CBE Overview

Table of Contents

1 Introduction

This document outlines the functions and capabilities of the CBE chip, the first-generation implementation of the general-purpose System-on-Chip (SoC) architecture, CBEA. In particular, the logical and implementation specifications characteristic to the CBE that affect performance will be discussed. Details regarding the use of specific functions will be provided in the following documents.

Reference Materials

User's Manuals

- PPE User's Manual
- SPE User's Manual

Logical Specifications

- Cell Broadband Engine™ Architecture
- Synergistic Processor Unit Instruction Set Architecture
- PowerPC Architecture Book, Version 2.02
  - Book I: PowerPC User Instruction Set Architecture
  - Book II: PowerPC Virtual Environment Architecture
  - Book III: PowerPC Operating Environment Architecture

  (International Business Machines Corporation, last updated: February 24, 2005)
  http://www-106.ibm.com/developerworks/eserver/library/es-archguide-v2.html

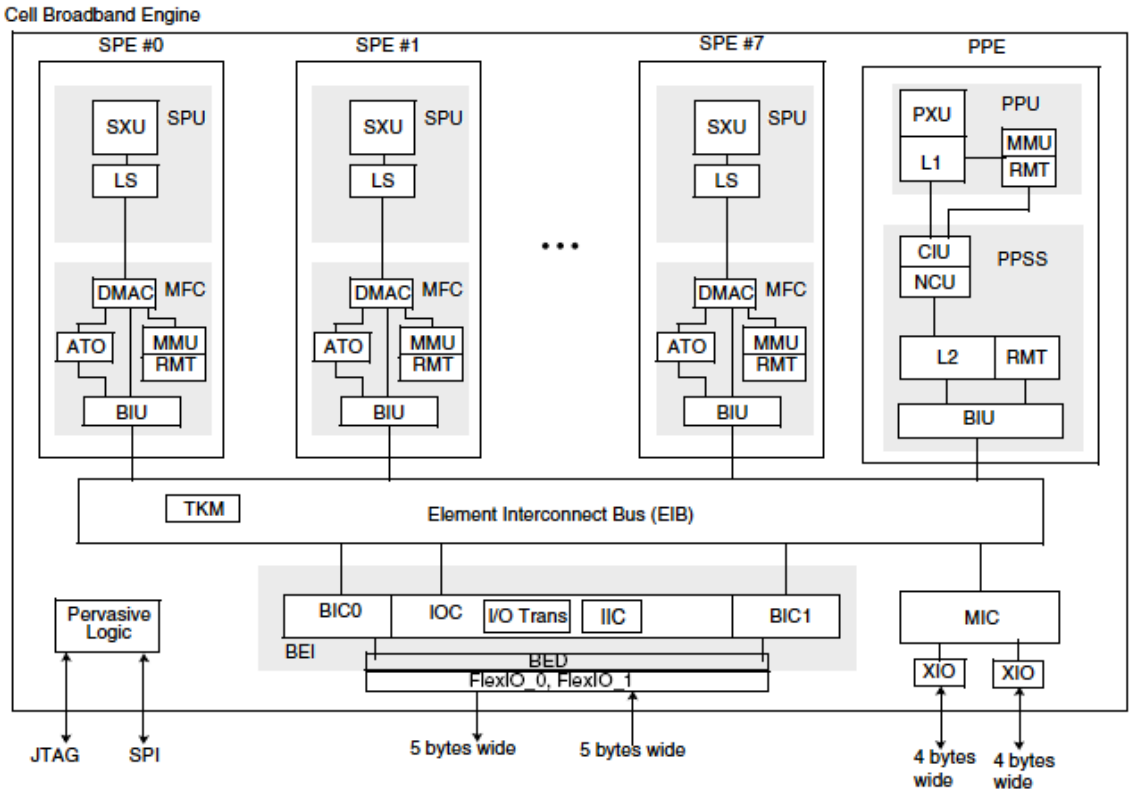
  (The above reference was available on December 13, 2006. Note, however, that it is possible for the applicable page to be moved or modified.)

2 CBE Overview and Components

CBE is the first SoC (System on Chip) implementation based on CBEA. It has elements of an embedded system — a processor core that realizes a high cost performance through a DSP-like pipeline configuration, and a high manageability realized through explicit data transfers independent of cache systems — as well as elements of the newest server systems, such as logical partitioning and a shared-memory multiprocessor system. Figure 1 is a schematic diagram of the CBE chip. The main modules of CBE, which are connected via a proprietary, shared-memory bus called EIB, are as follows.

- PPE
- Eight SPEs
- IOC+BIC
- MIC

Figure 1 CBE Overview



ATO	Atomic Unit	MFC	Memory Flow Controller
BED	CBE Distribution	MIC	Memory Interface Controller
BEI	Cell Broadband Engine Interface Unit	MMU	Memory Management Unit
		NCU	Noncacheable Unit
BIC	Bus Interface Controller	PPE	PowerPC Processor Element
BIU	Bus Interface Unit	PPSS	PowerPC Processor Storage Subsystem
CIU	Core Interface Unit	PPU	PowerPC Processor Unit
DMAC	Direct Memory Access Controller	PXU	Processor Execution Units (without L1)
EIB	Element Interconnect Bus	RMT	Replacement Management Table
IIC	Internal Interrupt Controller	SPE	Synergistic Processor Element
IOC	I/O Interface Controller	SPI	Serial Peripheral Interface
I/O Trans	I/O Address Translation	SPU	Synergistic Processor Unit
JTAG	IEEE 1149 test access port	SXU	Synergistic Execution Units (without LS)
L1	Level 1 Cache	TKM	Token Management Unit
L2	Level 2 Cache	XIO	Extreme Data Rate (XDR) I/O Cell
LS	Local Store		

PPE and **SPE** are processor units with different characteristics. The differences between these two types range from the nature of the instruction set to the directionality of pipeline tuning.

SPE has a local memory (local storage: LS) instead of a cache hierarchy run by most common processors. Since memory outside LS cannot be accessed with load/store instructions, SPE relies on a DMA transfer unit called MFC for all external data transfers. In addition to high cost performance capabilities similar to DSP, the processor core has a DMA engine that can access virtual memory spaces and a synchronization mechanism compatible with PowerPC Architecture. For this reason, it can coexist with, and share user data with, PowerPC Architecture-compatible processors all under the management of an operating system.

In normal operation, PPE and SPE operate on the same processor clock cycle. The CBE has one PPE and eight SPEs; the latter is where most of the chip's computation capabilities lie. Therefore, to make effective and efficient use of the CBE, it is important to execute jobs requiring high computing capabilities as much as possible on the SPEs.

The PPE can use software currently available for PowerPC Architecture-compatible processors. Because some system management functions are executable only by the PPE, the core of the operating system is executed in the PPE.

The main memory of CBE is XDR DRAM, which is directly attached to the CBE on an 8-byte wide XDR DRAM interface (two 4-byte wide ports). Though the specifications of the XDR DRAM interface allow up to 24 memory devices to be connected, configurations that are actually usable will depend on the specifications of the usable XDR DRAM memory devices. **MIC** is a memory interface unit including the XDR DRAM memory controller. It receives memory access requests from EIB and carries out buffering, scheduling, and data transfers.

The external interface Flex I/O is a bus interface that can be used either as a multiprocessor bus interface (BIF) or an I/O bus (IOIF) depending on the mode switch. The bus width can be adjusted in byte units. **BIC** is the running controller when Flex I/O is in BIF mode; **IOC** is the controller in IOIF mode. In this document, only IOIF will be discussed. I/O devices such as the network adaptor, graphics controller, and HDD controller, are connected to the CBE via IOIF. When thus connected, I/O devices with bus master functionality can access memory spaces inside the CBE directly, and internal CBE masters (PPE and SPE) can directly access the I/O devices connected to the IOC.

