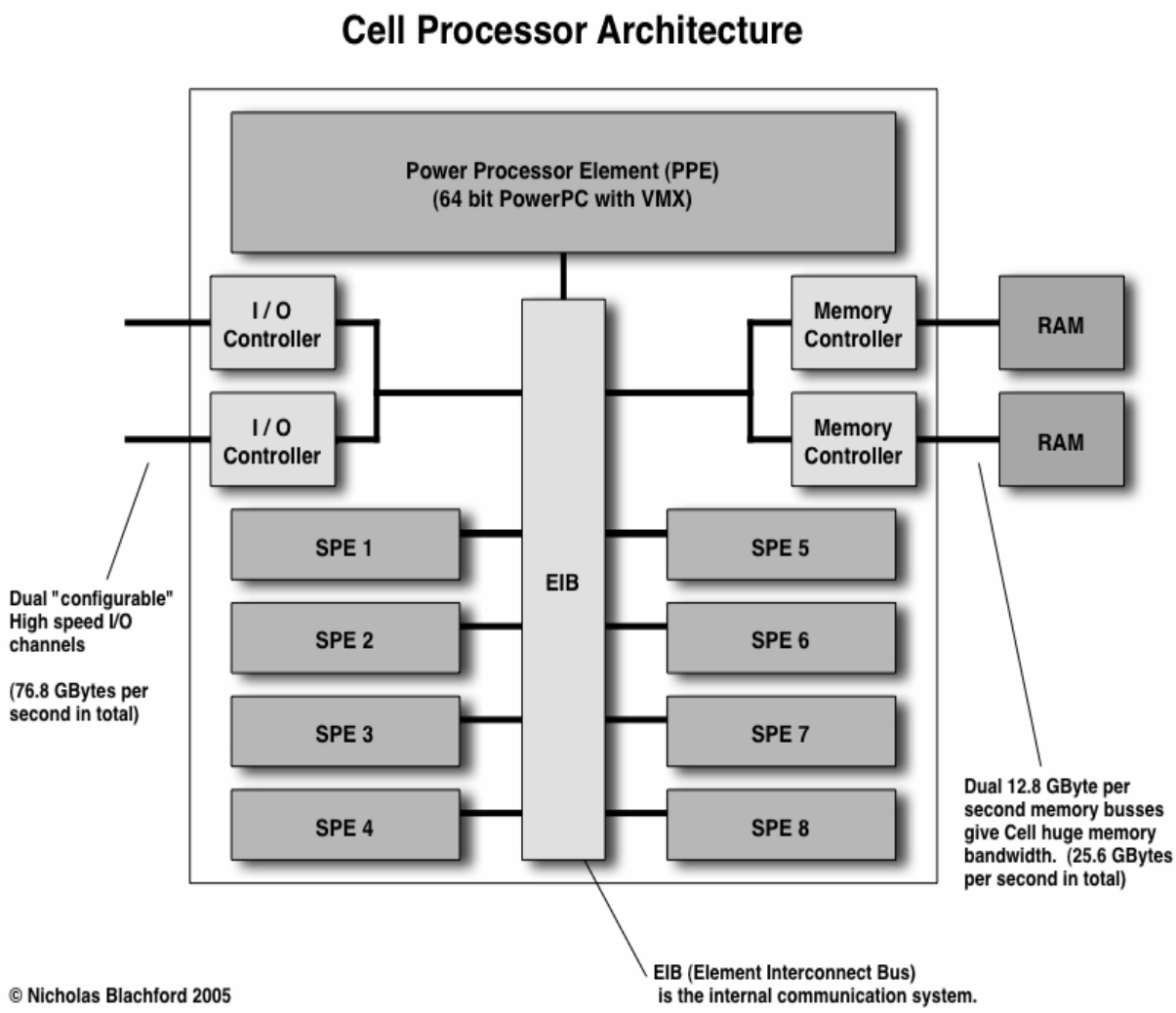


Figure 2-1 Cell Broadband Engine Overview

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. A significant difference between the SPE and the PPE 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). However, 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.

### 2.3 PowerPC Processor Element (PPE)

The PPE is the main processor. It contains a 64-bit PowerPC Architecture reduced instruction set computer (RISC) core. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It supports both the PowerPC instruction set and the Vector/SIMD Multimedia Extension instruction set. The PPE contains two main units: the PowerPC Processor Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS), shown in figure 2-2.

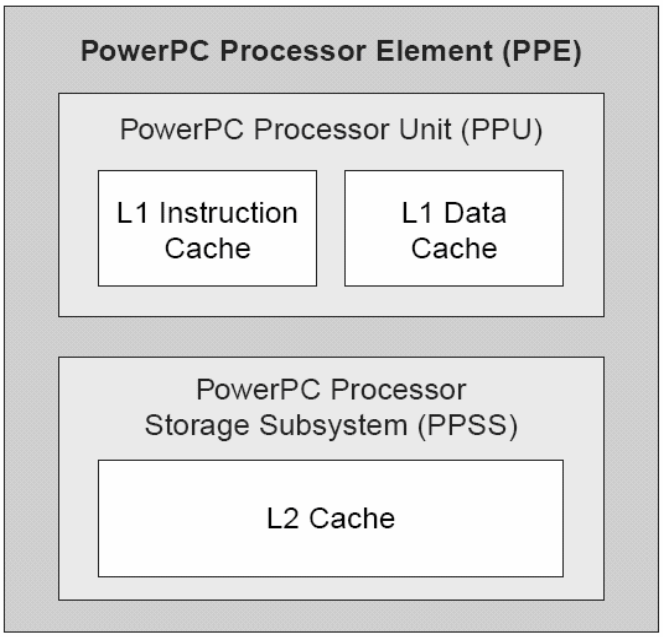


Figure 2-2 PPE Block Diagram

### 2.3.1 PowerPC Processor Unit (PPU)

The PPU is the processing unit in the PPE. It contains six execution units. It also contains a primary cache comprised of a 32KB instruction cache and a 32KB data cache. The PPU executes the PowerPC Architecture instruction set and the Vector/SIMD Multimedia Extension instructions. It has duplicate sets of the PowerPC and vector user-state register files (one set for each thread) plus one set of the following functional units:

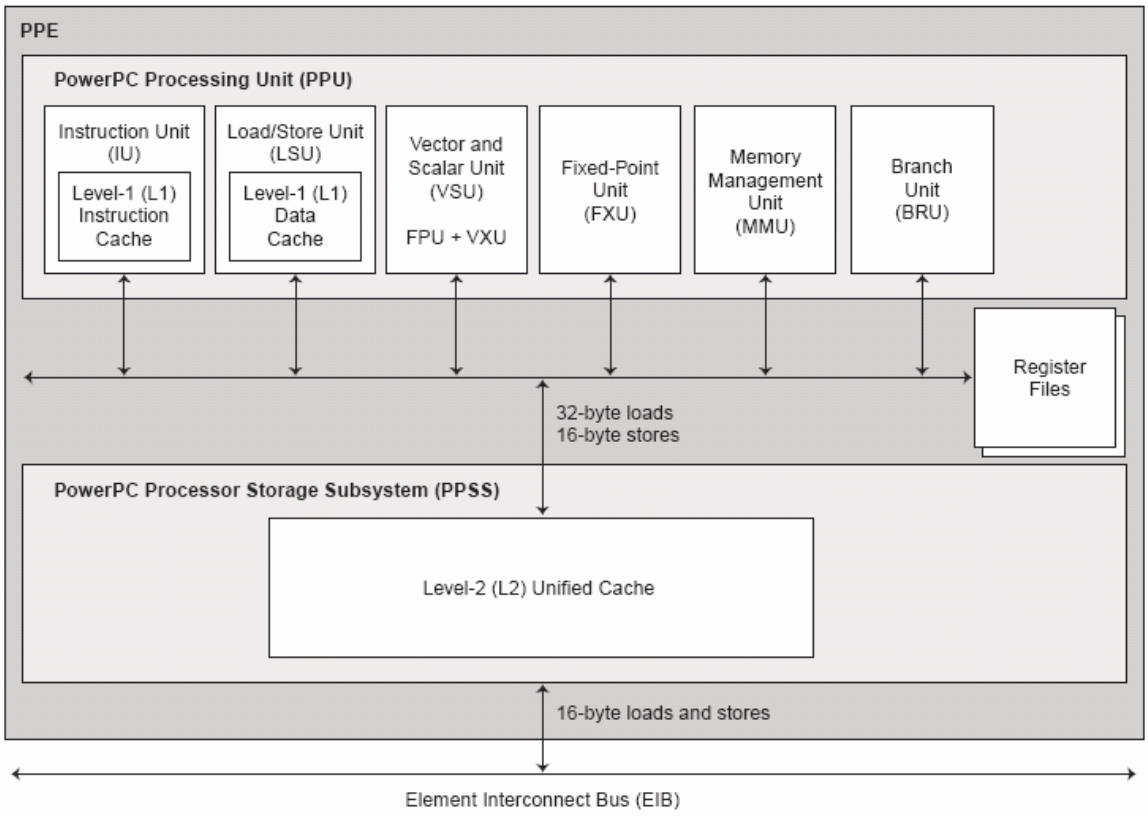


Figure 2-3. PPE Functional Units

- **Instruction Unit (IU):** The IU performs the instruction-fetch, decode, dispatch, issue, branch, and completion portions of execution. It contains the L1 instruction cache, which is 32 KB, 2- way set-associative, parity protected. The cache-line size is 128 bytes.
- **Load and Store Unit (LSU):** The LSU performs all data accesses, including execution of load and store instructions. It contains the L1

data cache, which is 32 KB, 4-way set-associative, write-through, and parity protected. The cache-line size is 128 bytes.

- **Vector/Scalar Unit (VSU):** The VSU includes a Floating-Point Unit (FPU) and a 128-bit Vector/SIMD Multimedia Extension Unit (VXU), which together execute floating-point and Vector/SIMD Multimedia Extension instructions.
- **Fixed-Point Unit (FXU):** The FXU executes fixed-point (integer) operations, including add, multiply, divide, compare, shift, rotate, and logical instructions.
- **Memory Management Unit (MMU):** The MMU manages address translation for all memory accesses. It has a 64-entry Segment Lookaside Buffer (SLB) and 1024-entry, unified, parity protected Tanslation Lookaside Buffer (TLB).

### 2.3.2 PowerPC Processor Storage Subsystem (PPSS)

The PPSS handles all memory accesses by the PPU. The PPSS has a unified, 512-KB, 8-way set-associative, write-back L2 cache. Like the L1 caches, the cache-line size for the L2 is 128 bytes. The cache has a single-port read/write interface to main storage that supports eight software-managed data-prefetch streams. It includes the contents of the L1 data cache but is not guaranteed to contain the contents of the L1 instruction cache, and it provides fully coherent symmetric multiprocessor (SMP) support.

The PPSS performs data-prefetch for the PPU and bus arbitration. Traffic between the PPU and PPSS is supported by a 32-byte load port, and a 16-byte store port. The interface between the PPSS and EIB supports 16-byte load and 16-byte store buses.

### 2.3.3 PPE Registers

The PPE problem-state (user) registers are shown in Figure 2-4. All computational instructions operate on registers; no computational instructions modify main 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 loaded into a register, modified, and then stored back to the target location.

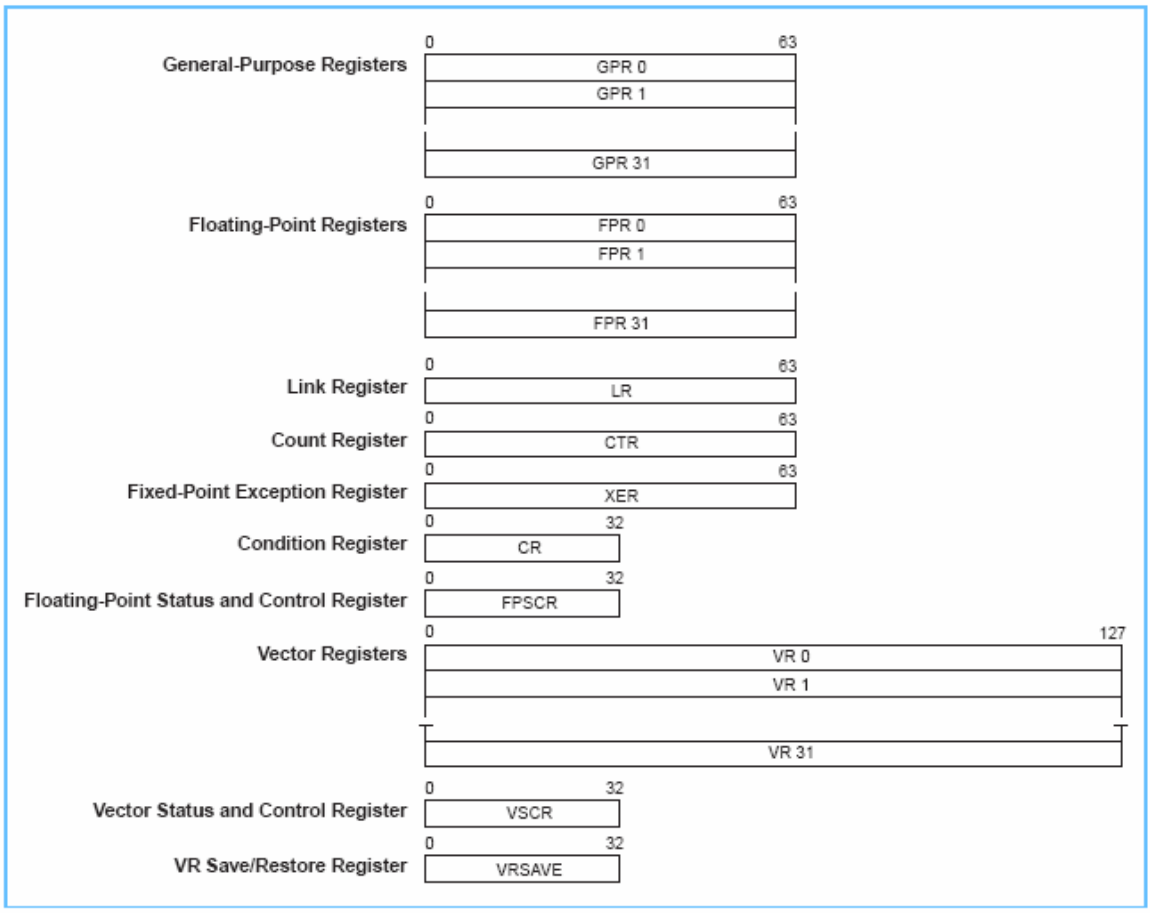


Figure 2-4. PPE User Register Set

### 2.3.4 Vector/SIMD Multimedia Extension Instructions

PPE supports PowerPC instructions as well as Vector/SIMD Multimedia Extension instructions. Vector/SIMD Multimedia Extension instructions can be freely mixed with PowerPC instructions in a single program. The 128-bit Vector/SIMD Multimedia Extension unit (VXU) operates concurrently with the PPU's 32-bit fixed-point unit (FXU) and 64-bit floating-point unit (FPU). Like PowerPC instructions, the Vector/SIMD Multimedia Extension instructions are four 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. 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.

