Chapter 2
The Power Processing Element (PPE)

Abstract In this chapter we discuss the design, implementation and functionality of the PowerPC Processing Element (PPE), which serves as the control plane of the Cell Broadband Engine; the OS of the Cell runs on the PPE. The PPE contains a 64-bit, dual-thread PowerPC Architecture RISC core and supports a PowerPC virtual-memory subsystem. It has a 32 KB L1 I-cache, as well as a 32 KB D-cache, along with a 512 KB L2 unified cache. We analyze the PPE in detail, as not only does it provide significant facilities for controlling the SPEs, the PPE in Cell can run existing PowerPC 970MP and 970FX compiled applications. Since PowerPC is often used as a control processor in embedded systems, we use the PPE in Cell to understand PowerPC instruction set in detail. The PPE in Cell includes the vector/SIMD multimedia extensions (AltiVec).

2.1 PPE: the Control Plane of the CBEA

Although the Cell Broadband Engine derives much of its touted Gigaflops from the 8 (limited to 6 in the PS3) Synergistic Processor Elements (SPEs), the Power Processing Element (PPE) is the *conductor*, presiding over the computing orchestra. In this chapter we shall focus solely on the PPE, as we examine its design and architecture with the intent of understanding its computational sweet-spot. Thereafter, in this book, we shall mostly concentrate on writing efficient SPE code by optimizing for SIMD, dual-issue and DMA. The role of the PPE shall become like the benevolent elder sibling or parent, watching out, doing its job, and mostly staying out-of-the-way of the blazing SPEs.

As discussed above the PPE performs the control plane functions that typically require a more general-purpose or traditional computing environment which is needed for Operating Systems, and system software. The PPE in CBEA compliant architectures is based on Version 2.02 of the PowerPC Architecture, which offers many of the features required for the application spaces targeted by the CBEA.

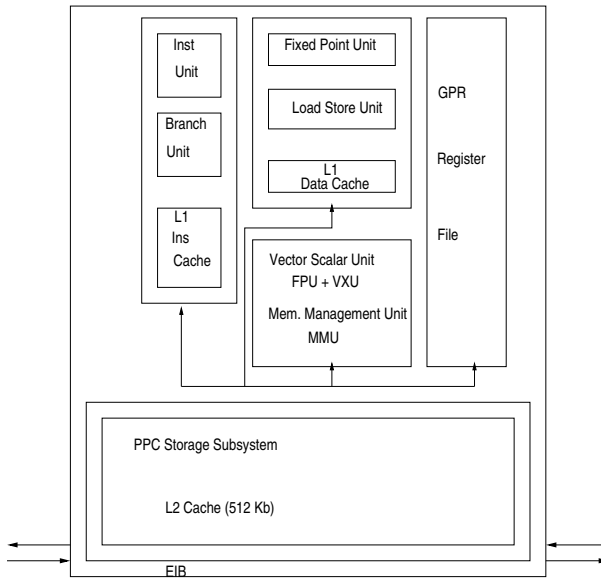The PPE was based on the PowerPC Architecture for four reasons [27]:

Fig. 2.1 A detailed view of the PPE

1. the PowerPC Architecture is a mature architecture that is applicable to a wide variety of platforms,
2. it supports multiple simultaneous operating environments through logical partitioning,
3. it contains proven microarchitectures that meet the frequency and power challenges of the targeted market segment,
4. use of the PowerPC Architecture leverages IBM's investment in the PowerPC ecosystem.

The PPE complies with the 64-bit implementation of the PowerPC Architecture [13, 14]. It is a dual-threaded processing core that includes an integer unit, a floating-point (FP) unit, a vector multimedia extensions (VMX) unit, and a memory management unit (MMU). See Figure 2.1 for a schematic. The PPE instruction cache (I-cache) is 32 KB and the data cache is 32 KB. In addition, an unified 512-KB Level 2 (L2) cache is included in the PPE, which also has a memory flow controller (MFC) that enables it to perform direct memory access (DMA) transfers to and from main memory, SPEs, or I/O. The MFC also provides memory-mapped I/O (MMIO) transfers to on-chip and off-chip devices. Communication to and from SPEs can be performed via DMA and/or mailboxes. Though based on the PowerPC Architecture, the PPE has made some architecture trade-offs. For example, it is pipelined extensively to allow it to operate at frequencies higher than 3 GHz. To reduce silicon area, program execution is performed in order. It supports allocation management which allows portions of a resource time to be scheduled for a specific resource allocation group. This simplifies real-time application programming. All the caches are cov-

ered by an allocation management scheme. The PowerPC Architecture hypervisor extension is also included in the design to allow multiple operating systems to run simultaneously via thread management.