3 Bus Structure and Transfer Capabilities

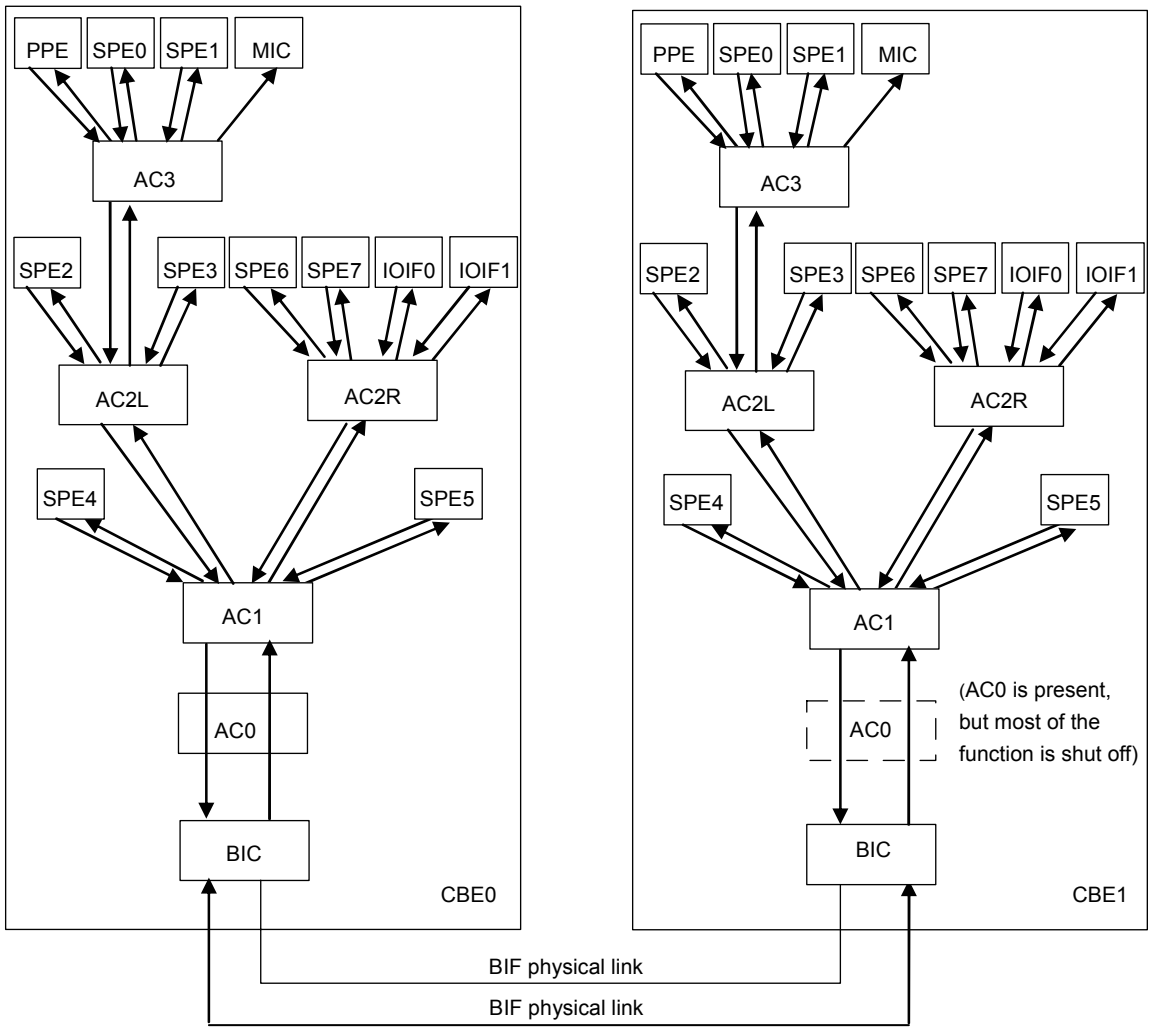
EIB

The internal modules of CBE — PPE, SPE, and others — are connected via EIB. EIB comprises a tree-structure command bus and a fourfold-ring-structure data bus.

Figure 2 illustrates the structure of the command bus (AC tree). In this diagram, two CBE chips are connected to each other with BIF links. A single CBE system command bus structure consists of the CBE tree structure on the left (CBE0), excluding the BIC (the interface to CBE1, the other CBE).

When access requests generated per master device reach the root of the tree-structure bus (AC0) via the command bus, the access requests are serialized and then broadcast to all master devices and target devices from there. Because of this structure, all devices inside the CBE receive requests in the same order. Devices then check the state of the requested memory block and return the result. Reply messages are first merged in AC0 and then broadcast to each device.

Figure 2 AC Tree Structure



Command transactions complete in this way, and then data transactions (data transfers from source to destination) are carried out. The maximum processing capability of the command bus is 1 access request per 2 processor cycles. A maximum of 128 bytes of data can be transferred in 1 access request.

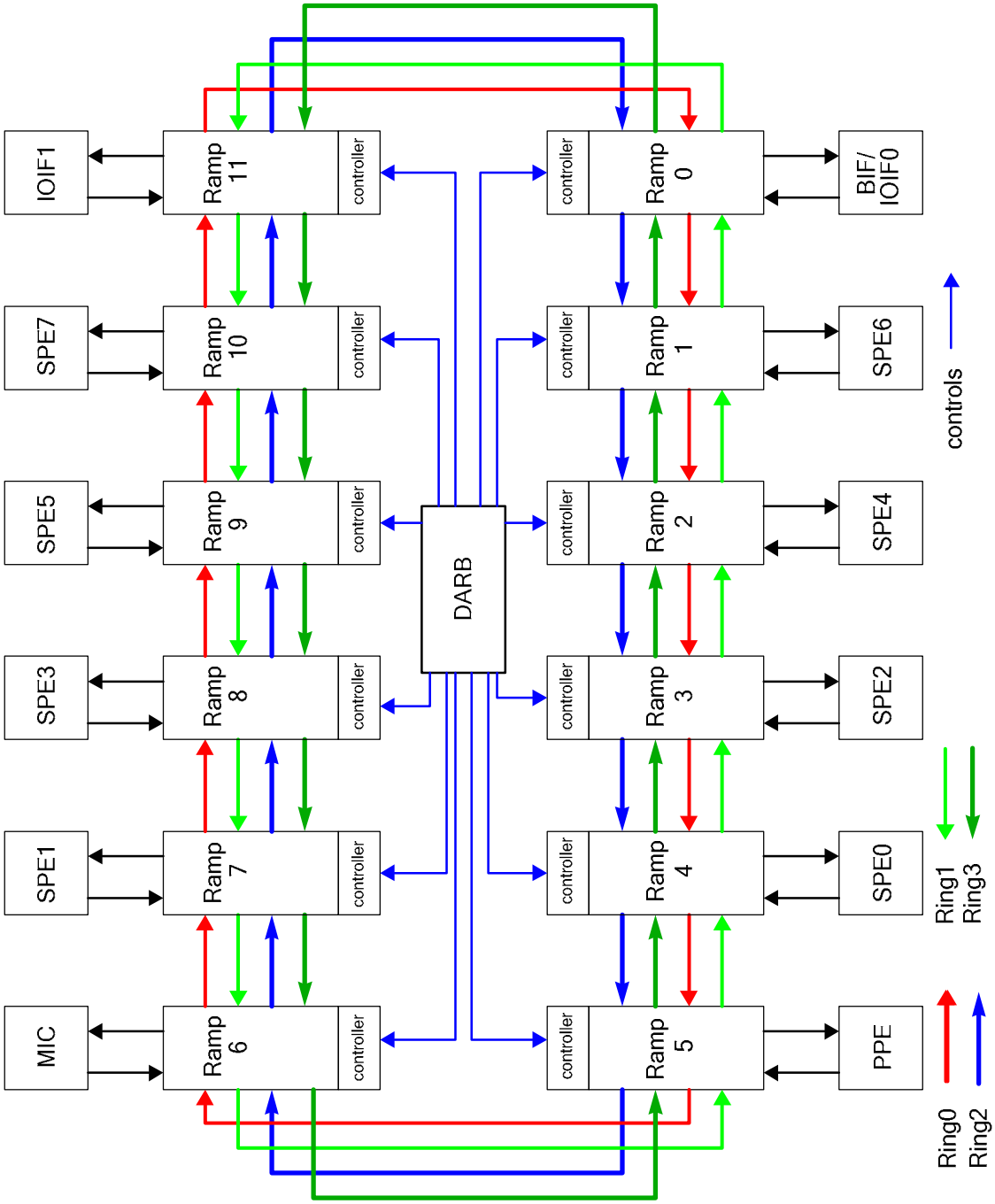
Figure 3 illustrates the EIB ring-structure data bus. Each of the paths between the nodes comprising the four rings has a data transfer capability of 8 bytes per processor cycle. However, since multiple data transfer transactions can be executed simultaneously on a ring, the total data transfer capability is more than 32 (8*4) bytes per processor cycle.

There are two main constraints on the performance of the EIB data ring. One is the maximum capability of the arbiter, which is 1 transaction per 3 processor cycles for each data ring (1 transaction corresponds to 1 access request). This limit restricts the maximum performance of the data bus. The other constraint is the possibility that a conflict may occur on the path between nodes depending on the transfer pattern. In certain cases a conflict may occur even though the transfers are topologically possible, because of restrictions on the arbiter's control algorithms.

In effect, the restrictions of the command bus and the data transfer bus limit the maximum number of memory access requests that the EIB can process to 1 request per 2 processor cycles. Since the maximum of data transferred in 1 request is 128 bytes, the maximum data transfer capability of EIB is 64 bytes per processor cycle.

The CBE memory system (including EIB) and data transfer system are designed so that peak performance can be attained when transferring 128-byte data aligned on natural address boundaries. Transfer bandwidths nearing peak performance are possible only if data is formatted in aligned 128-byte block units, or data accesses are executed on sufficiently large consecutive block units.

Figure 3 EIB Ring Structure



IOIF

The CBE is connected to external devices via two I/O interfaces, IOIF0 and IOIF1. IOIF0 and IOIF1 are variable-width interfaces with independent input/output lines and different ranges of byte-width variability. The byte-width ranges of IOIF0 and IOIF1 are as follows.