A vector is 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 instructions operate on vector operands. Vectors are also called Single-Instruction, Multiple-Data (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.

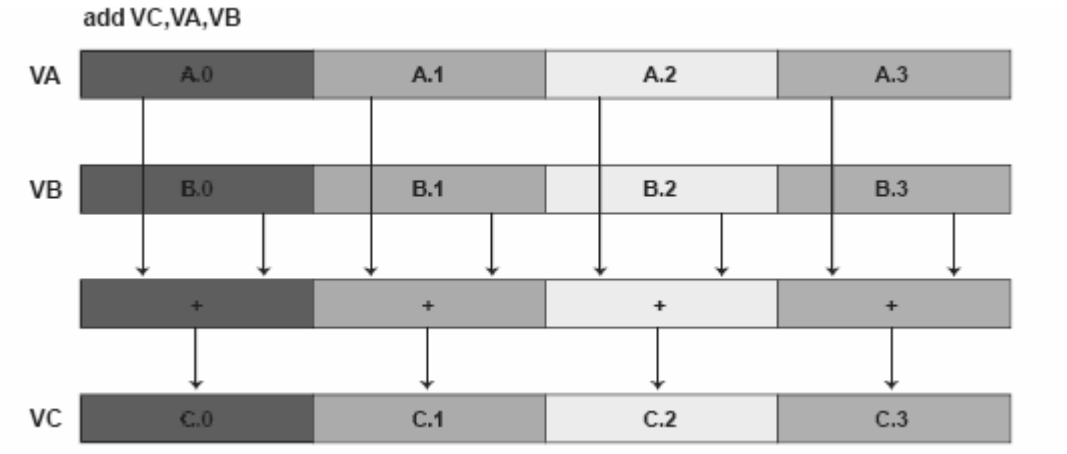


Figure 2-5. SIMD Add Operations

Support for SIMD operations is pervasive in the CBE processor. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instructions. 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 2-5 shows such an operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or 2 doublewords.

## 2.4 Synergistic Processor Elements (SPEs)

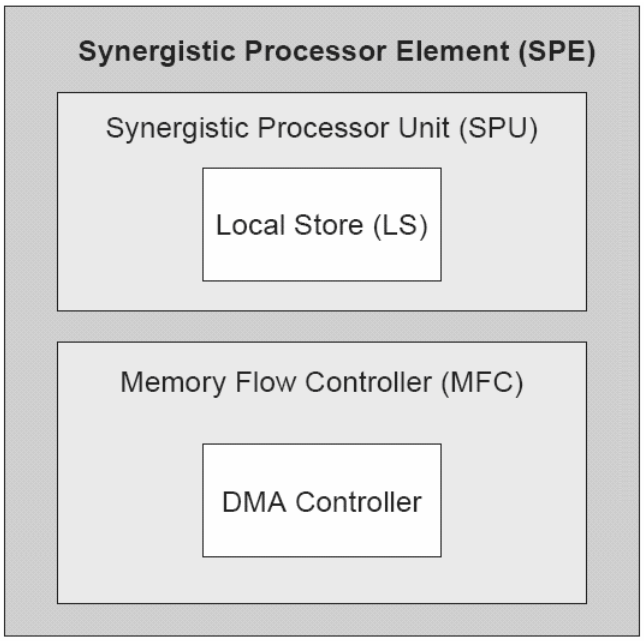


Figure 2-6. SPE Block Diagram

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 and their local stores. The SPEs are not intended to run an operating system. The SPE contains two main units: Synergistic Processor unit (SPU) and Memory Flow Control (MFC).

### 2.4.1 Synergistic Processor Unit (SPU)

Each SPE incorporates its own SPU to perform its allocated computational task. The SPUs use a unique instruction set designed specifically for their operations. It contains six execution units and a 256 KB local store. The SPU fetches instructions from its unified (instructions and data) 256-KB local store (LS), and it loads and stores data between its LS and its single register file for all data types, which has 128 registers, each 128 bits wide. The SPU has a DMA interface and a channel interface for communicating with its MFC, the PPE and other devices (including other SPEs). Each SPU is an independent processor element with its own program counter, optimized to run SPU programs. The SPU fills its LS by requesting DMA transfers from its MFC, which implements the DMA transfers using its DMA controller. Then, the SPU fetches and executes instructions from its LS, and it loads and stores data to and from its LS. The main SPU functional units are shown in Figure 2-7. These include the Synergistic Execution Unit (SXU), the LS, and the SPU Register File Unit (SRF).

The SPU can issue and complete up to two instructions per cycle, one on each of the two (odd and even) execution pipelines. Whether an instruction goes to the odd or even pipeline depends on the instruction type. The instruction type is also related to the execution unit that performs the function.

- **SPU Odd Fixed-Point Unit (SFS):** Executes quadword shift and rotates mask operations on bits, bytes, halfwords, and words, and shuffle operations on bytes.
- **SPU Even Fixed-Point Unit (SFX):** Executes arithmetic instructions, logical instructions, word SIMD shifts and rotates, floating-point compares, and floating-point reciprocal and reciprocal square-root estimates.

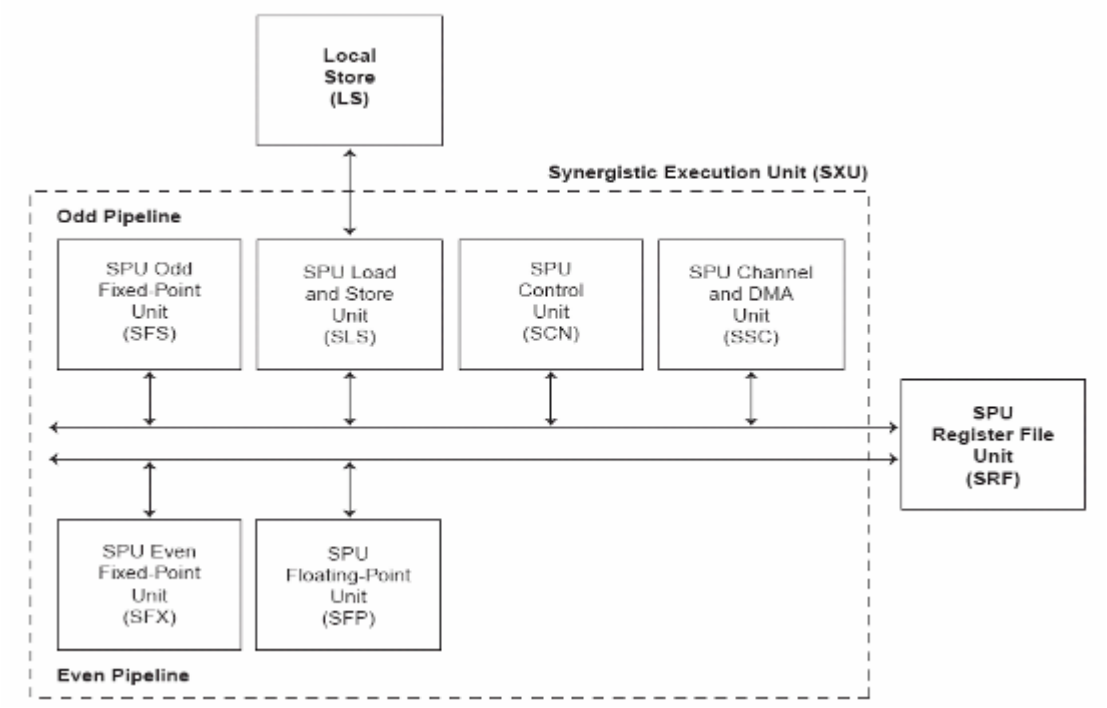


Figure 2-7. SPU Functional Units

- **SPU Floating-Point Unit (SFP):** Executes single- and double-precision floating-point instructions, integer multiplies and conversions, and byte operations. The SPU supports only 16-bit multiplies, so 32-bit multiplies are implemented in software using 16-bit multiplies.
- **SPU Load and Store Unit (SLS):** Executes load and store instructions and load branch-target- buffer (BTB) instructions. It also handles DMA requests to the LS.
- **SPU Control Unit (SCN):** Fetches and issues instructions to the two pipelines, executes branch instructions, arbitrates access to the LS and register file, and performs other control functions.
- **SPU Channel and DMA Unit (SSC):** Enables communication, data transfer, and control into and out of the SPU.

### 2.4.2 Local Store (LS)

The Local Store (LS) is a 256-KB, single-ported, noncaching memory. It

stores all instructions and data used by the SPU. It supports one access per cycle from either SPE software or DMA transfers. SPU instruction prefetches are 128 bytes per cycle. SPU data-access bandwidth is 16 bytes per cycle, quadword aligned. DMA-access bandwidth is 128 bytes per cycle. It is the only memory that can be referenced directly from the SPU. It's controlled by software running on SPU.

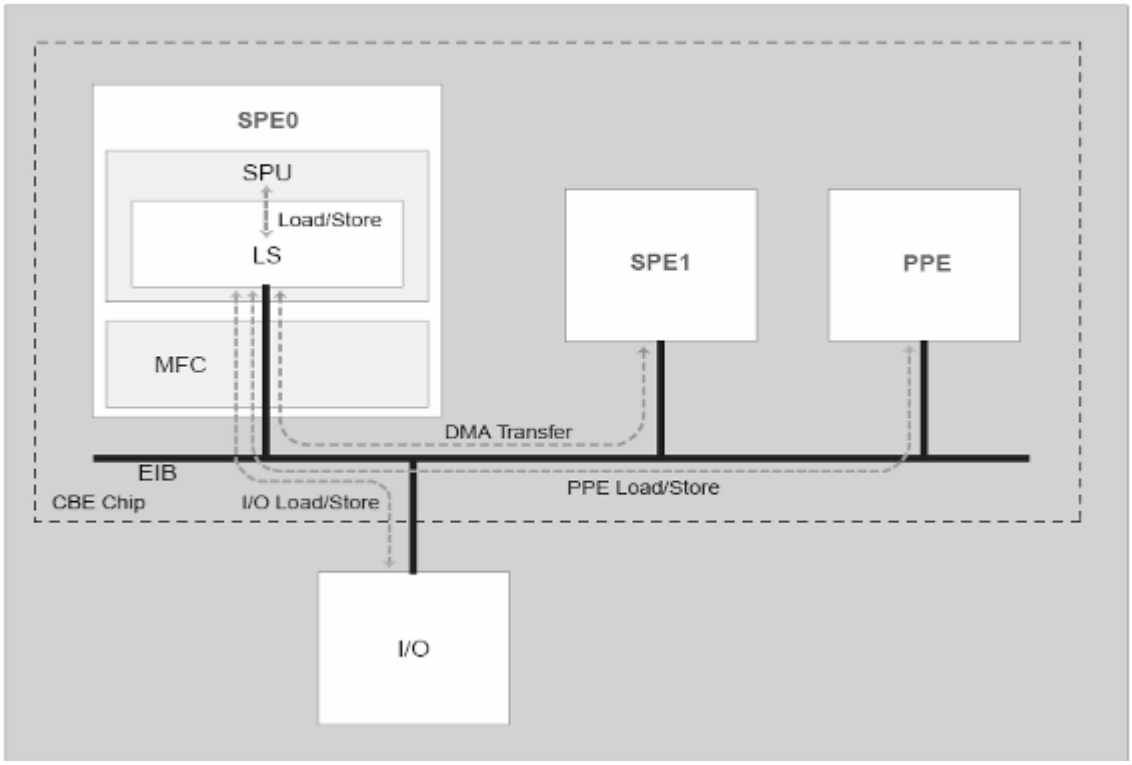


Figure 2-8. LS Access Methods

The SPU accesses its LS with load and store instructions, and it performs no address translation for such accesses. Privileged software on the PPE can assign effective-address aliases to LS. This enables the PPE and other SPEs to access the LS in the main-storage domain. The PPE performs such accesses with load and store instructions, without the need for DMA transfers. However other SPEs must use DMA transfers to access the LS in the main-storage domain. Figure 2-8 illustrates the methods by which an SPU, the PPE, other SPEs, and I/O devices access the SPU's associated LS,

when the LS has been aliased to the main storage domain.

### 2.4.3 Memory Flow Controller (MFC)

Each SPU has its own Memory Flow Controller (MFC). The MFC serves as the SPU's interface, by means of the Element Interconnect Bus (EIB), to main-storage and other processor elements and system devices. The MFC's primary role is to interface its LS with the main-storage domain. It does this by means of a DMA controller that moves instructions and data between its LS and main storage. The MFC also supports synchronization between main storage and the LS, and communication functions (such as mailbox and signal-notification messaging) with the PPE and other SPEs and devices.

### 2.4.4 Register File

The SPU's 128-entry, 128-bit register file stores all data types—integer, single- and double-precision floating-point, scalars, vectors, logical, bytes, and others [1]. It also stores return addresses, results of comparisons, and so forth. All computational instructions operate on registers—there are no computational instructions that modify storage. Figure 2-9 shows SPE register file.


Figure 2-9. SPE Register Set

## 2.5 Element Interconnect Bus (EIB)

The Element Interconnect Bus (EIB) is the communication path for

commands and data between all processor elements on the CBE processor and the on-chip controllers for memory and I/O. The EIB supports full memory-coherent and symmetric multiprocessor (SMP) operations [3].