	CBE to External Device Bus Width	External Device to CBE Bus Width
IOIF0	0 to 4 bytes	0 to 4 bytes
IOIF1	0 to 2 bytes	0 to 2 bytes
IOIF0+IOIF1	Maximum of 5 bytes	Maximum of 5 bytes

IOIF0 is used for devices that require high-speed data transfers (such as graphics processors) and IOIF1 is for devices that do not require such high speeds. Except for the bus width, the two interfaces have the same specifications in terms of function and bus protocol.

Since IOIF multiplexes the address and data on the same physical interface, the effective data transfer capability is smaller than the I/O signal bandwidth (bus width * I/O signal rate).

IOIF supports half-rate mode. In half-rate mode, the speed is half that of full-rate mode, but physical implementation restrictions such as cable length and connector specifications are mitigated.

In the Reference Tool system, IOIF0 will be set to the full rate of 4-byte output and 3-byte input, and IOIF1 will be set to the half rate of 1-byte output and 1-byte input.

MIC

The CBE uses XDR DRAM as its main memory, which is connected to the CBE via an 8-byte wide interface. The memory interface is not simply one port, but is composed of two independent 4-byte ports. Since the address bus is independent of the data bus, the maximum data bandwidth of the memory system is 8 bytes per memory cycle.

Due to the characteristics of XDR DRAM, a performance penalty will occur if read accesses and writes to the same port alternate. MIC can mask this penalty by carrying out reads (or writes) in succession to some extent when scheduling access requests in the request buffer.

The CBE XDR DRAM has 8 banks per port and comprises 16 memory banks altogether. The bank number is chosen according to the lower bits of the 128-byte block address. All banks are accessed once each if 2KB of consecutive blocks is accessed. To realize a high effective bandwidth, it is necessary to access all banks evenly.

MIC carries out memory accesses in aligned units of 128 bytes, like CBE's other data transfer mechanisms. Accesses to blocks smaller than 128 bytes are processed the same as accesses to 128-byte blocks — as one memory access transaction — so the effective access capability drops when smaller blocks are accessed.

MIC supports partial writes and thereby faster writes to parts of a 128-byte block. When MIC writes to sub-blocks smaller than 128 bytes, it updates the sub-block directly instead of reading, updating, and writing back the entire 128-byte block. For partial writes to be effective, the size of a sub-block must be a multiple of 16 bytes and the sub-block must be aligned in units of 16 bytes.

Bandwidth Management

The stability of memory system bandwidths, in particular the effective bandwidth and latency of external interfaces, is an important element in the stable execution of real-time applications. Bandwidth management on the CBE is concerned with the three external interfaces, IOIF0, IOIF1, and XDR DRAM.

Bandwidth Management Overview

Bandwidth management is carried out in RAG (Resource Allocation Group) units. All master devices (not only the internal PPE and SPEs but also the I/O devices connected to IOIF) are assigned to one of the four RAGs. The total bandwidth used by all the master devices of an RAG can be controlled. How much of each bandwidth (IOIF0, IOIF1, XDR DRAM) is allocated to each RAG can be defined.

If only one RAG is using the bandwidth, it can dominate the entire bandwidth. However, if all RAGs attempt to use resources exceeding their allocated bandwidths, then the use of the bandwidth will be adjusted according to the predefined ratios.

Example:

SPE0, SPE1, SPE2 are allocated to RAG0
SPE3, SPE4, SPE5 are allocated to RAG1
XDR DRAM assignment: RAG0:RAG1 = 50%:40%

Case 1:

If SPE0-2 are suspended, and SPE3-5 attempt to use the entire XDR DRAM bandwidth, RAG1 can use the entire XDR DRAM bandwidth. The three master devices assigned to RAG1 (SPE3-5) are given equal bandwidths in round-robin fashion.

Case 2:

If SPE0-2 and SPE3-5 attempt to use the entire XDR DRAM bandwidth, the bandwidths used by RAG0 and RAG1 respectively will be adjusted to reflect the ratio (50%:40%) specified in the resource management table.

If there are multiple RAGs that can use the leftover bandwidth, the priority of each RAG in using the extra bandwidth can also be defined.

Example:

SPE0, SPE1, SPE2 are allocated to RAG0
SPE3, SPE4, SPE5 are allocated to RAG1
SPE6, SPE7 are allocated to RAG2
XDR DRAM assignment: RAG0:RAG1:RAG2 = 20%:20%:30%
Priority over the use of extra bandwidths: RAG0 > RAG1 > RAG2

If SPE0-2 are suspended, SPE3-5 attempt to use 60% of the XDR DRAM bandwidth, and SPE6-7 attempt to use the entire XDR DRAM bandwidth, RAG1 can use 60% of the XDR DRAM bandwidth as requested. RAG2 can use the rest of the XDR DRAM bandwidth (40%).

The bandwidth management mechanism does not take into account all factors affecting the memory system. For this reason, if the RAG units are set to consume 100% of the total bandwidth, it is possible for the actual load on resources to exceed 100%, which can adversely affect system performance by increasing memory access latencies, for example. On the other hand, due to characteristics of the memory system implementation, bandwidths and latencies can actually improve by maintaining a certain load on the memory system. Therefore it is always necessary to determine the settings appropriate for the actual system configuration and usage conditions.

One-way Interface Pairs

IOIF0 and IOIF1 have independent input and output lines. Therefore, it is necessary to manage the input bandwidth (from external devices to the CBE) separately from the output bandwidth (from the CBE to external devices). For example, the input bandwidth will be used when an internal CBE master reads data from an I/O device, and when writing data, the output bandwidth. In turn, the output bandwidth will be used when an external I/O device reads data from an internal CBE memory, and when writing data, the input bandwidth.

Therefore, IOIF0's input/output lines and IOIF1's input/output lines are handled as individual resources, and are configured so that bandwidths can be allocated separately to each.

Accesses Using Two Resources

Depending on the access pattern, one request may use multiple resources. For example, when an external device connected to IOIF0 reads data from XDR DRAM via EIB, this access uses both the IOIF0 output bandwidth and the XDR DRAM bandwidth. In such cases, bandwidth management must take into account that one access uses both bandwidths.

Multi-bank Memory

XDR DRAM has 16 memory banks. If the 16 banks are accessed equally, the peak bandwidth of the XDR DRAM interface can be used; on the other hand, if accesses are only to certain banks, the effective bandwidth will decrease. From the point of view of resource loads, poorly distributed accesses will have 16 times as much of an access load as that of evenly distributed accesses, when on the same memory access bandwidth. Therefore, any method of managing memory access bandwidths will most likely fail if it does not take banks into consideration.

In view of this situation, the current bandwidth management handles each bank as an individual resource. When a master device accesses XDR DRAM, the applicable bank is identified from the address, and then bandwidth management of that particular bank is carried out. In addition to the bandwidth of each bank, the total bandwidth of the two XDR DRAM interfaces will also be managed separately as an individual resource.

Management Data

In summary, the bandwidth management mechanism handles 22 resources (2 XDR DRAM interfaces, 16 banks, IOIF0 input/output, IOIF1 input/output). The bandwidths of these resources are allocated to the 4 RAGs.

Performance Characteristics

Measurement results of memory system performance for a typical case are shown below. The core clock frequency and the XDR DRAM interface data rate are 3.2 GHz. The following values are actual measurement data using the Reference Tool; note, however, that memory access latencies and bandwidths depend on system settings and operation conditions, and results may differ in an actual use environment.

Table 1 Performance Data for PPU-Initiated Requests

	Data Transfer Size	Resource Allocation Off	Resource Allocation On
L1 hit read latency	8B	1.6 ns	1.6 ns
L2 hit read latency	8B	12.6 ns	12.6 ns
L2 miss read latency	8B	124.7 ns	124.7 ns
L1 hit read bandwidth, FXU load	8B	23.3 GB/s	23.3 GB/s
L1 hit read bandwidth, FPU load	8B	12.6 GB/s	12.6 GB/s
L1 hit read bandwidth, VMX load	16B	10.1 GB/s	10.1 GB/s
L2 hit read bandwidth, FXU load	8B	7.2 GB/s	7.2 GB/s
L2 hit read bandwidth, FPU load	8B	6.4 GB/s	6.4 GB/s
L2 hit read bandwidth, VMX load	16B	5.3 GB/s	5.3 GB/s
L2 hit read bandwidth, DCBT L2->L1	128B	69.7 GB/s	69.9 GB/s
L2 miss read bandwidth, FXU load	8B	1.0 GB/s	1.0 GB/s
L2 miss read bandwidth, FPU load	8B	1.0 GB/s	1.0 GB/s
L2 miss read bandwidth, VMX load	16B	0.9 GB/s	0.9 GB/s
L2 miss read bandwidth, DCBT XDR->L1	128B	5.8 GB/s	5.8 GB/s
L2 hit write bandwidth, FXU store	8B	11.1 GB/s	11.1 GB/s
L2 hit write bandwidth, FPU store	8B	12.8 GB/s	12.8 GB/s
L2 hit write bandwidth, VMX store	16B	18.6 GB/s	18.6 GB/s
L2 hit write bandwidth, DCBZ L1->L2	128B	17.6 GB/s	17.6 GB/s
L2 miss write bandwidth, FXU store	8B	4.9 GB/s	4.9 GB/s
L2 miss write bandwidth, FPU store	8B	4.9 GB/s	4.9 GB/s
L2 miss write bandwidth, VMX store	16B	5.7 GB/s	5.5 GB/s
L2 miss write bandwidth, DCBZ L1->XDR	128B	12.7 GB/s	12.6 GB/s
SPU LS read latency	4B	63ns	63ns
SPU LS read bandwidth, FXU load	8B	0.12 GB/s	0.12 GB/s
SPU LS read bandwidth, FPU load	8B	0.11 GB/s	0.11 GB/s

	Data Transfer Size	Resource Allocation Off	Resource Allocation On
SPU LS read bandwidth, VMX load	16B	0.22 GB/s	0.23 GB/s
SPU LS write bandwidth, FXU load	8B	1.58 GB/s	1.63 GB/s
SPU LS write bandwidth, FPU load	8B	1.58 GB/s	1.63 GB/s
SPU LS write bandwidth, VMX load	16B	1.61 GB/s	1.66 GB/s

Table 2 Performance Data for SPE-Initiated Requests

	Data Transfer Size	Resource Allocation Off	Resource Allocation On
DMA Get from Memory latency	128B	180 ns	155 ns
DMA Get from Memory bandwidth	128B	13.2 GB/s	15.8 GB/s
DMA Put to Memory bandwidth	128B	22.8 GB/s	25.1 GB/s
Simultaneous DMA Get and Put (2SPUs)	128B	20.3 GB/s	20.6 GB/s
Inter-Local Storage DMA Get bandwidth	128B	23.6 GB/s	22.9 GB/s
Inter-Local Storage DMA Put bandwidth	128B	25.6 GB/s	25.6 GB/s

4 Address Space

The CBE has a single real address space that can be accessed by all master devices (PPE, SPEs, I/O devices via IOIF). In addition to normal read-write memory for placing instructions and data, the real address space also includes MMIO registers that when accessed, validate various settings or execute DMA engine kicks. The main elements of the real address space are as follows.

- Main Memory (normal memory)
- SPE LS (normal memory)
- SPE MFC Control Registers (MMIO registers)
- Other Control Registers (MMIO registers)
- I/O Space (normal memory, MMIO registers)

PPE and SPEs have one MMU each and support the address conversion method defined by PowerPC Architecture. Effective addresses generated by PPE load/store instructions or the MFC are translated into real addresses via a two-step translation mechanism using SLB and TLB.

For an application program to access an area in real address space, the operating system must map to the effective address space in units of pages (the smallest size is 4KB and is variable). At this time, the behavior of the memory system when a processor accesses a page can also be specified by changing the properties of the page.

The **Write Through Required attribute** requests that the page be managed according to the write-through policy when cached.

The **Caching Inhibited attribute** forbids loading the page into cache memory. Performance will decline significantly since the processor will have to access memory directly instead of through the cache.

The **Memory Coherence Required attribute** requests that the page maintain coherence when cached.

The **Guarded attribute** inhibits the processor from executing speculative memory accesses. Speculative memory accesses aim to improve performance by executing memory accesses that are not yet certain to be needed. An example of a speculative memory access is when a read access on a branch path is executed before the branch direction is resolved.

The Guarded attribute by itself does not have much significance since neither the PPE nor the SPE executes speculative memory accesses on the CBE, but when used in conjunction with the Caching Inhibited attribute, it affects the ordering of memory accesses described in "Section 7: Processor Synchronization."

By setting page protection conditions, whether or not the contents of a page are read, written to, or executed can be controlled.

Of the objects that are mapped to memory, pages that include MMIO registers are normally given the Caching Inhibited attribute and the Guarded attribute. Thus the programmer can carry out accesses to MMIO registers as intended, by inhibiting the optimization of memory accesses by the cache or by the memory interface of the processor core.

The LS of each SPU is given the Caching Inhibited attribute when mapped. This is because the consistency of the cache cannot be maintained if LS is cached, since LS accesses via SPU load/store instructions are not included in cache coherence maintenance processes.

Example:

```
PPE                                  SPE0
load LS location A  --------->       store LS location A

* location A is the location of memory in SPE0 LS mapped to in cacheable state.
* The store of SPE0 will be executed after the load of PPE completes.
```

In this example, after the PPE caches LS Location A, SPE0 updates LS Location A with an SPU store instruction. At this time, the data value in the PPE cache and the value of LS Location A as seen from the SPU will not be consistent.

5 PPE

The PPE is a processor compatible with 64-bit PowerPC ISA (instruction set architecture) and supports the PowerPC SIMD extension, VMX (AltiVec). Figure 4 and Figure 5 illustrate the core pipeline configuration of the PPE.

The PPE issues two instructions per cycle in program order. Registers are not renamed. The PPE has a fully pipelined 64-bit integer operation pipeline and a double-precision floating-point multiply-add pipeline. The PPE has one 128-bit wide VMX unit, and principal VMX instructions can be issued every cycle.

The PPE has a 32KB 2-way set associative L1 instruction cache, a 32KB 4-way set associative L1 data cache, and a 512KB 8-way set associative L2 instruction-and-data cache. The L1 data cache is controlled according to the write-through policy, and the L2 cache according to the write-back policy.

Figure 4 PPU Pipeline (Branch and Fixed-Point)

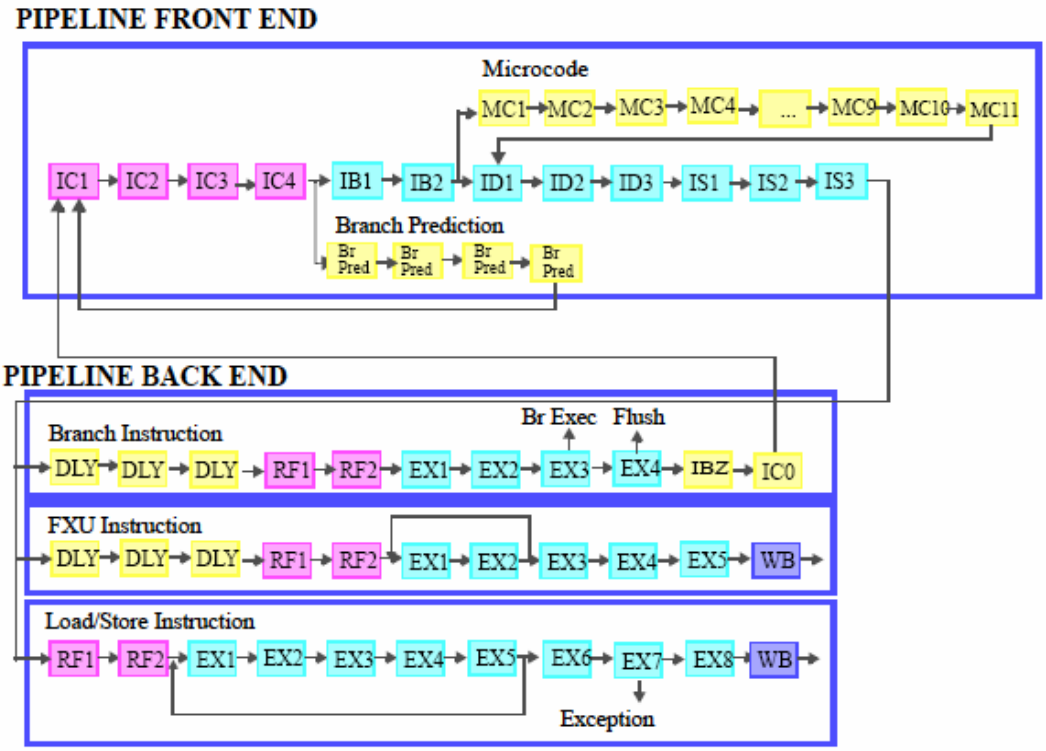
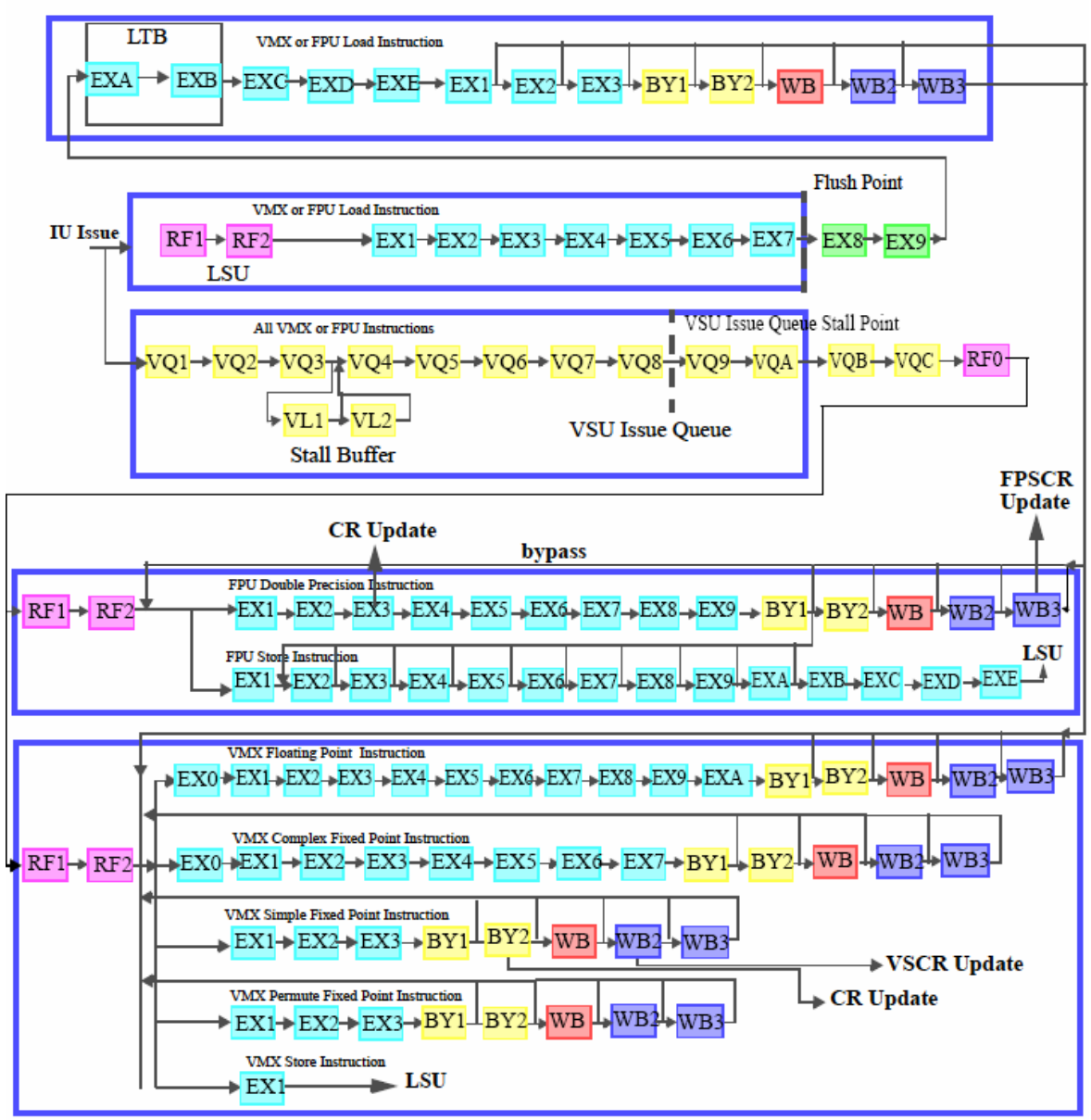