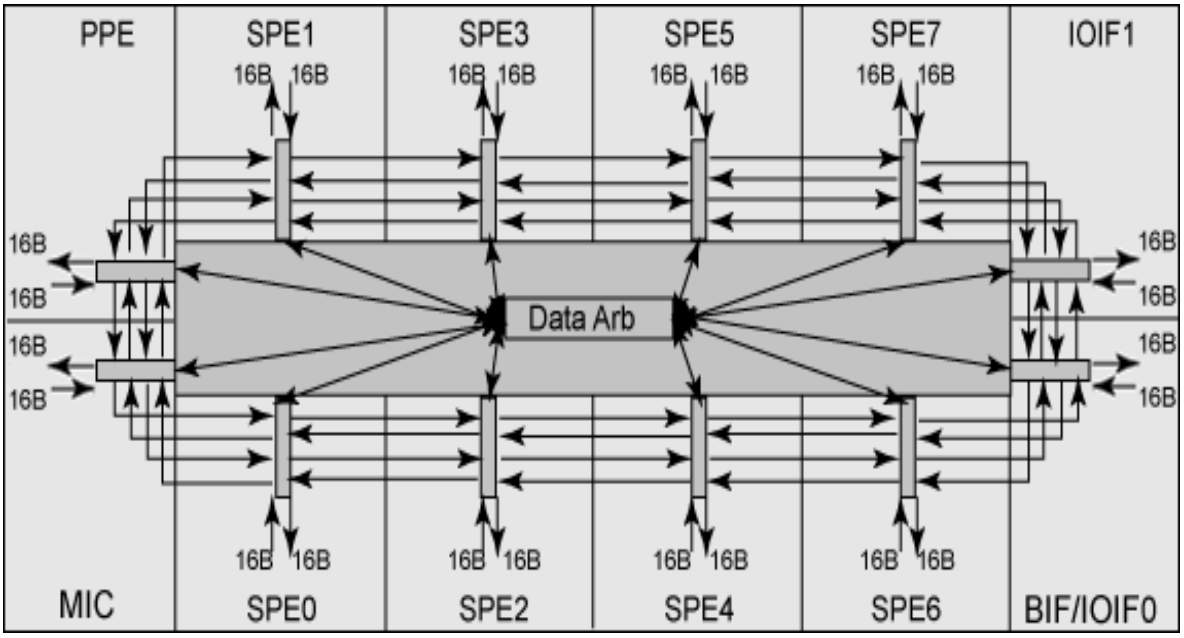


Figure 2-10. Element Interconnect Bus

EIB works on half the processor clock rate. The EIB is a 4-ring structure (two clockwise and two counterclockwise). Each ring supports 3 transfers simultaneously. The EIB's internal bandwidth is 96 bytes per cycle (4 rings * 3 simultaneous transfers/ring * 16 bytes/ring /2). A Resource Allocation Management (RAM) facility resides in the EIB and privileged software can use it to regulate the rate at which resource requestors (the PPE, SPEs, and I/O devices) can use memory and I/O resources.

### 2.6 Memory Interface Controller (MIC)

The on-chip Memory Interface Controller (MIC) provides the interface between the EIB and physical memory. It supports one or two Rambus Extreme Data Rate (XDR) memory interfaces. Memory accesses on each interface are 1 to 8, 16, 32, 64, or 128 bytes. Up to 64 reads and 64 writes can be queued.

### 2.7 Cell Broadband Engine Interface (BEI)

The 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.

The on-chip Cell Broadband Engine Interface (BEI) Unit supports I/O interfacing. It includes a Broadband Interface Controller (BIC), I/O Controller (IOC), and Internal Interrupt Controller (IIC). It manages data transfers between the EIB and I/O devices and provides I/O address translation and command processing. The BEI supports two Rambus FlexIO interfaces.

### 2.8 CELL Architecture Advantages

- **Faster Processing of Compute-Intensive Algorithms:** With the combination of single-precision 32-bit floating-point and 16-bit integer (or fixed-point) processing, the Cell BE processor handles compute-intensive algorithms and is particularly impressive for those requiring many floating-point calculations.
- **Higher Bandwidth Interconnect:** Each SPE contains a 256 KB high-speed local store and a DMA (direct memory access) engine for moving data and code to and from XDR memory and even to other SPEs – all via the EIB interconnect. The DMA engine can simultaneously read and write at the rate of 24 GB/s to and from the EIB and can handle numerous outstanding DMA requests.
- **Backward Compatibility with PowerPC Applications:** Because the Cell architecture is compatible with PowerPC processors, existing PowerPC applications can run on the Cell BE processor without modification. This flexibility provides a convenient entry point for programmers with symmetric multiprocessor (SMP) experience and eases the porting of existing software, including the operating system, to the Cell architecture.

- **Low Voltage/Low Power with High Performance/High Frequency:** The small number of gates per cycle enables the Cell BE processor to operate at low voltage and low power while maintaining high performance and high frequency. By using the SIMD architecture for both the vector media extensions (VMX) on the PPE and the instruction set of the SPEs, both performance and power efficiency are improved.
- **Efficient Communication and Ease of Programming:** The Cell BE processor's high-bandwidth memory and on-chip, coherent, high-bandwidth EIB deliver higher performance on memory bandwidth-intensive applications by enabling high-bandwidth internal interactions among the SPEs and the PPE. This coherency allows the SPEs and the PPE to share a single address space for efficient communication and ease of programming.

## 2.9 Systems based on CELL Architecture

There are two systems available based on CELL architecture. One is Sony PLAYSTATION 3 ant other is IBM BladeCenter QS20. Their details are given below.

### 2.9.1 IBM BladeCenter QS20

It is the first Cell Broadband Engine based system [6]. It is a high performance blade especially suitable for some compute intensive, single-precision, floating-point workloads. It helps to accelerate these targeted workloads to many times the speed of a traditional microprocessor, including image processing, signal processing, and graphics rendering applications.
Its specifications are as follows:

- **Processor:** Two 3.2 GHz Cell BE Processors
- **L2 Cache:** 512KB per Cell BE Processor, plus 256KB of local store memory for each SPE
- **Memory:** 1GB (512MB per processor)
- **Disk Storage:** 40GB IDE HDD

- **Networking:** Dual Gigabit Ethernet
- **Optional connectivity:** 1 or 2 InfiniBand 4x adapters connected via PCI-Express
- **Operating System:** Fedora core 5 Linux

### 2.9.2 Sony PLAYSTATION 3

PLAYSTATION 3, also called PS3, is another system based on the CELL architecture. It's basically a very high resolution gaming console. It provides direct support for installing and booting foreign operating systems. Of course, many of the game-related features such as video acceleration are locked out for the third-party operating systems, but this series focuses on more general-purpose and scientific applications anyway. Note the PS3 has one of the SPE disabled, and one SPE reserved for system use, leaving seven processing units at your disposal [7].

Terra Soft Solutions has developed Yellow Dog Linux 5 in cooperation with Sony specifically for the PS3. Yellow Dog Linux (also known as YDL) has been an exclusively PowerPC-based distribution since its inception, so it was not surprising that Sony contracted it to develop the next version of YDL specifically for the PS3. YDL 5.0 includes libspe [Appendix A] support so that one can utilize power of all the SPEs available within PS3. We can get free version of YDL (YDL 5.0) from Terra Soft Solutions website which can be installed on PS3. See below for instructions on installing the YDL 5 onto the PS3.

#### Preparing PS3 for Installation

- Start your PS3 and perform initial settings if you are using it first time.
- Go to Settings, then System Settings, and choose Format Utility.
- Select Format Hard Disk, and confirm your selection twice.
- Select that you want a Custom partitioning scheme.
- Select that you want to Allot 10GB to the Other OS. This will automatically reserve the remaining disk space for the PS3's game

operating system. When finished, it will restart the system.

- Once the PS3 restarts, it's ready to have Linux installed on it.

**YDL Installation Steps**

- Download the YDL bootloader from Terra Soft and save it as otheros.bld. This bootloader will be installed by gameOS preinstalled on PS3.
- On your flash drive create a directory called PS3. Immediately under the PS3 directory, create another directory called otheros and copy otheros.bld in this directory.
- Insert the flash drive into the PS3.
- Go to Settings, then System Settings, and then choose Install Other OS.
- Confirm the location of the installer, and follow the screens for the installation process. Note that this only installs the bootloader, not the operating system.
- When the installer finishes, go to Settings, then System Settings, and select Default System. Then choose Other OS and press the X button.
- Insert the YDL 5 DVD.
- Plug in your USB keyboard and mouse.
- Now restart the system. You can either do this by holding down the PS button on the controller and then choosing Turn off the system, or by simply holding the power button down for five seconds. Then turn the system back on.
- When it boots back up, you will get a kboot prompt.
- At this prompt type install if your output is going through the HDMI port, or installtext if you are going analog. After this YDL installation will start in a low resolution graphics mode.
- After completion of installation when you reboot PS3, you need to type in ydl480i at the kboot: boot prompt if you are using analog output. Otherwise it will likely change the output to a resolution that the analog output isn't capable of.

# 3 CELL SDK

IBM provides the Software Development Kit for programming and simulating the CELL Architecture. The Software Development Kit 2.1 (SDK) for the Cell Broadband Engine (Cell B.E.) is a complete package of tools to enable you to program applications on the Cell B.E. Processor. The SDK includes both PPU and SPU compilers for all the supported platforms [8]. A Cell B.E. application can run natively on a BladeCenter QS20, PlayStation 3 or in the IBM Full-System Simulator (simulator), which is supported on all of the host platforms. The SDK is composed of following items:

- SDK 2.1 Installation Guide
- Programmer's Guide
- Cell Broadband Engine Programming Tutorial
- XL C/C++ compiler
- IBM Full-System Simulator
- SIMD math AND MASS libraries
- Sample libraries and files

## 3.1 Prerequisites

This section shows some prerequisites for installation of CELL SDK on your host system. The Cell/B.E. SDK runs in Fedora Core 6, which must be installed before you install the SDK. Table 3.1 shows the recommended system configuration for CELL SDK installation.

Table 3.1 Recommended system configuration for CELL SDK installation

| **System** | **Recommended minimum configuration** |
|---|---|
| x86 or x86-64 | 2GHz Pentium® 4 processor |
| PowerPC | 64-bit PPC with a clock speed of 1.42 GHz. 32-bit PPC platforms are not supported. |

All systems should have:

- 5 GB Hard disk space to install the source package and the accompanying development tools
- 1 GB RAM on the host system

### 3.2 Installing the SDK

This section shows how to install CELL SDK on your host system. The cellsdk shell script handles the SDK installation. The SDK's tools are primarily installed in /opt/ibm, although some of the toolchain commands are installed in the regular path. for example, the spu-gcc command is installed in /usr/bin on a PPC machine and in /opt/cell on an x86 machine. The simulator is installed in /opt/ibm/systemsim-cell, and the SDK in /opt/ibm/cell-sdk/prototype/. The script checks to see what features might be needed and what features are available. If some of the other installation files (from the BSC Web site) are needed, they are automatically downloaded to the directory /tmp/cellsdk-1.1.

Do the following to install the SDK :

1. As root download the SDK ISO disk image, CellSDK21.iso from the Cell/B.E. SDK alphaWorks Web site:
   http://www.alphaworks.ibm.com/tech/cellsw
2. Create a mount directory and make sure nothing else is mounted on this directory:
   mkdir –p /mnt/cellsdk
3. Mount the disk image on the mount directory:
   mount –o loop CellSDK21.iso /mnt/cellsdk
4. Change directory to /mnt/cellsdk/software:
   cd /mnt/cellsdk/software
5. Install the SDK by using the following command and answer any prompts:
   ./cellsdk install

6. Optionally build the samples and libraries and copy into the sysroot image for the simulator:
   cd /opt/ibm/cell-sdk/prototype/src
   ./cellsdk build
7. Change directory to any directory which is not the mount directory or below it:
   cd /
8. Unmount the disk image: umount /mnt/cellsdk

## 3.3 IBM's Full-System Simulator- SystemSim

### 3.3.1 Simulator Overview

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. It supports functional simulation of complete systems based on the Cell Broadband Engine processor, including simulation of the PPE, SPUs, MFCs, memory, disk, network, and system console [9]. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development and analysis. The simulator also includes support for performance simulation (or timing simulation) of certain components to allow users to analyze performance of Cell Broadband Engine applications. 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. Figure 1-1 shows the simulation stack. The simulator is part of the software development kit (SDK), which is available through IBM alphaWorks Emerging Technologies at http://www.alphaworks.ibm.com/tech/cellsystemsim.

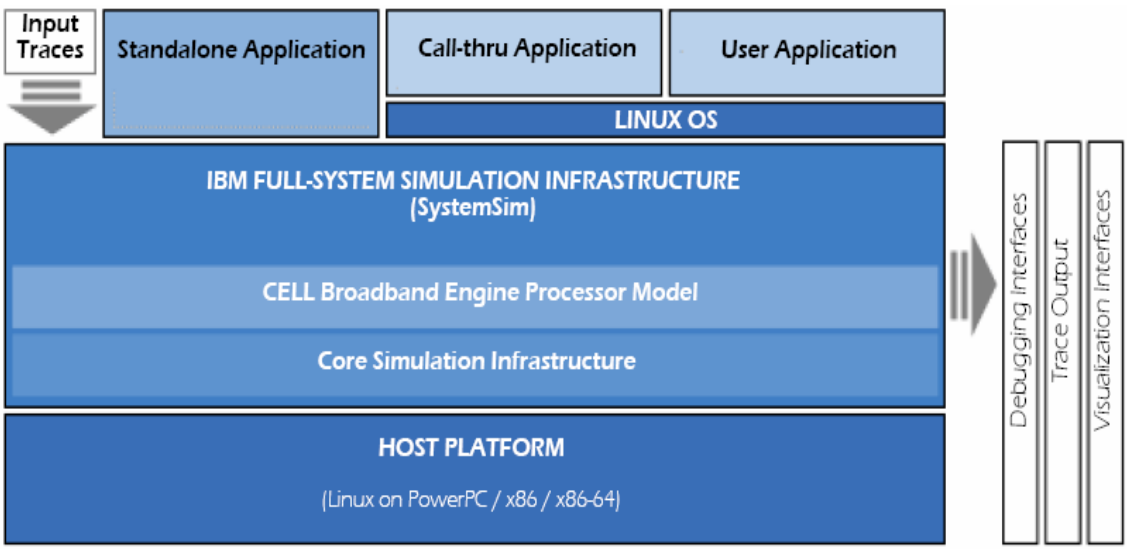


Figure 3.1 Simulator Stack for the Cell Broadband Engine

If accurate timing information and performance statistics are not required, the simulator can be used in its *functional only* mode, simulating the architectural behavior of the system to test the functions and features of a program. For performance analysis, the simulator can be used in *performance simulation* mode. The simulator 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.

### 3.3.2 Invoking the Simulator

The simulator is invoked using the systemsim command. This command is in the bin directory of the systemsim-cell release, which should be added to the user's PATH before invoking systemsim.

When the simulator starts, it loads an initial run script which typically configures and initializes the simulated machine. The name of the initial run script can be passed to systemsim with the -f option. When not specified on the command line, the simulator will look in the current directory for the file .systemsim.tcl, and if present, will use this file as the initial run script.

Otherwise, it will use the file systemsim.tcl in the lib/cell directory of the systemsim-cell release. When specified using the -f option, the name of the initial run script can contain an absolute or relative path. The simulator searches for Initial run scripts with a relative path by first looking in the current directory, and then in the lib/cell directory of the systemsim-cell release, and finally in the lib directory of the systemsim-cell release. If the simulator fails to find the initial run script specified with the -f option, it issues an error message and exits.

It is generally the task of the initial run script to locate the operating system and filesystem images to be used by simulated machine. For the Cell simulator, the default initial run script searches for a Linux kernel image named vmlinux and a filesystem image named sysroot_disk. The script will look first in the current directory and then in the systemsim-cell/images/cell directory, and uses the first instance it finds for these images. If the script fails to find either of these images in one of these locations, it will print an error message and terminate the simulator.

The following examples illustrate various ways to invoke the simulator. These examples assume that the simulator was installed into /opt/ibm/systemsim-cell.

(1). To run the simulator without the GUI, issue:
*PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim*
If the user has created a run script named .systemsim.tcl in the current directory, the simulator will use this as the initial run script. Otherwise, the simulator uses systemsim.tcl in the lib/cell directory of the systemsim-cell release as the initial run script.

(2). To start the simulator with the GUI window enabled, specify the "-g" option on the command line when invoking systemsim. For example, to run

the simulator with the GUI using either the user's .systemsim.tcl or the simulator's lib/cell/systemsim.tcl as the initial run script, issue:
*PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -g*

(3). To run the simulator without the gui, without a console window (-n), in quiet mode (-q), using the initial run script myrun.tcl, issue:
*PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -n -q -f myrun.tcl*
When the simulator starts, the window in which it was started becomes the simulator command window where you can enter simulator commands. The simulator also creates the console window (unless this was disabled with -n) which is initially labeled UART0 in the window's title bar, and a GUI window if this was requested with the -g option.

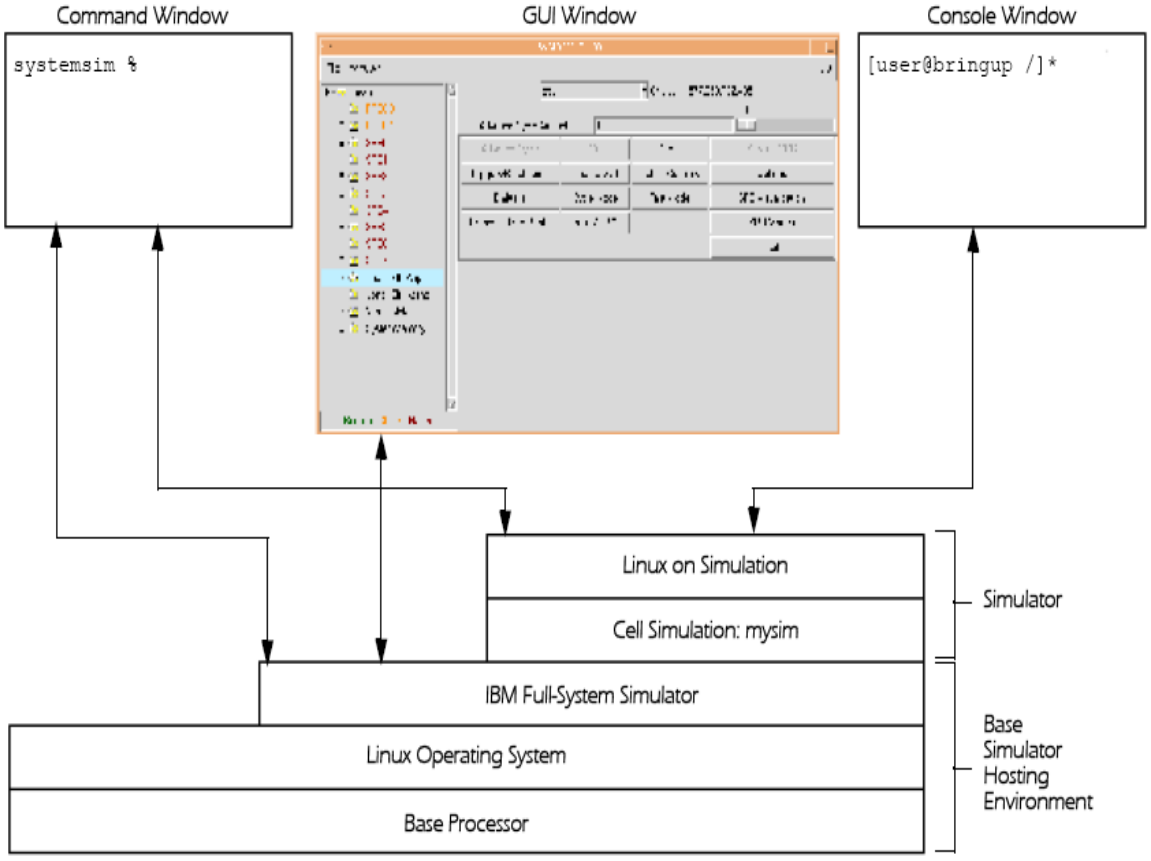


Figure 3.2 Simulator Structure and Screens

### 3.3.3 Operating-System Modes

A key attribute of the IBM Full-System Simulator is its ability to boot and run a complete PowerPC system. By booting an operating system, such as Linux, the IBM Full-System Simulator can execute many typical application programs that utilize standard operating system functionality. Alternatively, applications can be run in standalone mode, in which all operating system functions are supplied by the simulator and normal operating system effects do not occur, such as paging and scheduling. The IBM Full-System Simulator can also execute SPU programs in standalone mode on a given SPU. These two approaches to running applications on the simulator are referred to as Linux mode and standalone mode [9].

- **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 nonsimulation (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.

However, there are limitations that apply when building an application to be loaded and run by the simulator without an operating system. For example, applications should be linked statically with any libraries they require since the standard operating system shared libraries are not available in standalone mode. Another example is support for virtual memory address

translation. Typically, the operating system provides address-translation support. Since an operating system is not present in standalone 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’.

### 3.3.4 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.

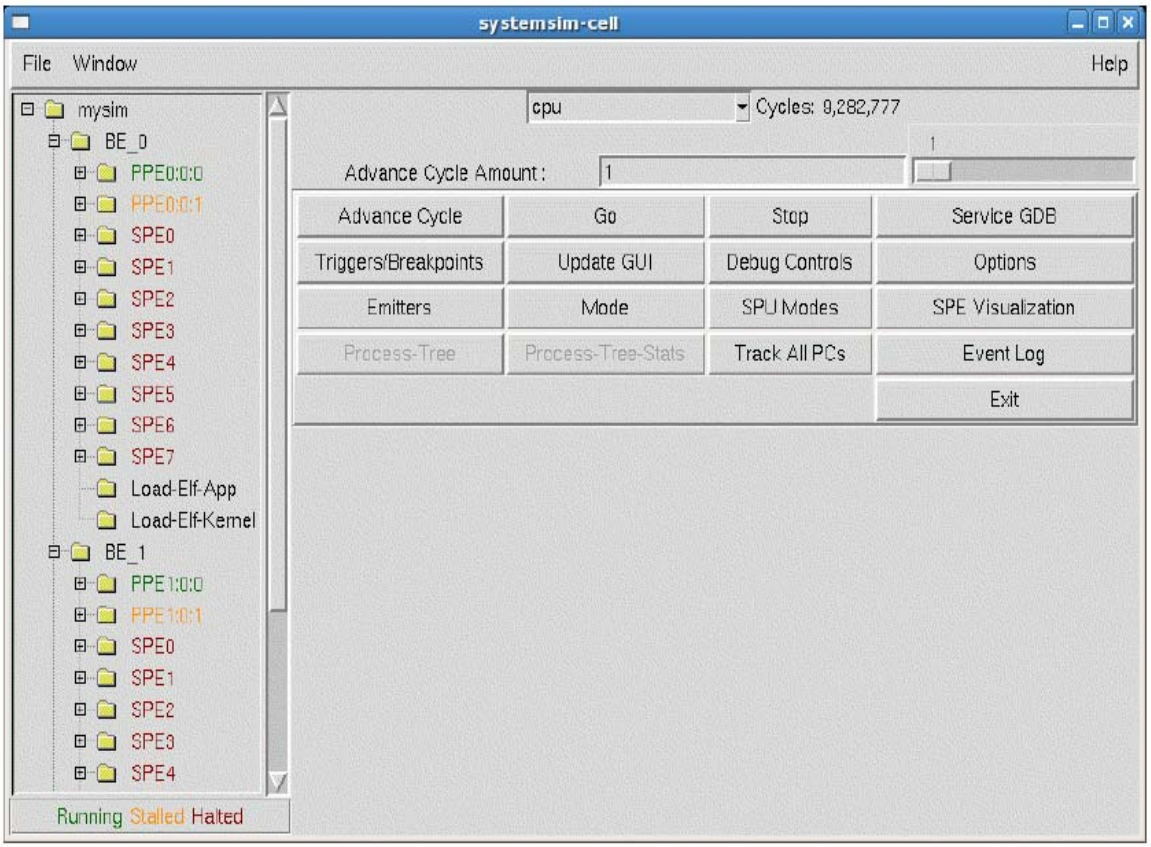


Figure 3.3 Graphical User Interface for the Simulator

The main GUI window has two basic areas: the vertical panel on the left, and 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. When the simulator is started it creates a simulated machine containing a Cell Broadband Engine processor and displays the main GUI window, labeled with the name of the simulator program. When the GUI window first appears, click the Go button to boot the Linux operating system. If the simulator is launched in SMP, or dual Cell-based system, the vertical panel in the main window displays each BE with its components, as shown in Figure 3.4

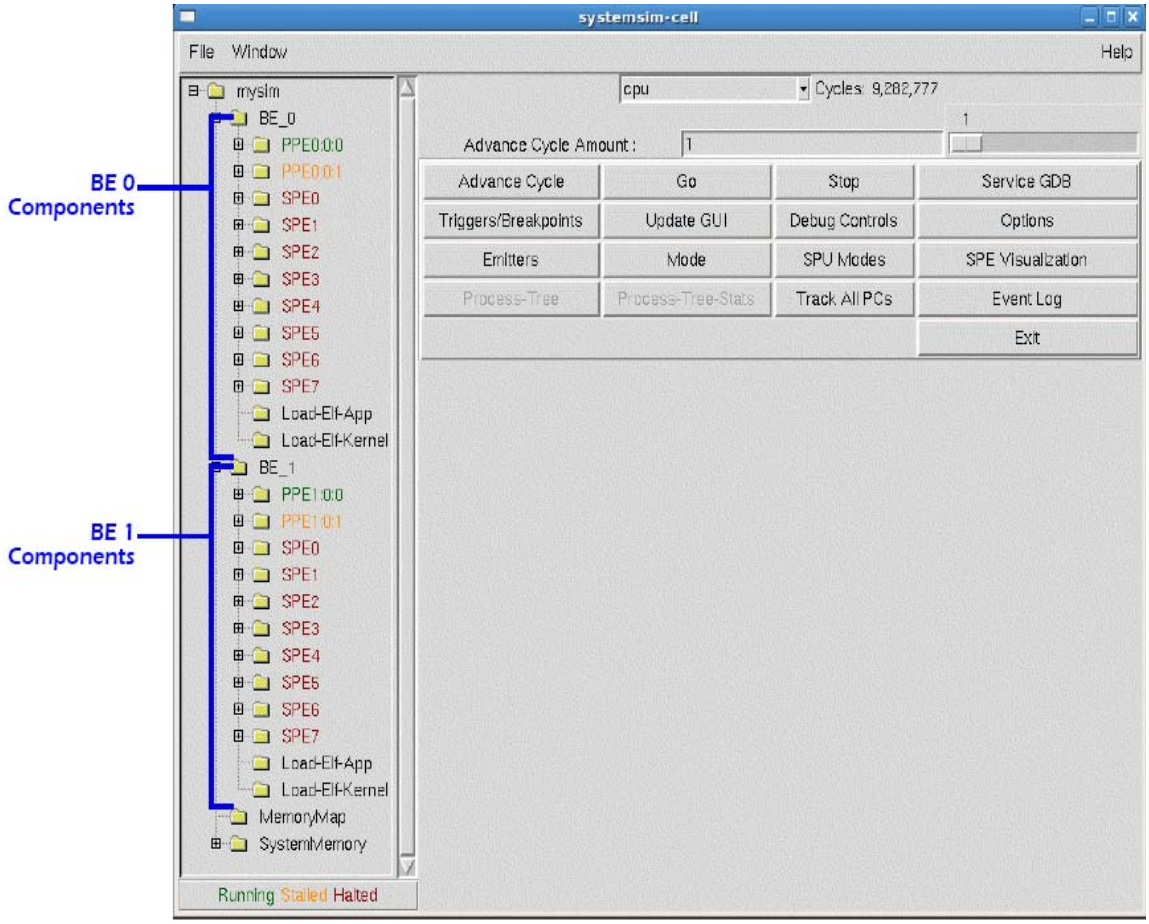


Figure 3.4 Simulator GUI started in SMP mode

### 3.3.5 Accessing the Host Environment: The Callthru Utility

The callthru utility allows you to copy files between the host system and the simulated system while it is running. This utility runs within the simulated system and accesses files in the host system using special callthru functions of the simulator. The source code for this utility is provided with the simulator in the sample/callthru directory as a sample of the use of the simulator callthru functions. In the Cell SDK, the callthru utility is installed as a binary application in the simulator system root image in the /usr/bin directory. The callthru utility supports the following options:

→ To write standard input into <filename> on the host system, issue
*callthru sink <filename>*

→ To write the contents of <filename> on the host system to standard output, issue
*callthru source <filename>*

Redirecting appropriately lets you copy files between the host and simulated system. For example, to copy the /tmp/matrix_mul application from the host into the simulated system and then run it, issue the following commands in the console window of the simulated system:
*callthru source /tmp/matrix_mul > matrix_mul*
*chmod +x matrix_mul*
*./matrix_mul*

Another commonly used feature of the callthru utility is the exit option, which will stop the simulation, similar to the stop button of the GUI, but initiated by the callthru utility inside the simulator rather than through user interaction. This is especially useful for constructing “scripted” executions of the simulator that involve alternating steps in the simulator and the simulated system.