Figure 5 PPU Pipeline (Floating point and VMX)



The PPE supports multithreading. The main pipelines are single-threaded but have two threads' worth of register files and other resources so that two threads can be executed simultaneously without overhead for context switching. When multithreading, the PPE appears virtually to the software as two cores. Though the execution performance of each thread may be lower than that of single-thread mode, the combined instruction throughput of the two threads may be higher.

The PPE supports logical partitions (LPARs). An LPAR runs multiple independent operating systems simultaneously on a single CBE chip by supporting the Hypervisor, which has higher execution rights than normal operating systems.

The main memory is divided into several contiguous physical memory blocks and each is assigned to a partition. Each I/O device is also under the management of a partition. An operating system running inside a partition cannot access another partition's resources unless permitted by the Hypervisor. Therefore, even if the operating system of a certain partition crashes, or is hacked by an intruder, the operating systems and applications running in other partitions are protected as long as the Hypervisor is not affected.

L2 Replacement Management

High-capacity caches are effective in improving the average performance of non-real-time jobs, but incur large penalties in the worst-case scenario. There are situations in a real-time system when it is important to control penalties incurred by cache misses. To this end, the PPE L2 has a replacement management mechanism.

L2 has an 8-way set associativity. Therefore when a cache miss occurs, one of eight sets is selected for replacement. Normally, a set is selected in view of optimizing the cache's average hit rate, but L2 replacement management allows the programmer to control the replacement method and therefore to choose which data to leave in the cache.

With L2 replacement management, a correspondence relationship can be set up between the effective address range and the list of cache set numbers. If an access generated by the PPE causes an L2 cache miss and it is within the specified address range, it will be replaced by the specified set in the set list. The address range must be within a naturally aligned 2^n size area. The EA range may have a total of four L2 control mechanisms (two for instruction accesses and two for data accesses). A default set list for when none of the EA ranges is applicable can also be set.

Example:

EA range	Set list
0x100000-0x1fffff	0
Default	1-7

If a memory access to a location between 0x100000 and 0x200000 causes a cache miss, the cache replacement will be executed only with Set 0. Therefore, the data in this area is cached using just 64KB of the 512KB L2. Data in other areas will be cached in the 448KB of Sets 1-7.

Example:

EA range	Set list
0x10000-0x1ffff	0
Default	1-7

Here, the memory size of the 0x10000-0x1ffff range is 64KB, and the size of L2 for caching this area is 64KB as well. In this case, cache misses will not occur once data in the range is cached after a cold miss that occurred when data was not cached in L2. The cache line may be invalidated in cache coherence maintenance operations, but it would have the effect of appearing to the processor as though the cache area were locked.

L2 replacement management can be utilized in the following situations.

Interrupt vectors and event handling code can be kept in L2 so that interrupt causes requiring urgent responses can be handled quickly.

In many cases, streaming data is used only once. When all of the streaming data kept in a high-capacity buffer is accessed, all the cached data will be replaced by streaming data that will not be reused. To prevent such a situation, a streaming data buffer can be cached to just a part of L2.

Interrupt Handling

Refer to "PPE User's Manual".

6 SPE

Each SPE is composed of an SPU and an MFC. The SPU is a processor core with a SIMD instruction set that handles 128-bit data. The SPU also has a 256KB local memory (local storage: LS) instead of a cache hierarchy. Programs executed by an SPE and all data accessed by load/store instructions are placed in LS.

Data transfers between LS and external memory are conducted by MFC, a unit separate from the SPU. The MFC is controlled by the SPU via a channel interface and is responsible for carrying out DMA transfers between LS and external devices, as well as synchronization processes between processor modules.

The configuration of the SPE has similarities with existing DSPs for embedding purposes and DMA transfer engines, but also several important differences.

Advanced circuit tuning

The SPE is a full custom design and is based on the assumption that it will be used for a variety of applied purposes as a general-purpose processor. Typical DSPs are designed with an emphasis on embedding purposes (as an IP core) or on customization per purpose. In contrast, the SPU has high capabilities based on its advanced tuning functionality.

Emphasis on code generation by the compiler

SPU instruction sets are defined with an emphasis on orthogonality and do not use special registers. In addition to lightening the programmer's load, the goal is to achieve a stable performance in a wide variety of applications by eschewing case-specific performance tuning and implementing general-purpose functions.

Allowing context switches

SPE contexts are defined, and preemptive context switching is allowed. Processes running on the SPU and MFC can be paused, their states saved in memory and then later restored. Since the MFC can schedule large data transfers, there may be an extremely long wait time if the context is saved only after all active DMA commands are complete. With context switching, such a wait can be avoided by pausing the MFC before all active DMA commands are complete, saving partially processed DMA commands to memory as part of the context, and then restoring them.

Compatibility with PowerPC Architecture

The MFC has an MMU compatible with PowerPC Architecture that is used to convert effective addresses to real addresses when the MFC accesses shared memory. Also, the MFC supports a high-speed synchronization mechanism particular to the SPE, as well as a synchronization mechanism compatible with PowerPC Architecture. In effect, the SPE can carry out parallel processing while sharing its application data structure with a PowerPC Architecture processor core.

SPU Logic Specifications Overview

The SPU is a processor with a newly developed SIMD instruction set. The instruction set is a highly orthogonal RISC instruction set and all instruction words are 32-bit wide, but in contrast to typical 32-bit/64-bit RISC processors, it has 128 128-bit wide registers, and the main instructions process a total of 128 bits of data at once. The main data structures handled by the SPU are as follows.

Data type	Number of elements	Main operation processes
128bit logical	1	Logical operations Shift, rotate Load/store
8bit integer	16	Sum of absolute difference Shuffle byte
16bit integer	8	Addition, subtraction
16bit integer	4	Multiply-add operations (32bit output)
32bit integer	4	Addition, subtraction
32bit floating point	4	Multiply-add operations
64bit floating point	2	Multiply-add operations

SPU load/store instructions generate 32-bit LS addresses using fixed 32-bit slots in the 128-bit data. LS addresses are not converted and are used as they are for memory accesses. Access to areas exceeding the LS size (256KB for the CBE) will be wrapped around. External memory cannot be accessed using load/store instructions. Transactions between the SPU and external devices are carried out via read/write channel instructions.