→To stop the simulator and return control back to currently active run script or the GUI / command line, issue
*callthru sink <filename>*

### 3.3.6 Simulator Support for Performance Analysis

The simulator provides several modes of functional-only and performance simulation. In most cases, the simulation mode can be changed dynamically at any point in the simulation. However, certain “warm-up” effects may affect the results of performance simulation for some portion of the simulation following a change to cycle mode.

- Simple (functional-only) mode models the effects of instructions, without attempting to accurately model the time required to execute the instructions. In simple 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.

- Fast mode is similar to functional-only mode in that it fully models the effects of instructions while making no attempt to accurately model execution time. In addition, fast mode bypasses many of the standard analysis features provided in functional-only mode, such as statistics collection, triggers, and emitter record generation. Fast mode simulation is intended to be used to quickly advance the simulation through uninteresting portions of program execution to a point where detailed analysis is to be performed.

- Cycle (performance) 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 mode allows you to:

- Gather and compare performance statistics on individual components (such as the SPU) or full systems.
- Characterize the system workload.
- Forecast performance at future loads, and fine-tune performance benchmarks for future validation.

In the cycle 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

The performance models described above can be enabled from the GUI using the performance models dialog or with simulator commands. The performance models can be enabled at any time after the simulated machine has been defined, but typically are not enabled until after the operating system has been booted. To enable the performance models in a simulation, complete the following steps:

To enable the performance models from the graphical user interface:

1. Click the Perf Models button on the main GUI window.

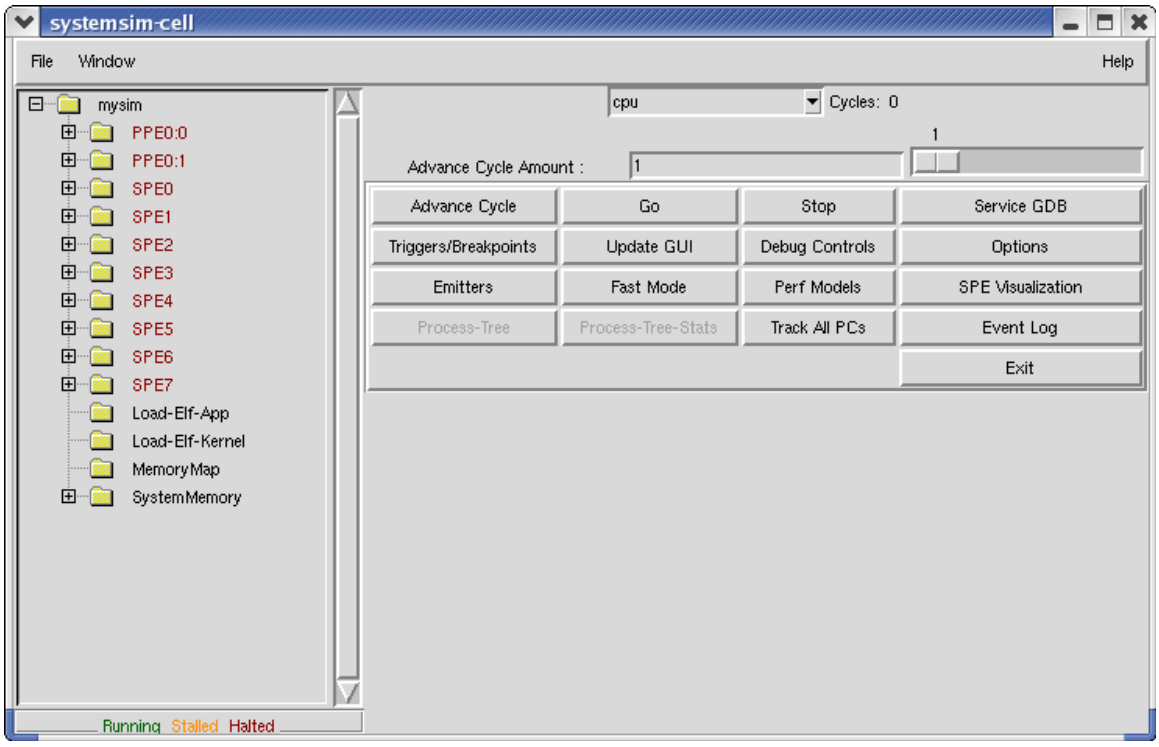


Figure 3.5 SystemSim Cell Graphical User Interface

2. SystemSim displays the following window that provides checkboxes to enable the performance model for SPEs.

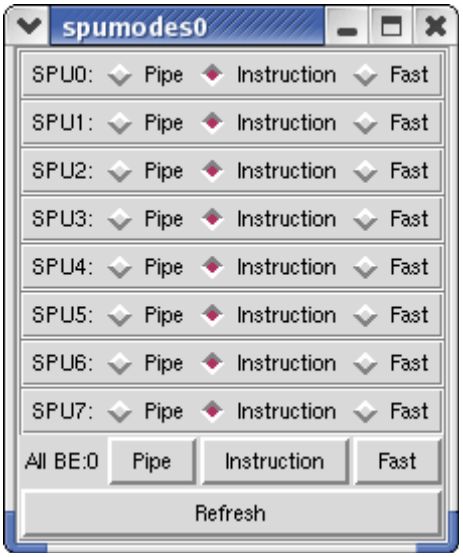


Figure 3.6 SystemSim Spu Performance Modes

a. For each individual SPU, click the level of timing mode to simulate: Pipe, Instruction, or Fast.

b. To enable the same timing mode for all SPUs in the BE, click the corresponding button.

c. Click Refresh to synchronize the window with any changes to the modeling mode that may have been updated by the command line interface or the tree view.

### 3.3.7 Performance Profile Checkpoints

The simulator can collect performance statistics for each SPU running in pipeline mode that are useful in determining the sources of performance egradation, such as channel stalls and instruction-scheduling problems. You can also use performance profile checkpoints to delimit a specific region of code over which performance statistics are to be gathered. Performance profile checkpoints 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. he application program interface (API) for the performance profile checkpoints is defined in the profile.h header file. This file provides the C-language procedures, named prof_cp{n}() where n is a numeric value ranging from 0 to 31, that generate the special no-op 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 examples illustrate 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, the simulator prints data identifying the calling SPE and the associated timing event. The data is displayed on the simulator control window in the following format:

SPU*n*: CP*m*, *xxxxx*(*yyyyy*), *zzzzzzz*

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, and zzzzzz is the cycle counter.

# 4 PROGRAMMING THE CELL

## 4.1 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. C language extension for both PPE and SPE instruction set and their compilers are provided with SDK.

In order to maximize the performance of the Cell, the following two points need to be paid attention to [5]:

- Operate multiple SPEs in parallel to maximize operations that can be executed in certain time unit.
- Perform SIMD parallelization on each SPE to maximize operations can be executed per instruction.

## 4.2 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 [5]:

- 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

The main model for partitioning an application is PPE-centric, as shown in Figure 4-1.

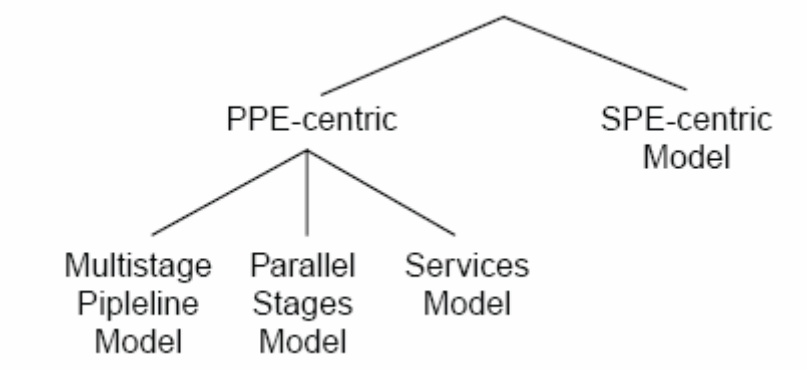


Figure 4-1 Application Partitioning Model

In the PPE-centric model, the main application runs on the PPE, and individual tasks are offloaded 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.

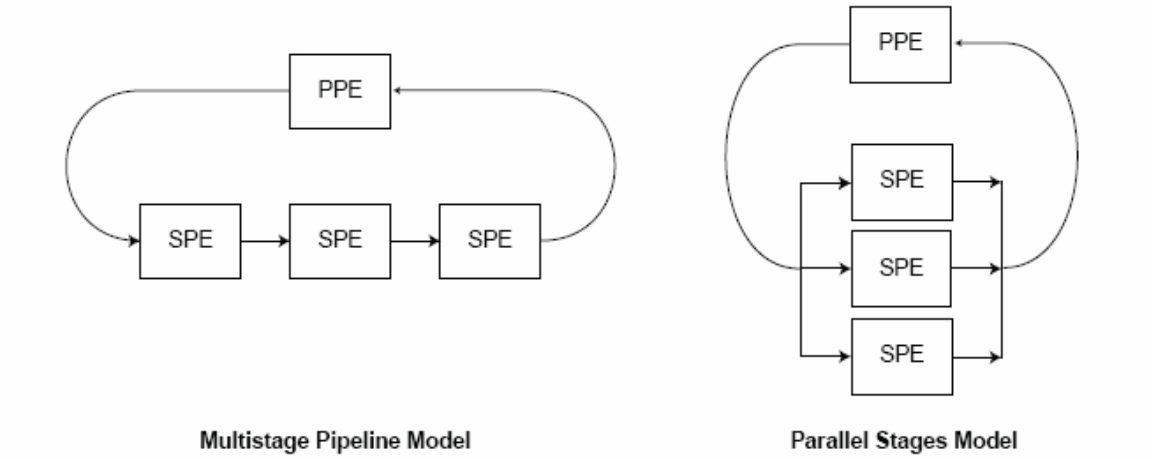


Figure 4-2 PPE-Centric Multistage Pipeline Model and Parallel Stages Model

There are three ways in which the SPEs can be used in the PPE-centric model (1) Multistage Pipeline Model (2) the Parallel Stages Model and (3) the Services Model [5]. The first two of these are shown in Figure 4-2.

If a task requires sequential stages, the SPEs can act as a multistage pipeline. The left side of Figure 4-2 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 Model increases the data-movement requirement because data must be moved for each stage of the pipeline.

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 4-2.

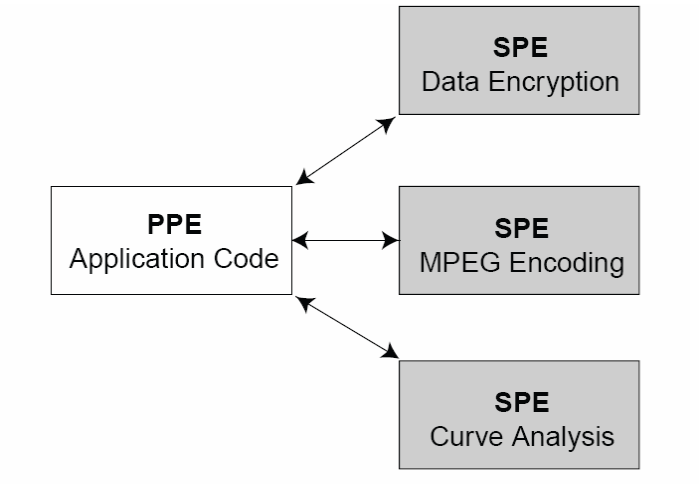


Figure 4-3 PPE-Centric Services Model

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 4-3 shows the PPE-centric Services Model.

## 4.3 Data Partitioning

With Cell programming, how the work is partitioned among available SPEs is an important consideration. Although there are many ways of doing this, the following explanation is based on a model that subdivides data to enable concurrent processing. The data-partitioning application model parallels the same program across multiple SPEs. Data is partitioned by the PPE program and uniformly distributed to SPEs. Figure. 4.4 provide an image of this approach.

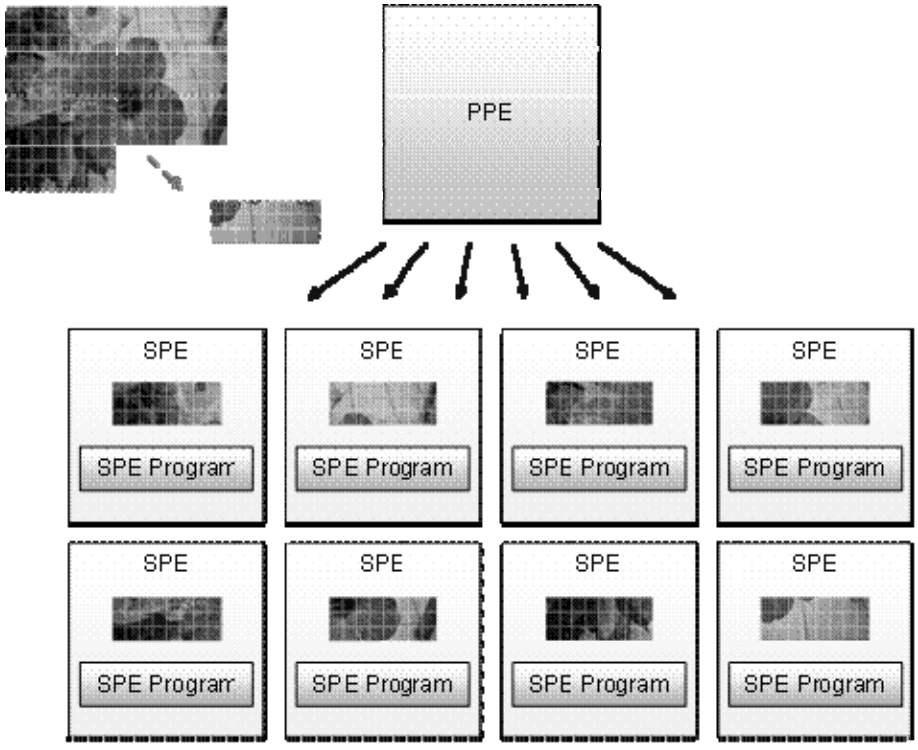


Figure 4-4 Data Partitioning for Parallel Processing

### 4.4 Program Control and Data Flow

In line with the Cell architecture, let's take a look at how PPE and SPE programs are executed, together with how necessary data is transmitted and received [5].

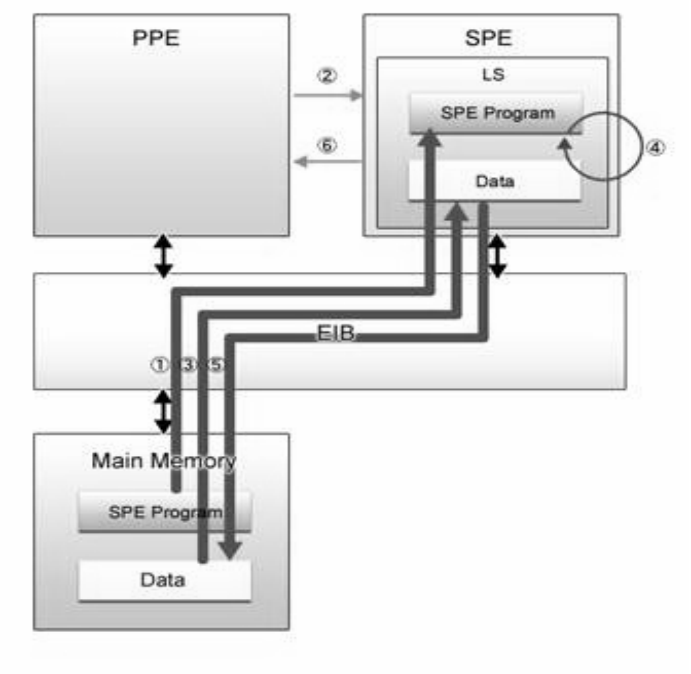


Figure 4-5 SPE program execution sequence

1. PPE program loads the SPE program to the LS.
2. PPE program instructs the SPEs to execute the SPE program.
3. SPE program transfers required data from the main memory to the LS.
4. SPE program processes the received data.
5. SPE program transfers the processed result from the LS to the main memory.
6. SPE program notifies the PPE program of the termination of processing.

### 4.5 Advanced Cell Programming

This section shows advanced Cell programming techniques which can be used

to improve performance further [13].

### 4.5.1 Communication between PPE and SPE

The described earlier SPE makes use of the MFC to transfer data between itself and another SPE or the PPE. The MFC incorporates a DMA controller to allow DMA transfer.

Now let's take a look into some other means of communication offered by the MFC to support the transactions between an SPE and its external world (other SPEs and the PPE). Typical among them are the mailboxes and signal notification registers.

#### 4.5.1.1 Mailboxes

While DMA transfer allows transfer of up to 16K bytes of data between the main memory and each SPE's LS, mailboxes are designed for transfer of 32-bit data between the PPE and the SPE.

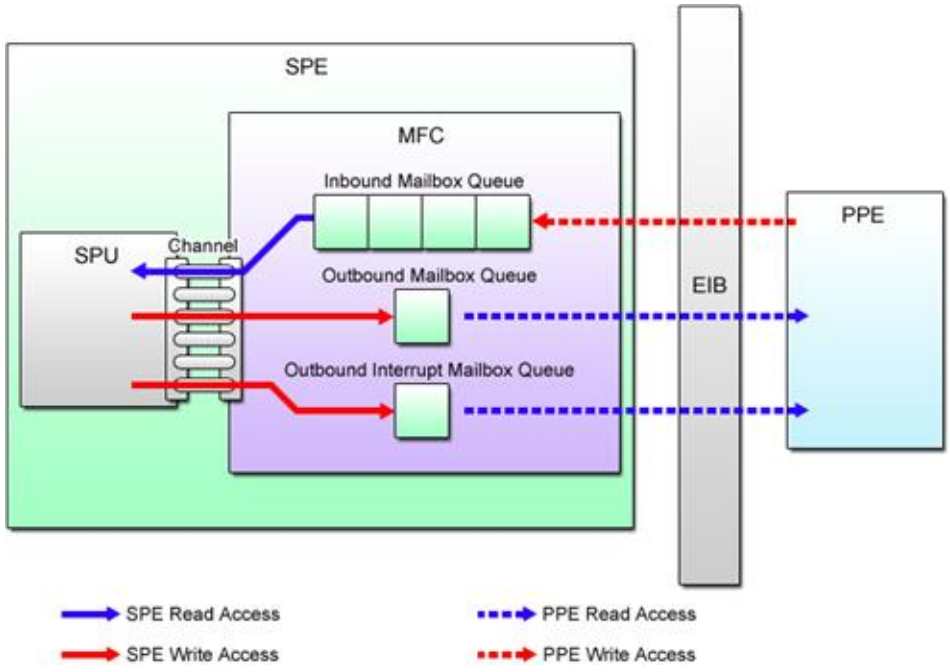


Figure 4-6 Mailboxes

To put it another way, mailboxes are fit for transferring small data such as

status information and parameters. Structurally, mailboxes are FIFO queues. The MFC provides three types of mailbox queues, each with a different behavior and data transfer direction as shown in Fig. 4.6.

- **SPU Inbound Mailbox:** Used to send data from the PPE to the SPE. This mailbox has space for storing up to four 32-bit messages at a time. If no message is found when the SPE program accessed the queue, the SPE stalls until data is written by the PPE program.
- **SPU Outbound Mailbox:** Used to pass data from the SPE to the PPE. This mailbox has the capacity to accept only one 32-bit message. If the SPU outbound mailbox is full, writing of the next data is suspended until the PPE reads the data from the queue.
- **SPU Outbound Interrupt Mailbox:** Like the SPU outbound mailbox, this is used to send data from the SPE to the PPE. When this mailbox is written, however, an interrupt event is generated to notify the PPE when to read the data.

These mailboxes can be accessed either from the SPE or PPE programs. The SPE program uses a channel interface to access the mailboxes. The PPE program does this via mailbox API functions offered by libspe2.

#### 4.5.1.2 Signal Notification Registers

SPU signal notification registers (SNRs) are 32-bit registers used to send signals, such as control messages and events, to an SPE from other SPEs or the PPE. There are two SNRs for each SPE.

The sending processor (either PPE or SPE) writes the signal value in the form of 32-bit data into the SNR of the receiving processor (one of other SPEs). When the value is read by the receiving processor, all bits in the SNR are reset to zero. If the SNR is empty when it is read, the receiving processor stops execution until the signal is written.

Each bit of the 32-bit signal data can be assigned to the program's own specific meaning. By doing so, the signal can notify the SPE of anything from

the change of status to the completion of processing. The SNRs can also be configured for overwrite mode or logical OR mode. The overwrite mode is useful in a one-to-one signaling environment, whereas the logical OR mode enables many-to-one signaling. Either of these modes can be selected for each SNR, independently of the other.

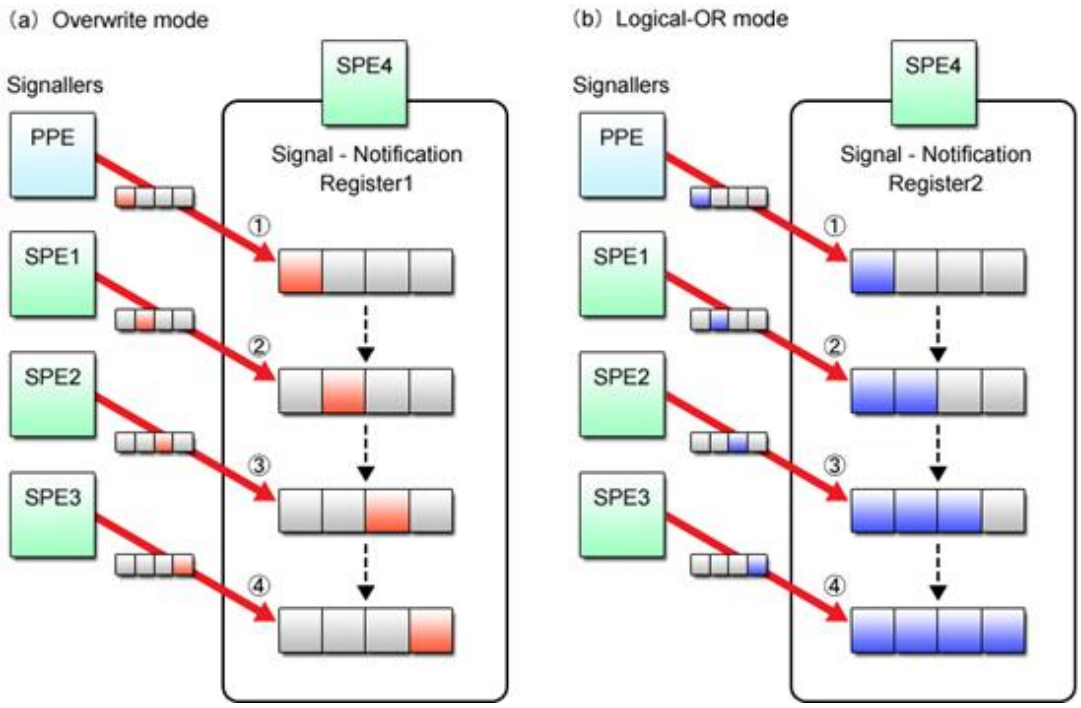


Figure 4-7 SPU Signal Notification Modes

SNRs are similar to the SPU inbound mailbox. A major difference between the two is that SNRs can be used not only for data transfer between the PPE and SPE but also for that between SPEs, whereas the mailbox is limited to use between the PPE and SPE only. Capitalizing on the logical OR mode that allows many-to-one data transfer, the SNRs also make barrier synchronization of multiple SPE programs possible.

### 4.5.2 Effective Utilization of DMA Transfer

This section gives a closer look into the precautions we have to bear in mind when programming DMA transfer for the Cell and learn about the techniques for getting the most from this data transfer system.

#### 4.5.2.1 Transferable Data Size and Address Alignment

It is previously explained that DMA transfer of data is a multiple of 16 bytes. However, the Cell can also handle DMA transfer smaller than this data size. Any data that meets the following size and alignment requirements is pursuant to the Cell's DMA transfer constraints.

- **DMA Transfer Size:** The data size must basically be specified in a multiple of 16 bytes (16, 32, 48, etc.). For DMA transfer of data smaller than 16 bytes, the Cell also supports data size of 1, 2, 4 or 8 bytes. The maximum data size that can be transferred at one time is 16K bytes.
- **Address Alignment in DMA Transfer of More than or Equal to 16 Bytes:** When the data to be transferred is greater than or equal to 16 bytes, the addresses on both sending and receiving sides (i.e., effective address and LS address) must be aligned on 16-byte boundaries. The execution performance of DMA transfer is maximized when the addresses of the data area on both sides begin on a 128-byte boundary.
- **Address Alignment in DMA Transfer of Less than 16 Bytes:** When the data to be transferred is smaller than 16 bytes, the addresses of the data area on both sending and receiving sides must satisfy the following conditions.

  → Address alignment is consistent with the transfer size, i.e., addresses are aligned on the byte boundary equal to the transfer size.

  → The lower 4 bits (indicating the relative position from the 16-byte boundary) of the effective address and the LS address are the same.

  → Taking DMA transfer of 4-byte data as an example, Fig. 4.3 graphically illustrates these requirements. To review again, the effective address and the LS address must not only be aligned on 4-byte boundaries but also begin at the same relative position from the 16-byte boundary.

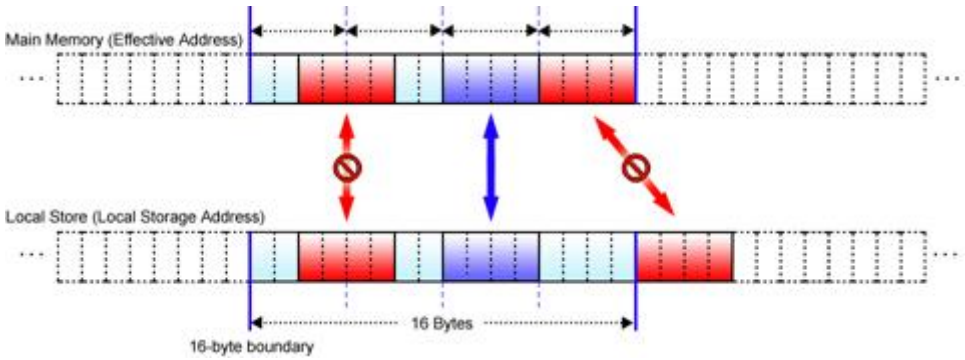


Figure 4-8 DMA Transfer (16 Bytes or Less)

Unless the above-mentioned rules are observed, the MFC will stop, resulting in a DMA alignment error interrupt, which may cause abnormal termination of both the PPE and SPE programs.

#### 4.5.2.2 DMA Double Buffering

For most applications, DMA transfer accounts for a large percentage of the time required for processing by the SPE program, which is to say, improving efficiency of DMA transfer is a significant means of making applications run faster. In this section, therefore, introduces double buffering, a technique for improving the efficiency of DMA transfer, while maximizing SPE performance.

Let's start by quickly reviewing the efficiency of the DMA transfer. With the basic programs, DMA transfer is performed in the following sequence.

(1) Initiates DMA transfer (GET) to the input buffer.
(2) Waits for Step (1) to complete.
(3) Processes data and stores the calculated result in the output buffer.
(4) Initiates DMA transfer (PUT) from the output buffer.
(5) Waits for Step (4) to complete.
(6) Repeats Steps (1) through (5).

This sequence can be described as shown in Fig. 4.9 (a) in terms of time series from the viewpoint of input/output buffer operations. It can also be diagrammed as shown in Fig. 4.9 (b) when viewed from the relationship

between MFC and SPU operations.