Channel instructions read or write data to the channel of the number specified in the operand. Every channel is either a read channel or a write channel; there are no channels that can execute both reads and writes. Each channel is handled as a one-way FIFO with a minimum data entry of 1 and has an interface to check the FIFO depth and the number of data kept in the FIFO. Reads from the channel constitute taking data out of the FIFO, and writes to the channel, inserting data into the FIFO.

Attempting to take data out from an empty channel or put in data to a channel with data in all its entries will cause a pipeline stall. Pipeline stalls can be avoided by polling the channel status. Stalling can be an effective way to conserve power, however, since activity in the SPU pipeline is suppressed when stalled.

The number of channels that can be logically defined by SPU instruction sets is 128 (each is either a read or a write channel). The SPE uses 33 of these in communication with the MFC. There are three channel widths that can be logically defined (read/write data units of the FIFO): 32-bit, 64-bit, and 128-bit, but only the 32-bit channel is implemented.

The SPU can also take interrupts from external devices. When an interrupt is taken, the program counter is saved and execution branches to a fixed address (0 address).

General-purpose processors shift to privileged mode when jumping to an interrupt vector, but since the SPU does not differentiate between user mode and privileged mode, interrupts can be thought of as substitutes for polling. Without interrupts, it is necessary to embed polling code in the code to detect the occurrence of events independent of the process currently in progress; by using interrupts, such polling can be avoided.

For SPEs, the SPU takes interrupts only from the MFC. The completion of DMA commands and the success of processor synchronization are examples of interrupt causes.

SPU Performance Characteristics

The overall characteristics of SPU operation pipelines are as follows.

- 2 instructions issued per cycle
- 128-bit data bus
  - 1 128-bit logical operation per cycle
  - 4 parallel single-precision floating-point multiply-add operations per cycle
  - 4 parallel 32-bit ALU operations per cycle
  - 4 parallel 16-bit integer multiply-add operations (32-bit output) per cycle
  - 8 parallel 16-bit ALU operations per cycle
  - 16 parallel application-specific 8-bit integer operations per cycle
  - 2 parallel 64-bit double-precision floating-point multiply-add operations per 7 cycles
- Latency
  - 6 cycles for loads
  - 3 cycles for integer operations
  - 7 cycles for single-precision floating-point multiply-add operations
  - 13 cycles for double-precision floating-point multiply-add operations

The SPU does not implement dynamic scheduling or register renaming or other complex mechanisms that are useful for improving performance. Performance tuning will depend on static instruction scheduling by the programmer or compiler. The 128 registers of the SPU are useful in the static scheduling of code that has instructions with long latencies.

Though branch penalties are large (because the number of pipeline stages is large) the SPU does not have a branch prediction mechanism. Branch penalties are reduced using a combination of the following three methods.

- Instruction scheduling based on static branch prediction
- Elimination of branch instructions by using select-bit instructions as conditional move instructions
- Branch hint instructions

By scheduling appropriate branch hint instructions a certain number of cycles before branch instructions, pipeline stalls due to branch processing can be avoided. Ideally, processing will complete without any pipeline stalls.

The pairs of branch hint instructions and branch instructions can be positioned in a pipeline fashion. For example, Branch Instruction 1 and the corresponding Branch Hint Instruction 1, and Branch Instruction 2 and the corresponding Branch Hint Instruction 2 can be scheduled in the following order: Branch Hint Instruction 1, Branch Hint Instruction 2, Branch 1, and Branch 2. If the scheduling satisfies timing constraints, the penalties of Branch 1 and Branch 2 are both hidden as a result. Branch hint instructions can be used to reduce penalties of multi-directional branches since they use register values to specify branch destinations.

In conclusion, the SPU 1) implements a sufficient number of logical registers for effective static scheduling, and eliminates dynamic scheduling and register renaming and other high-end performance-improvement mechanisms, 2) does not implement branch prediction logic but speculative executions and branch hint instructions to mitigate branch penalties,
3) implements LS and a DMA transfer engine instead of a cache hierarchy so that access to external devices is independent of the processor pipeline core. In such ways, the SPU has simplified the pipeline configuration while maintaining a high performance.

MFC

The MFC governs transactions between the SPE and external devices. MFC functions are as follows.

- Mediates between the SPU and external control commands (run, stop, etc.) via MMIO registers
- Transfers data between the SPU LS and shared memory space (DMA transfers)
- Mediates transactions between the SPU and external devices, handles atomic operations and external events

DMA transfers are executed when the MFC interprets MFC commands. There are two ways to supply the MFC with MFC commands. Devices external to the SPE supply MFC commands by accessing MMIO registers inside the MFC. Inside the SPE, the SPU supplies MFC commands via the channel interface. When scheduled, MFC commands are held in the 16-entry DMA queue.

The MFC interprets MFC commands in the DMA queue in order, and accordingly executes data transfers between LS and shared memory space, or synchronization processes, or memory hierarchy management. Each MFC command has a tag that takes a value from 0 to 31. The completion of an MFC command can be checked by specifying the tag; all MFC commands with the same tag will be checked at this time. The SPU will then be notified of the completion via an SPU tag status update external event.

Data transfer commands transfer data between LS and shared memory space. Get and put commands transfer data between contiguous spaces in LS and contiguous spaces in shared memory space. Get commands transfer data from the effective address space to LS, and put commands transfer data from LS to the effective address space. The LS address (32-bit), effective address (64-bit), and transfer block size are required as parameters. Transfer blocks can be specified in 1-byte, 2-byte, 4-byte, 8-byte, and multiples of 16-byte sizes. Transfer blocks must be aligned to natural data boundaries of the block size unit.

Get list and put list commands transfer data between contiguous spaces in LS and multiple blocks in the effective address space. Two LS addresses (each 32-bit), the DMA list size, and the upper 32 bits of the effective address are required as parameters. One of the LS addresses specifies the data block address, and the other specifies the DMA list address. The DMA list has as many entries as specified by the list size. Each entry has the lower 32 bits of the effective address and the block size. Together with the upper address of the effective address specified by the MFC command, this information specifies one transfer block.

Data is transferred between multiple blocks in the effective address space specified by the DMA list and the area that has as its base address the LS address specified by the DMA command. The get list command copies data of multiple blocks to contiguous spaces in LS. The put list command copies data from contiguous spaces in LS to multiple blocks. The DMA list entry has a stall and notify bit. When this bit is on, transfers using the DMA list will pause at this entry, and a stall and notify external event will occur.

Synchronization process commands are MFC commands that assist synchronization processes between two SPEs or between the PPE and SPE. Examples are PPE-compatible atomic operations and sendsig commands used to notify other processor cores of successful synchronizations via shared memory space. Details regarding synchronization process commands can be found in "Section 7: Processor Synchronization."

Memory hierarchy control commands are commands that control the cache hierarchy (if one exists) between the SPE and a memory device. Prefetch, zero clear, and flush operations are defined, but there is no cache memory hierarchy on the CBE affected by these commands.

Order of Data Transfers

To maintain a highly effective memory system, the CBE implements a memory system that is weakly ordered. In such a memory system, the order in which memory access requests appear in an instruction execution sequence may not match the order in which they are executed.

The PPE checks the real address and prevents data-dependent memory access requests from being executed in an order other than program order so that inconsistencies between the program order and the results will not occur. Unlike the PPE, however, the MFC does not have a logic to check data dependency relationships, so the execution results of DMA commands may not always match the order in which the DMA commands were entered.

Example:

```
get from memory location A to LS location B
put to memory location A from LS location C
```

The order of get and put commands is not guaranteed. Therefore, the value that is copied to LS Location B may possibly be either the initial entry of Memory Location A or the value copied from LS Location C with the put command.

Example:

```
get from memory location A to LS location B
put to memory location C from LS location B
```

If executed in program order, this will be a copy from Memory Location A to Memory Location C. However, since the order of get and put commands is not guaranteed, it is possible for Memory Location C to be written with the old value of LS Location B.

There are a few cases when commands may get out of order, not only during the execution of multiple independent MFC commands, but even during the execution of a single MFC command. One such case is a put command that specifies as its target block an effective address space with a real address page that is mapped more than once. In this case, the update result of the real address page in question is undefined. Another case is with a put list command in which an overlap exists between the memory blocks specified by two different entries of a DMA list.

There are several methods, as follows, to specify the order explicitly. By employing these methods, the order can be effectively controlled.

By **waiting for the completion of an MFC command before scheduling another MFC command**, the order can be maintained. The completion of an MFC command is detected when it can be guaranteed that the command following it will not affect it.

Fences or barriers embedded in commands can be included in data transfer commands to maintain the order. Fences and barriers are tag-dependent and therefore control the execution order only of commands with the same DMA tag number. An MFC command with a fence is executed after all preceding MFC commands.

Example:

```
get from memory location A to LS location B (TAG=0)
putf to memory location C from LS location B (TAG=0)
```

putf is a put command with a fence. Here it is guaranteed that the contents of Memory Location A will be copied to Memory Location C via LS Location B.

Example:

```
get from memory location A to LS location B (TAG=0)
putf to memory location C from LS location B (TAG=0)
put to memory location A from LS location D (TAG=0)
```

Example:

```
get from memory location A to LS location B (TAG=0)
putf to memory location C from LS location B (TAG=0)
put to memory location C from LS location D (TAG=0)
```

MFC commands following an MFC command with a fence may be executed before the command with a fence. Furthermore, MFC commands before and after the command with a fence may be executed in any order. Therefore, in the former of the two examples above, it is possible for the contents of LS Location D to be copied to Memory Location A, and then for that to be copied to Memory Location C via LS Location B. In the latter example, it is possible for the contents of LS Location D to be copied to Memory Location C before the contents of LS Location B are copied to Memory Location C.

By using an MFC command with a barrier, MFC commands preceding an MFC command with a barrier will be executed both before the command with a barrier and the MFC commands following the command with a barrier.

Example:

```
get from memory location A to LS location B (TAG=0)
putb to memory location C from LS location B (TAG=0)
put to memory location A from LS location D (TAG=0)
```

putb is a put command with a barrier. In this case, the copy from LS Location D to Memory Location A will never be processed before the copy from Memory Location A to LS Location B. However, MFC commands following an MFC command with a barrier may be executed before the command with a barrier. Therefore

Example:

```
get from memory location A to LS location B (TAG=0)
putb to memory location C from LS location B (TAG=0)
put to memory location C from LS location D (TAG=0)
```

In this example, it is still possible for the copy from LS Location D to Memory Location C to be executed before the copy from LS Location B.

Atomic commands (getllar, putllc, putlluc) do not have tags and cannot be ordered by fences or barriers. To schedule an atomic command after another command, a tag completion wait must be used to check explicitly that the preceding command has completed, and then input the atomic command to the DMA queue. To schedule a different command after an atomic command, it is necessary to check the completion of the atomic command using the Atomic Command Status channel and then input the command in the queue.

The queuing-type atomic command putqlluc has a tag and follows the ordering rules defined for its tag group. Also, since putqlluc is handled as a command with a fence, its ordering among other MFC commands is guaranteed.

Barrier commands are independent MFC commands that maintain the order of commands. Barrier commands guarantee that MFC commands preceding the barrier will be executed before MFC commands following the barrier, regardless of tag values.

Example:

```
get from memory location A to LS location B (TAG=0)
barrier
put to memory location C from LS location B (TAG=1)
```

Though get and put have different tag values, inserting a barrier between them will maintain the order.

MFC sync and **MFC eieio commands** can also be used to maintain the order. Though these commands appear to the SPE to have the same function as a barrier command, their ordering is dependent on tags. The differences between these commands and fences/barriers will be apparent when synchronizing

between processors in a multiprocessor system or between processors and devices. When ordering MFC commands inside just one SPE, there is no reason to use these commands that have a large load on the memory system.

Transfer Class ID

MFC commands transferring blocks larger than 128 bytes are broken down into memory access requests of 128-byte unit blocks. For example, get or put of a 4KB block aligned on a 128-byte boundary will be split into 32 separate memory accesses.

These memory access requests are held in the 16-entry SBI queue. In this way, an MFC command can send a maximum of 16 simultaneous memory access requests to the memory system. Entries in the SBI queue are released when data transfers to or from LS complete. Therefore, a request remains in the SBI queue until the command phase of the memory access completes and data is taken out from LS (for write accesses) or data read is written to LS (for read accesses).

When all entries in the SBI queue are occupied, no memory accesses can be issued, so the number of entries may affect the effective value of bandwidths. For example, a read-memory bandwidth with a 150ns latency will be restricted by the number of SBI queue entries and will therefore have a maximum rate of approximately 14GB/s (1/150*128*16).

If executable get and put commands exist in the DMA queue at the same time, MFC will alternate read and write accesses to send to the SBI queue. At this time, if the latency of read accesses imposes constraints upon the bandwidth, the latency of write accesses also becomes restricted.

Furthermore, if the get and put commands access different memory devices and one of the memory bandwidths is small, access to that device takes over the queue entries and affects the effective bandwidth of the other process.

Example:

```
put to XDR DRAM location C from LS location D size 16KB
get from IO location E to LS location F size 16KB
```

In the above example, a get from I/O device memory and a put to XDR DRAM are executed simultaneously. If the latency of the I/O device memory is large (or the bandwidth is small), the bandwidth of the put will decrease as a result.

To solve this problem, the Transfer Class ID field of MFC commands may be used to differentiate requests made to different memory devices. By restricting the number of usable SBI queue entries per transfer class, interference between transfer classes may be eliminated.

For example, suppose Transfer Class 0 requests are allowed a maximum of 10 entries and Transfer Class 1 requests are allowed a maximum of 6. No interference in the SBI queue resources will occur between the MFC commands of the two transfer classes.

In CBE chip implementation, a maximum of three transfer classes may be used. MMIO registers of the MFC can be set to adjust the following properties of each transfer class.

- The number of SBI queue entries that MFC commands of that transfer class can use
- Issue slot settings

Issue slots can be set so that reads and writes of that class are issued alternately, or in order from the first MFC command (in the latter case, consecutive reads and writes are issued).

External Events

Non-synchronized events detected by the MFC are notified to the SPU using an external event handling mechanism. The external event handling mechanism's interface to the SPU uses three channels, Channel 0 to 2. The SPU checks the status of external events via this channel interface and resets the event status by acknowledging certain events. Events that do not need to be detected can be masked. If SPU interrupts are enabled, an SPU interrupt will occur when an event occurs. The following events are examples of external events.

Events generated by a synchronizing mechanism between processors

- multi source sync
- privileged attention
- lock-line reservation lost
- signal notification1
- signal notification2
- SPU Outbound Mailbox available
- SPU Outbound Interrupt Mailbox available
- SPU Inbound Mailbox

Timer events

- decrementer

Events reflecting the status of DMA transfers

- DMA queue
- DMA list command stall and notify
- SPU tag status update

See "Cell Broadband Engine™ Architecture" for details regarding these events.

7 Processor Synchronization

To make effective use of multiple processor modules (PPE and SPE) and attain a high system performance, it is necessary to utilize appropriate processor synchronization and data exchange mechanisms for efficient parallel processing. For synchronization between processors, it is also necessary to consider the ordering of memory accesses. See below for details.

Shared Memory Access

Internal CBE modules and external devices connected via IOIF share a physical address space. Data exchange and synchronization between multiple master devices are carried out via a shared memory space.

Because EIB controls consistency of the cache using the snoop protocol, when masters access data cached in the cache memory, the consistency of data values is maintained without any need for extra programming. EIB also supports interventions. When an SPE or an I/O device attempts to read data cached in PPE L2, data is transferred directly from L2 to the master device that attempted the read access, instead of being read from memory. Direct transfers from L2 are advantageous performance-wise because they do not use the main memory bandwidth and have a short latency.

PPE Synchronization Operations

See "Book II: PowerPC Virtual Environment Architecture, Version 2.02" for details regarding synchronization operations in PowerPC Architecture.

PPE-Compatible Synchronization Mechanisms of the SPE

The SPE can execute a series of synchronization operations that are compatible with standard PowerPC synchronization mechanisms. SPEs can work within the same system as a PowerPC processor by using these operations.

Data in shared memory can be updated atomically using two MFC commands, getllar and putllc. getllar reads one 128-byte block of effective address space out to LS and reserves it. The 128-byte block must be aligned naturally. When the reserved block is updated by another master device, the reservation is cancelled. putllc is the command that writes one 128-byte block from LS to the effective address space, but before writing, it checks the reservation state of the block. If the reservation has been cancelled, putllc will not execute the write. The success (or failure) of putllc can be checked using the Atomic Command Status channel.

This is the same as the PPE carrying out lwaux and stwcx instructions in conjunction, but the size of the reserved data here is large (128 bytes). Although this method does not guarantee PPE compatibility, multiple synchronization variables can be placed in the 128-byte block for processing.

Atomic commands (putllar, putllc, putlluc) are different from typical MFC commands in that they do not have tags. They are not affected by fences/barriers in commands, MFC sync, MFC eieio, nor barrier commands. The completion of an atomic command is detected using the Atomic Command Status channel.

Reservation of memory blocks using getllar may be used in combination with putllc to carry out atomic updates, but it may also be used to make memory block polling more efficient. When the reservation of a block reserved with getllar is lost, a lock-line reservation lost external event occurs. A lock-line reservation lost external event here means that either of the following two situations occurred: the memory block was updated by another master device, or the block reservation could not be maintained because an SPE context switch (for example) occurred. Therefore, the external memory accesses needed for memory block polling can be eliminated by following the procedure below.

Example:

(1) getllar memory location A
(2) if (the value of memory location A is the specified value) then the next process
(3) Wait until a lock-line reservation lost external event occurs
(4) Return to (1)

MFC eieio and MFC sync are MFC commands that are used to control the order of memory accesses. They are able to support functions of the same name (eieio and sync) defined in PowerPC ISA. They will be described in the "MFC Ordering Rules" section below.

High-Speed Synchronization Mechanisms of the SPE

The MFC has two 32-bit signal notification registers (SNR) in order to carry out rapid synchronization between SPEs or between an SPE and an external device. An SNR is mapped to shared memory space and can be updated by the PPE, SPE, and other master devices. The SPU inside an SPE can check the SNR state using channel interfaces and external event mechanisms.