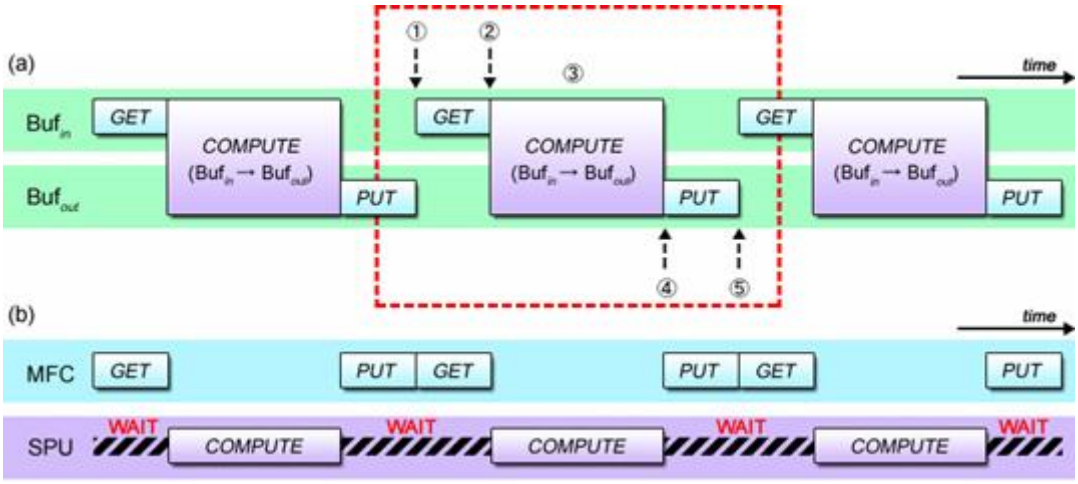


Figure 4-9 Basic Programs Involving DMA Transfer

What's clear from Fig. 4.9 (b) is that the MFC does nothing while the SPU is executing its computational task, or the SPU simply stalls until the DMA transfer by the MFC is complete, making it impossible to fully realize the SPU's performance potentials. That's why a variety of techniques have been developed to achieve more efficient DMA transfer. Double buffering is one of such techniques. The double buffering method uses two separate buffers in parallel for both input and output so that while one of them is used for computation, the other one can be filled or emptied by DMA transfer. In contrast to double buffering, the previously explained DMA transfer using only a single pair of input and output buffer is called single buffering.

Fig. 4.10 (a) illustrates the time series processing flow of double buffered DMA transfer. Fig. 4.10 (b) shows this processing flow reconfigured to highlight the concurrency between the MFC process and SPU process.

As can be seen from Fig. 4.10 (b), the MFC performs DMA transfers in parallel with the SPU's calculations, and the SPU does not spend a time just to wait for the completion of the transfers. Double buffering greatly improves the SPE's computational capability by enabling DMA transfers in parallel with

calculations.

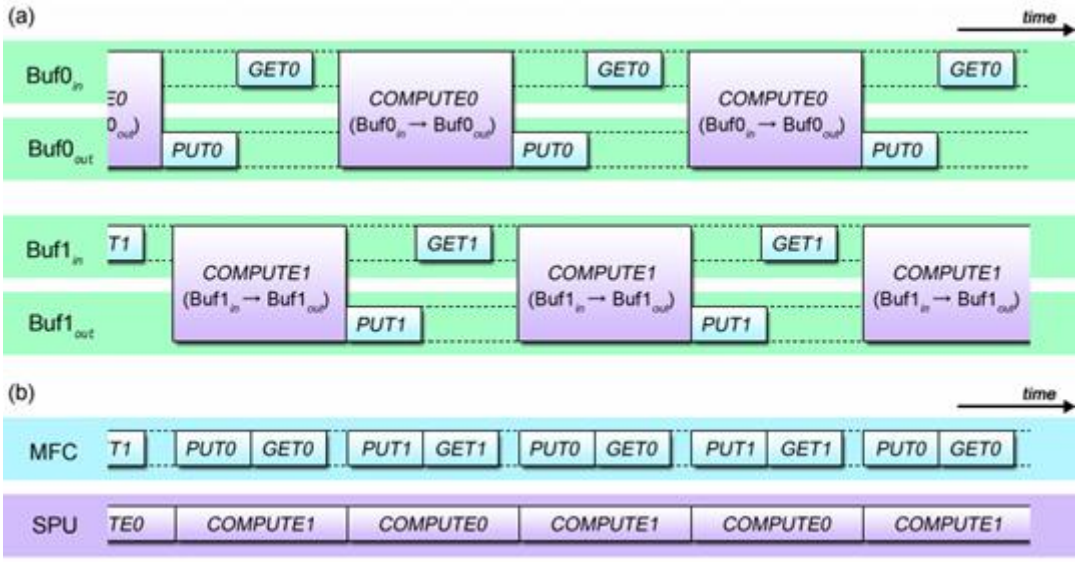


Figure 4-10 DMA Double Buffering Method

# 5 DIP ALGORITHMS AND THEIR IMPLEMENTATION

## 5.1 Spatial Domain Filter

### 5.1.1 General

Filtering of the image can be done in two ways: in frequency domain and in spatial domain. In spatial domain filtering, operation is performed directly on pixels of an image. The process consists of moving the filter mask from point to point in an image. At each point (x, y) the response of the filter at that point is calculated using a predefined relationship. For spatial filters, the response is given by sum of products of the filter coefficients and the corresponding image pixels in the area spanned by the mask. This process is similar to the process of convolution and is referred as “convolving a mask with an image” [10].

We can implement various filters such as smoothing filter, sharpening filter, laplacian filter etc. just by selecting appropriate filter mask.

### 5.1.2 Proposed Algorithm

This section shows the algorithm of spatial domain filter for CELL BE. It divides the data such that the computation power of all processors can be used simultaneously. It also makes use of vectorization in SPEs to achieve better performance.

In this algorithm, the image is first divided into 8 equal parts by PPE. For example if the image is 256 X 256 pixels, then it is divided into 8 equal parts of 256 X 32 size and passed to each SPE. This image part is shown in figure 5-1.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | - - - - - - - - - | 1,256 |
| 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | - - - - - - - - - | 2,256 |
| 3,1 | 3,2 | 3,3 | 3,4 | 3,5 | - - - - - - - - - | 3,256 |
| - | | | | | | |
| - | | | | | | |
| - | | | | | | |
| - | | | | | | |
| - | | | | | | |
| - | | | | | | |
| 32,1 | 32,2 | 32,3 | 32,4 | 32,5 | - - - - - - - - - | 32,256 |

Figure 5-1 Image portion transferred to SPE

Then the convolution of the mask must be done with the entire image. As we know that SPEs support SIMD so we can take advantage of it and perform the convolution of mask with more than one pixel at a time. Following example shows this task:

a) First of all 9 vectors are taken. The pixel values in each vector are shown in following figure. Here (x,y) means value of pixel at (x,y).

| | | | |
|---|---|---|---|
| 1,1 | 1,2 | 1,3 | 1,4 |
| 1,2 | 1,3 | 1,4 | 1,5 |
| 1,3 | 1,4 | 1,5 | 1,6 |
| 2,1 | 2,2 | 2,3 | 2,4 |
| - | | | |
| - | | | |
| - | | | |
| - | | | |
| 3,3 | 3,4 | 3,5 | 3,6 |

Figure 5-2 Vectors containing the pixel value

b) The mask (example mask) to be multiplied is shown in following figure. Let us call it m.

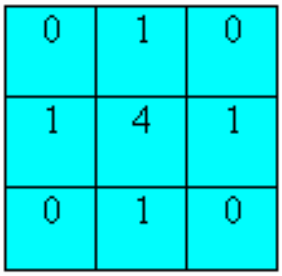


Figure 5-3 Filter mask

Now vector 1 is multiplied by first element of mask. Vector 2 is multiplied by second element of mask and so on.

c) Now all the vectors are added. And result is divided for scaling.
Let us say the image data is I and mask is m. Then the first element of result will be

I(1,1)m(1,1) + I(1,2)m(1,2) + I(1,3)m(1,3) + I(2,1)m(2,1) + I(2,2)m(2,2) + I(2,3)m(2,3) + I(3,1)m(3,1) + I(3,2)m(3,2) + I(3,3)m(3,3)

Which is convolution of pixel at position I(2,2) with the mask. Similarly second element of result is convolution of I(2,3) with the mask and so on.
Thus we get convolution of four pixels with the mask simultaneously.

d) Repeat the steps (a.) to (c.) for next four pixels.
After completion of work by SPE, i.e. application of mask on all the parts of the image by SPE, results are collected by PPE and filtered image is generated.

### 5.1.3 Implementation and Result

Algorithm for spatial domain filter was developed in four different ways: (A) original (scalar) version which can run on any single core system such as

Intel Pentium-4 (B) parallel version without vectorization, (C) parallel version with four element vectors and (D) parallel version with eight element vectors. The algorithm was applied on a 256 X 256 gray scale image and performance was measured. While processing the image using parallel versions, horizontal lines were generated at regular interval due to boundary problems while applying filter to all the parts. This problem was solved by overlapping of two rows.

Original version of the algorithm was executed on SimpleScalar simulator (configured for Pentium 4) [Appendix B], while parallel versions of the program were executed on simulator called Systemsim, which is provided with Cell Software Development Kit by IBM. In all the cases performance was measured in terms of clock cycles. The results are shown in following table:

Table 5-1 Performance results for Spatial Domain Filters

| | | **Compiler** | **Simulator** | **Clock Cycles** |
|---|---|---|---|---|
| **Case A** | **Scalar Version** | gcc | Simplescalar | 2,207,290 |
| **Case B** | **Parallel Version without Vectorization** | gcc | IBM Systemsim | 520,591 |
| **Case C** | **Parallel Version with four element Vectors** | gcc | IBM Systemsim | 350,641 |
| **Case D** | **Parallel Version with eight element Vectors** | gcc | IBM Systemsim | 208,300 |

So from Table 1, we can observe that if we use all the eight SPEs, but without vectorization the speedup achieved is (2,207,290)/ (520,591) = 4.23. Here we are using eight cores but Speedup achieved is not eight. This is because of inter-processor communication and memory stalls because all the eight cores can't access memory simultaneously.

Now let us consider the effect of the vectors. The speedup achieved using 4 element vectors over without vectorization i.e. (Case B/ Case C) is (520,591)/ (350,641) = **1.48**. Thus instead of 4 the speedup achieved is only 1.48. This is because conversion from vector to scalar and scalar to vector incurs lots of overhead. Similarly if we use 8 element vectors the speed achieved is (520,591)/ (208,300) = **2.49** instead of 8.

The maximum performance is achieved using parallel version with eight element vector. In that case the speedup achieved (maximum speedup) is (Case A/ Case D) = (2,207,290)/ (208,300) = **10.59**.

We can calculate theoretical speedup using Amdahl's law [Appendix C]. As we saw Cell provides two level of parallelism: one is multiple cores and other is SIMD within each core. Therefore program can be improved 8X8=64 times (using 8 element vector). In our case the portion of the program that has been parallelized takes 99.8 % of the whole program execution time. Therefore using Amdahl's law

$$Speedup = \frac{1}{(1-0.998)+\frac{0.998}{64}} = 56.85$$

Thus theoretical speedup is around 57 but we are getting the speedup around 11. This difference is because the reasons given above.

## 5.2 JPEG

JPEG is one of the most popular still image compression standards. It defines four different coding processes (1) Baseline process (2) Extended DCT based process (3) Lossless process (4) Hierarchical process [12].

JPEG baseline process includes following steps:

- Colour space transformation
- DCT
- Quantization
- Zig-zag ordering

- Entropy coding

In baseline process the input and output data precision is limited to 8 bits. Following sections explains various stages of baseline process in detail.

### 5.2.1 Colour space transformation

In this step image is transformed into a suitable colour space. This is a no-op for gray scale, but color images are generally transformed into a luminance/chrominance color space (YCbCr, YUV, etc). Following figure shows relation between RGB and YCbCr. The luminance component is gray scale and the other two axes are color information. The reason for doing this is that we can afford to lose a lot more information in the chrominance components than we can in the luminance component: the human eye is not as sensitive to high-frequency chroma info as it is to high-frequency luminance. We don't have to change the color space if we don't want to, since the remainder of the algorithm works on each color component independently, and doesn't care just what the data is. However, compression will be less in this case.

$$y = \frac{8432}{32786}R + \frac{16425}{32768}G + \frac{3176}{32768}B + 16 \quad \text{------ (1)}$$

$$Cb = -\frac{4818}{32786}R - \frac{9627}{32768}G + \frac{14345}{32768}B + 128 \quad \text{------ (2)}$$

$$Cr = \frac{14245}{32786}R - \frac{12045}{32768}G - \frac{2300}{32768}B + 128 \quad \text{------ (3)}$$

### 5.2.2 Discrete Cosine Transform

After color space transformation the image is first subdivided into pixel blocks of size 8X8, which are processed left to right, top to bottom. Then all the 64 pixels of the block are level shifted by subtracting $2^{(n-1)}$, where $2^{(n)}$ is the maximum gray level. The 2-D discrete cosine transform of the block is computed in accordance with equation 4.

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos(2(x+1)u\pi/16)\cos(2(y+1)v\pi/16) \quad \text{------ (4)}$$

Where

$C(x) = 1/\sqrt{2}$ x=0

$C(x) = 1$ Otherwise

The output of the transformation will result in the mean value, the DC coefficient, to be located on the top left corner of the data unit and higher frequency coefficients will be further away from this DC coefficient. These are called AC components.

### 5.2.3 Quantization

After output from the DCT, each of the 64 DCT coefficients is uniformly quantized in conjunction with a 64-element Quantization Table, which must be specified by the application (or user) as an input to the encoder [11]. Each element can be any integer value from 1 to 255, which specifies the step size of the quantizer for its corresponding DCT coefficient. The purpose of quantization is to achieve further compression by representing DCT coefficients with no greater precision than is necessary to achieve the desired image quality. Quantization is a many-to-one mapping, and therefore is fundamentally lossy. It is the principal source of lossiness in DCT-based encoders. Quantization is defined as division of each DCT coefficient by its corresponding quantizer step size, followed by rounding to the nearest integer.

$F_Q$ (u, v) = Integer Round (F (u, v)/Q (u, v)) ----- (5)

JPEG standard defines default quantization tables for both luminance and chrominance components. These tables can be used if user doesn't want to specify so. Following tables shows these tables.

Table 5.2 Luminance Quantization Table

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 16 | 11 | 10 | 16 | 24 | 40 | 51 | 61 |
| 12 | 12 | 14 | 19 | 26 | 58 | 60 | 55 |
| 14 | 13 | 16 | 24 | 40 | 57 | 69 | 56 |
| 14 | 17 | 22 | 29 | 51 | 87 | 80 | 62 |
| 18 | 22 | 37 | 56 | 68 | 109 | 103 | 77 |
| 24 | 35 | 55 | 64 | 81 | 104 | 113 | 92 |
| 49 | 64 | 78 | 87 | 103 | 121 | 120 | 101 |
| 72 | 92 | 95 | 98 | 112 | 110 | 103 | 99 |

Table 5-3 Chrominance Quantization Table

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 17 | 18 | 24 | 47 | 99 | 99 | 99 | 99 |
| 18 | 21 | 26 | 66 | 99 | 99 | 99 | 99 |
| 24 | 26 | 56 | 99 | 99 | 99 | 99 | 99 |
| 47 | 66 | 99 | 99 | 99 | 99 | 99 | 99 |
| 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
| 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
| 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
| 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |

### 5.2.4 Zig-zag Ordering

After quantization, the DC coefficient is treated separately from the 63 AC coefficients [11]. The DC coefficient is a measure of the average value of the 64 image samples. Because there is usually strong correlation between the DC coefficients of adjacent 8x8 blocks, the quantized DC coefficient is encoded as the difference from the DC term of the previous block in the encoding order as shown in following figure.