There are two modes for updating SNR. In overwrite mode, 32-bit data written to SNR is substituted for SNR. In or mode, a logical OR operation is run on the 32-bit data written to SNR and the old SNR values to obtain the new SNR values. By using or mode, each of the 32 bits can be made to correspond with an event. So that all events are accounted for, the SNR channel interface has the function to execute status checks and resets of observed events atomically.

Synchronization processes can also be carried out using the synchronization variables in LS. However, channel interface and external event mechanism services such as stall controls and SPU interrupts cannot be used; synchronization events will be detected with polling using load instructions.

sendsig MFC commands are provided for updating SNRs and synchronization variables in LS. In the current implementation, there is no difference functionally from using put commands to update 32-bit words, but the use of sendsig commands is recommended for sending synchronization events from one SPE to another.

Mailbox Communication

The SPE has a mailbox interface for communication between the PPE and SPE. The mailbox is a 32-bit wide FIFO and has an interface for SPE-to-PPE communication and another for PPE-to-SPE communication.

With the SPU Outbound Mailbox and the SPU Outbound Interrupt Mailbox, the SPU can insert 32-bit data with a channel write, and external devices (including the PPU) can take out that data with an MMIO register read. These two mailboxes hold just one 32-bit data each. The difference between these two mailboxes is that the SPU Outbound Interrupt Mailbox has the function to cause an interrupt to the PPU when data is inserted by an SPU.

The SPE Inbound Mailbox allows the PPE to insert 32-bit data with an MMIO register write and an SPU to take out that data with a channel read. The SPE Inbound Mailbox has four levels of FIFO and can hold a maximum of four 32-bit data provided by the PPE. Besides using the channel interface, the SPU can conduct status checks using SPU Inbound Mailbox external events.

Appendix A: Multiprocessor Ordering Rules

The ordering rules for shared memory access as guaranteed in CBE implementation are described below. The CBE memory system is weakly ordered. In memory space without strict ordering constraints, it is possible for multiple memory accesses executed by one master to be observed by another master in an order different from program order.

Example:

```
PPE                          SPE0
store to memory location A   get from memory location B
store to memory location B   get from memory location A
```

In this example, it is possible for the old value of A to be read even after SPE0 reads the value of B updated by the PPE. This characteristic is useful for improving the effective performance by increasing the flexibility of the shared memory system, but must be taken carefully into account for synchronization processes between processors.

It is not only possible in a weakly ordered memory system for the program order and the order observed by an outside party to be different (as in the above example) but also for different results to be observed by different masters.

Example:

```
PPE                          SPE0
store to memory location A   get from memory location B
store to memory location B   get from memory location A

                             SPE1
                             get from memory location B
                             get from memory location A
```

In this example, it is possible that SPE0's memory access results and SPE1's memory access results will be different. To guarantee the order in such a case, the order of memory accesses must be enforced using appropriate synchronization operations.

Example:

```
PPE                          SPE0
store to location A          get from location B
sync                         getf from location A
store to location B
```

In this example, it is guaranteed that the updated value of A will be read by SPE0 if SPE0 reads the updated value of B.

The ordering rules defined in PowerPC Architecture require that data transfers adhere to the specified order; at the same time, they allow for the observed memory access order to be inconsistent among different processors. Therefore, in the following example sequence, PowerPC Architecture allows the possibility that both of the gets (by SPE0 from Location B and by SPE1 from Location A) may result in returning non-updated values. Also, it is not possible to determine whether the update of Location A by SPE0 or the update of Location B by SPE1 was executed first, since the results observed by SPE0 and by SPE1 are inconsistent.

Example:

```
SPE0                 SPE1
put to location A     put to location B
MFCsync              MFCsync
get from location B   get from location A
```

Some of the ordering rule definitions in this chapter are stronger than the descriptions in the "Cell Broadband Engine™ Architecture" document. These definitions are based on the current implementation of the Cell Broadband Engine™, and they are guaranteed to be correct with future chip implementations intended for systems like game consoles, on which software compatibility is very important.

MFC Ordering Rules

MFC ordering methods that affect multiprocessor synchronization are the fence and barrier attributes of commands, completion detections (tag completion waits and atomic command completion waits), independent barrier commands, MFC eieio, and MFC sync. The effect on the relationship between program order and the execution order of memory accesses is as explained in "Order of Data Transfers" in Chapter 6. Additionally, when they are used in a multiprocessor environment where memory access orders are observed by other processors, there will be differences dependent on the memory attributes.

Event ordering enforced by these synchronization mechanisms are consistent with the ordering rules defined by PowerPC Architecture. Ordering using these synchronization mechanisms, direct data level ordering (like interrupts from SPE to PPU and polling by PPU of data updates by SPE), instruction ordering of SPU (see next section), and synchronization instructions of PPU like sync and lwsync are maintained to be correct. Tag completion waiting guarantees ordering between the results of MFC commands and the results of subsequent SPU actions as observed by external entities.

* See " Book II: PowerPC Virtual Environment Architecture, Version 2.02" for details regarding ordering rules in PowerPC Architecture.

- When get/put/sendsig commands are executed, fences, barriers, and tag completion waits will order accesses to memory that have the Memory Coherence Required attribute but neither the Caching Inhibited nor the Write Through Required attribute.
- When get/put/sendsig commands are executed, fences, barriers, and tag completion waits will order accesses to normal memory (LS and XDR DRAM) that have the Caching Inhibited attribute.
- When get/put/sendsig commands are executed, fences, barriers, and tag completion waits will order get, put, and sendsig commands to the same SPE LS and MMIO register independent of the memory attributes.
- MFC sync and MFC eieio will order get, put, and sendsig commands independent of the memory attributes.
- Put commands with the same tag number targeting a page with both the Guarded attribute and the Caching Inhibited attribute are ordered. Also, the order of multiple access requests generated by a single put or put list command is maintained.

Ordering Rules for SPE Internal Accesses

In addition to being mapped to shared memory, SPE LS is also accessible from within the SPE, by SPU and MFC. Internal accesses are carried out on a different path from accesses via shared memory space. For this reason, ordering rules for internal accesses and LS accesses via shared memory space (accesses from outside the SPE) are different from normal ordering rules.

The instruction execution sequence of the SPU is ordered by two instructions, sync and dsync. dsync guarantees that LS updates by store instructions can be observed by accesses by external devices. sync is used for all other cases when pipeline synchronization is necessary. In the current implementation of SPU, neither sync instruction is necessary to define relationships with external accesses.

If multiple writes to one SPE LS are ordered and are carried out by one master device, then these writes are also ordered for internal accesses. Here it is necessary to note that accesses by multiple master devices are not necessarily ordered for internal accesses even if they are ordered in the shared memory system.

One direct consequence of this characteristic is that writes to the SPE from external devices are not cumulative when observed by internal accesses. Cumulative ordering means that if the order Event A - Event B and the order Event B - Event C are both true, then the order Event A - Event C is also true. It is

necessary to consider this characteristic when carrying out synchronization processes between SPEs using SNR or LS synchronization variables.

Example:

```
SPE0                  SPE1                  SPE2
put to SPE2 LS location A
putf to SPE1 SNR
                      check SNR of event from SPE0
                      put to SPE2 LS location B
                      putf to SPE2 SNR
                                            check SNR of event from SPE1
                                            load LS location B
                                            load LS location A
```

In this example, it is guaranteed that the SPE2 load from LS Location B will read the value updated by SPE1, but it is possible that the load from LS Location A will read the old value before being updated by SPE0. Note that in this example, load executed by SPE2 is the load of the SPU load instruction.

In such cases, synchronization problems can be resolved in one of the following two ways.

One way is to use MFC sync or MFC eieio commands for ordering. If these commands are used to order external accesses, there will be no inconsistency with the ordering of internal accesses. The other way is to use mssync, the multi-source synchronization facility in the MFC. The multi-source synchronization facility can be used by the SPU via channels and by external devices via MMIO registers, and by following the two steps below, it can solve the inconsistency between the external ordering and the ordering as observed in the SPE internally.

```
Step 1: Boot mssync (SPU: write channel, external devices: MMIO write)
Step 2: Check completion of mssync (SPU: read channel, external devices: MMIO
read)
```

Example:

```
SPE0                  SPE1                  SPE2
put to SPE2 LS location A
putf to SPE1 SNR
                      check SNR of event from SPE0
                      put to SPE2 LS location B
                      putf to SPE2 SNR
                                            check SNR of event from SPE1
                                            load LS location B
                                            mssync
                                            load LS location A
```

Example:

```
SPE0                  SPE1                  SPE2
put to SPE2 LS location A
MFC sync
put to SPE1 SNR
                      check SNR of event from SPE0
                      put to SPE2 LS location B
                      putf to SPE2 SNR
                                            check SNR of event from SPE1
                                            load LS location B
                                            load LS location A
```

In both of the above sequences, it is guaranteed that if SPE2 reads the new value of Location B (updated by SPE1), then it also reads the new value of Location A (updated by SPE0).