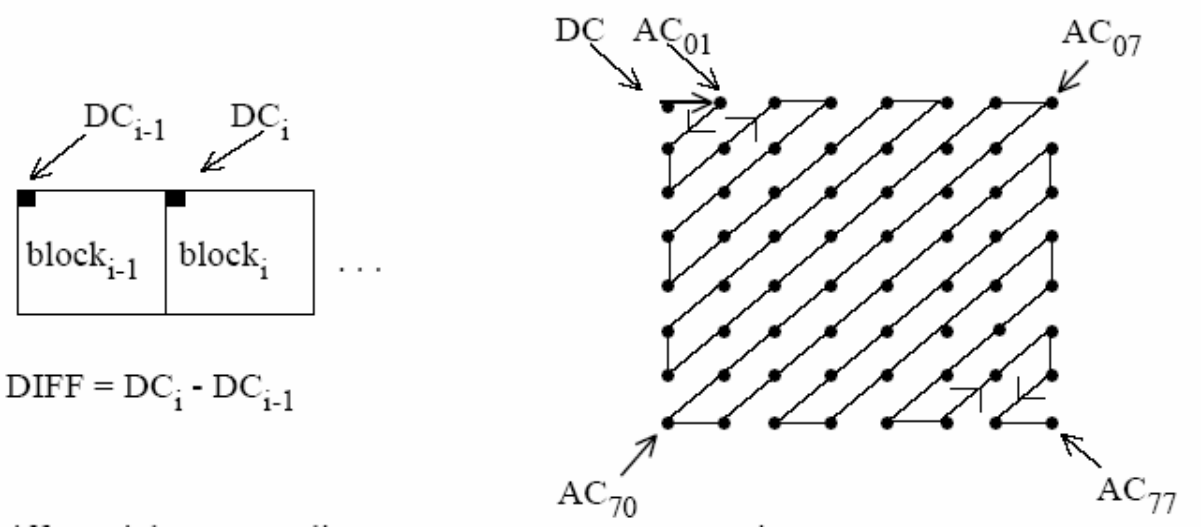


Figure 5-4 Differential DC encoding and Zig-zag sequence

After Differential DC encoding all of the quantized coefficients are ordered into the zig-zag sequence, also shown in above figure. This ordering helps to facilitate entropy coding by placing low-frequency coefficients (which are more likely to be nonzero) before high-frequency coefficients.

### 5.2.5 Entropy Coding

This is the final step of baseline process. This step achieves additional compression losslessly by encoding the quantized DCT coefficients more compactly based on their statistical characteristics. The JPEG proposal specifies two entropy coding methods - Huffman coding and arithmetic coding. The Baseline sequential codec uses Huffman coding [11].

Entropy coding is a two step process. First step is Run Length Coding and second step is Huffman coding. Huffman coding requires that one or more sets of Huffman code tables be specified by the application. The same tables used to compress an image are needed to decompress it. Huffman tables may be predefined and used within an application as defaults, or computed specifically for a given image. JPEG standard defines two Huffman tables one

for luminance components and other for chrominance components. These tables can be used as default if user doesn't want to define its own tables.

### 5.2.6 Implementation and Result

Various stages of JPEG baseline compression are explained above. From these stages DCT and quantization were implemented using 4 element vectors, while zig-zag ordering and entropy coding can be implemented as scalar because they can not be vectorized. The performance of these stages was measured individually for 8X8 block on single SPE as well as combined performance of both the stages was measured for entire image. The performance was measured in terms of clock cycles using simulator as explained in section 5.1.3. The performance results are shown in following table.

Table 5-4 Performance Results for DCT and Quantization

| | **Pentium 4** | **Cell Architecture** |
|---|---|---|
| **DCT (1 SPE, 8X8 Block)** | 1,032,710 | 289,036 |
| **Quantization (1 SPE, 8X8 Block)** | 9,584 | 4,064 |
| **DCT + Quantization (8 SPEs, 256X256 Image)** | 811,854,149 | 53,862,796 |

From above table we can see that a single SPE can perform (1,032,710)/ (289,036) = **3.57** times faster than P-4 for DCT, while for quantization it can perform (9,584)/ (4,064) = **2.35** times faster than P-4.

In the third case DCT and Quantization were performed on 256X256 gray scale image and performance was measured using 8 SPEs. In that case Speedup achieved is (811,854,149)/ (53,862,796) = **15.07**.

# 6 CONCLUSION AND FUTURE WORK

Throughout the thesis the architecture of the Cell Broadband Engine, programming methods for Cell and partitioning the application and data among cores have been studied. Then Image processing algorithms for Cell BE have been implemented and results have been compared with Pentium 4. This implementation led to the following conclusion:

The CELL is a multi-core architecture in which all the cores can be programmed separately from each other. It can perform many times faster than conventional (single-core) systems for applications which can be divided in such a way that power of all the cores can be fully utilized. Image processing applications are best suited for such environment because it requires similar operations to be performed on entire image. Hence work can be evenly distributed. However this distribution task is very tedious and requires lots of programming efforts.

The algorithms of Spatial domain filters, DCT and Quantization were implemented for Cell and performance was measured for 256X256 gray scale image on both Cell as well as Pentium 4. From the results it has been observed that the CELL performed around 11 times faster than Pentium 4 for Spatial Domain filters while for DCT and Quantization, it performed around 15 times faster. Thus Cell not only performed better than Pentium 4 but it performed better for application requiring more computations.

## 6.1 Future Work

As shown in section 5.1.3 the theoretical performance is 57 while performance achieved practically is around 11. The performance can still be improved by using advanced programming methods (section 4.5) such as double buffering and proper inter-processor communication.

Throughout this thesis all the algorithms were developed using parallel stage model (Section 4.2) i.e. same task is assigned to all the SPEs and all SPEs work in parallel. Instead of this multistage pipeline model can be used for application like JPEG or MPEG compression. Such applications requires sequential stages, the task of each stage can be assigned to different SPE. In this model the SPEs can act as a multistage pipeline. In this case the data flows from one SPE to other. The algorithms can be implemented using this approach and performance can be compared with parallel model.

# REFERENCES


[1] Cell Broadband Engine Programming Handbook, version 1.0, cellsdk/pdfs.

[2] Cell Broadband Engine Programming Tutorial, version 2.1, cellsdk/pdfs.

[3] David Krolak," The Element Interconnect Bus", Papers from the Fall Processor Forum 2005, http://www.ibm.com/developerworks/power/library/pa-fpfeib/

[4] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/CELL-Linux-L20070831ADDON/doc/CellProgrammingTutorial/BasicsOfCellArchitecture.html

[5] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/CELL-Linux-CL20070831_ADDON/doc/CellProgrammingTutorial/BasicsOfSPEProgramming.html

[6] IBM BladeCenter QS20, cellsdk/pdfs.

[7] http://www.ibm.com/developerworks/power/library/pa-linuxps3-1/

[8] Software Development Kit 2.1 Installation Guide, Version 2.1, cellsdk/pdfs.

[9] IBM Full-System Simulator User's Guide, Version 2.1, cellsdk/pdfs.

[10] Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", Pearson Prentice Hall

[11] Gregory K. Wallace, “The JPEG Still Picture Compression Standard”, IEEE Transactions on Consumer Electronics, December 1991.

[12] ITU –T81 “Information Technology – Digital Compression And Coding Of Continuous-Tone Still Images – Requirements And Guidelines” www.w3.org/Graphics/JPEG/itu-t81.pdf

[13] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/CELL-Linux-CL_20080201-ADDON/doc/CellProgrammingTutorial /AdvancedCellProgramming.html

# APPENDIX A LIBSPE

## A-1 Overview

The SPE Runtime Management Library (libspe) is the standardized low-level application programming interface (API) that enables application access to the Cell/B.E. SPEs. This library provides an API that is neutral with respect to the underlying operating system and its methods to manage SPEs. Implementations of libspe can provide additional functionality that enables access to operating system or implementation-dependent aspects of SPE runtime management. In general, applications do not have control over the physical SPE system resources. The operating system manages these resources. Applications manage and use software constructs called SPE contexts. These SPE contexts are a logical representation of an SPE and are the base object on which libspe operates. The operating system schedules SPE contexts from all running applications onto the physical SPE resources in the system for execution according to the scheduling priorities and policies associated with the runable SPE contexts. Libspe also provides the means for communication and data transfer between PPE threads and SPEs.

The basic scheme for a simple application using an SPE is as follows:

1. Create an SPE context.
2. Load an SPE executable object into the SPE context local store.
3. Run the SPE context. This transfers control to the operating system, which requests the actual scheduling of the context onto a physical SPE in the system.
4. Destroy the SPE context.

## A-2 PPE functions

To provide this functionality, libspe consists of the following sets of PPE functions to:

- Create and destroy SPE and gang contexts
- Load SPE objects into SPE local store memory for execution

- Start the execution of SPE programs and to obtain information about reasons why an SPE has stopped running
- Receive asynchronous events generated by an SPE
- Access the MFC (Memory Flow Control) problem state facilities

Some useful functions for creating SPE context and start execution of SPE program are given below:

- **spe_context_create:** Create a new SPE context.
- **spe_context_destroy:** Destroy the specified SPE context.
- **spe_image_open:** Open an SPE ELF executable.
- **spe_image_close:** Close an SPE ELF object.
- **spe_program_load:** Load an SPE main program.
- **spe_context_run:** Request execution of an SPE context.

# APPENDIX B SIMPLE SCALAR SIMULATOR

The Simple-Scalar tool set is a system software infrastructure used to build modelling applications for program performance analysis, detailed micro architectural modelling, and hardware-software co-verification. Using the Simple Scalar tools, users can build modelling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.

Simple-Scalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of Simple-Scalar can run programs from any of the above listed instruction sets.

The Simple-Scalar distribution comes with three types of simulators: Sim-fast, sim-safe and sim-outorder. Sim-fast does no time accounting, only functional simulation—it executes each instruction serially, simulating no instructions in parallel. Sim-fast is optimized for raw speed, and assumes no cache, instruction checking. A separate version of sim-fast, called sim-safe, also performs functional simulation, but checks for correct alignment and access permissions for each memory reference. The most complicated and detailed simulator in the distribution, by far, is sim-outorder. This simulator supports out-of-order issue and execution and gives the exact timing measurements.

As given above, Simple-Scalar simulator can simulate x86 instruction sets. We can pass configuration of Pentium-4 to Simple-Scalar so that it will give exact timing analysis for Pentium-4. Table B-1 gives the exact Pentium-IV processor's configuration.

Table B-1: Pentium 4 Processor Configuration

| | |
|---|---|
| Instruction fetch queue size (in insts) -fetch: ifqsize | 64 |
| Extra branch mis-prediction latency -fetch: mplat | 3 |
| bimodal predictor BTB size -bpred:bimod | 2048 |
| 2-level predictor config (<l1size> <l2size><hist_size>) -bpred:2lev | 1 1024 8 |
| Instruction decode B/W (insts/cycle) -decode:width | 4 |
| Instruction issue B/W (insts/cycle) -issue:width | 4 |
| run pipeline with in-order issue -issue: inorder | false |
| issue instructions down wrong execution paths -issue: wrongpath | true |
| register update unit (RUU) size -ruu: size | 16 |
| load/store queue (LSQ) size -lsq: size | 8 |
| l1 data cache config, i.e., {<config>\|none} -cache: dl1 | dl1:128:64:4:l |
| l1 data cache hit latency (in cycles) -cache: dl1lat | 1 |
| l2 data cache config, i.e., {<config>\|none} -cache:dl2 | ul2:16384:64:8:l |
| l2 data cache hit latency (in cycles) -cache: dl2lat | 6 |
| l1 inst cache config, i.e., {<config>\|dl1\|dl2\|none} -cache:il1 | il1:512:32:1:l |
| l1 instruction cache hit latency (in cycles) -cache:il1lat | 2 |
| l2 instruction cache hit latency (in cycles) -cache:il2lat | 7 |
| flush caches on system calls -cache:flush | false |
| convert 64-bit inst addresses to 32-bit inst equivalents -cache: compress | false |
| memory access latency (<first_chunk> <inter_chunk>) -mem:lat | 18 2 |
| Memory access bus width (in bytes) -mem: width | 8 |
| Instruction TLB config, i.e., {<config>\|none} -tlb: itlb | dtlb: 16:4096:4:l |
| data TLB config, i.e., {<config>\|none} -tlb: dtlb | dtlb: 16:4096:4:l |
| inst/data TLB miss latency (in cycles) -tlb: lat | 30 |
| total number of integer ALU's available -res:ialu | 2 |
| total number of integer multiplier/dividers available -res:imult | 1 |
| total number of floating point ALU's available -res:fpalu | 1 |
| number of floating point multiplier/dividers available -res:fpmult | 1 |

## APPENDIX C AMDAHL'S LAW

Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S. (For example, if an improvement can speed up 30% of the computation, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2.) Amdahl's law states that the overall speedup of applying the improvement will be

$$Speedup = \frac{1}{(1-P) + \frac{P}{S}}$$

To see how this formula was derived, assume that the running time of the old computation was 1, for some unit of time. The running time of the new computation will be the length of time the unimproved fraction takes, (which is 1 − P), plus the length of time the improved fraction takes. The length of time for the improved part of the computation is the length of the improved part's former running time divided by the speedup, making the length of time of the improved part P/S. The final speedup is computed by dividing the old running time by the new running time, which is what the above formula does.

Similarly in case of parallelization, Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and (1 − P) is the proportion that cannot be parallelized

(remains serial), then the maximum speedup that can be achieved by using N processors is

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

In the limit, as N tends to infinity, the maximum speedup tends to 1 / (1-P).

As an example, if P is 90%, then (1 – P) is 10%, and the problem can be sped up by a maximum of a factor of 10, no matter how large the value of N used. For this reason, parallel computing is only useful for either small numbers of processors, or problems with very high values of P: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce (1-P) to the smallest possible value.