As shown in Figure 2.1, the PPE consists of three main units: the instruction unit (IU), the execution unit (XU), and the vector/scalar execution unit (VSU), which contains the VMX and floating-point unit (FPU). The IU contains the Level 1 (L1) instruction cache (ICache), branch prediction hardware, instruction buffers, and dependency checking logic. The main division between the IU and the rest of the system is at the instruction issue 3 (IS3) stage, which is the main stall point for the PPE. The XU contains the integer execution units (FXUs) and the loadstore unit (LSU). The VSU contains all of the execution resources for FP and VMX instructions, as well as separate VMX and FP instruction queues in order to increase overall processor throughput.

Although the PPE is considered an in-order machine, several mechanisms allow it to achieve some of the benefits of out-of-order execution, without the associated complexity of instruction or memory access reordering hardware. First the processor can make forward progress on a thread even when a load from that thread misses the cache. The processor continues to execute past the load miss, stopping only when there is an instruction that is actually dependent on the load. This allows the processor to send up to eight requests to the L2 cache without stopping. This can be a great benefit to FP and SIMD code, since these typically have a very high data cache miss rate, and it is often easy to identify independent loads. In addition to allowing loads to be performed out of order, the PPE uses "delayed execution pipelines" to achieve some of the benefits of out-of-order execution. Delayed execution pipelines allow instructions that normally would cause a stall at the issue stage to move to a special "delay pipe" to be executed later at a specific point.

2.2 PPE Problem State Registers

The PPE contains the following types of registers (cf. [17]):

1. General-Purpose Registers (GPRs) – the 32 GPRs are 64 bits wide,
2. Floating-Point Registers (FPRs) – the 32 FPRs are also 64 bits wide, with single-precision results maintained internally as double-precision data in the IEEE 754 format,
3. Link Register (LR) – the 64-bit LR can be used to hold the effective address of a branch target,
4. Count Register (CTR) – the 64-bit CTR can be used to hold either a loop counter or the effective address of a branch target,
5. Fixed-Point Exception Register (XER) – the 64-bit XER contains the carry and overflow bits,
6. Condition Register (CR) – conditional comparisons are performed by first setting a condition code in the 32-bit CR with a compare instruction,
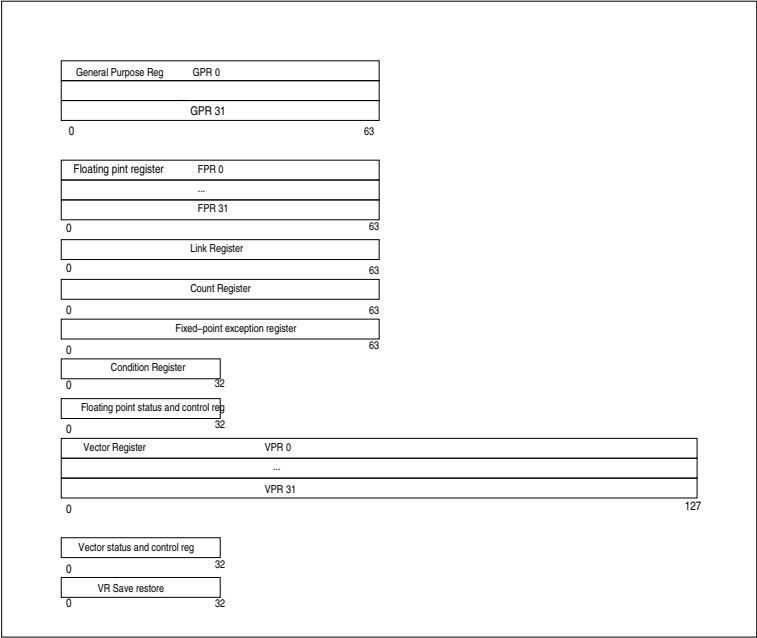
Fig. 2.2 Problem state registers of the PPE.

7. Floating-Point Status and Control Register (FPSCR) – 32-bits, and updated after every floating-point operation by the PPE,
8. Vector Registers (VRs) – 32 128-bit-wide VRs (useful in SIMD operations),
9. Vector Status and Control Register (VSCR) – 32-bit, similar to FPSCR above,
10. Vector Save Register (VRSAVE) – 32-bit, privileged mode only register.

The vector/SIMD multimedia extension defined by the CBEA is very similar to the PowerPC 970 (the major difference being the support for rounding-mode). For compatibility with SPU applications, the vector/SIMD multimedia extension unit in the PPE supports the rounding modes defined by the SPU instruction set architecture. The current PowerPC Architecture supports the base 4-KB page plus one additional large page to be used concurrently. The large-page size is implementation dependent. In the CBEA, many types of data structures are located in main storage, e.g., MMIO registers for the SPEs, local storage aliases, streaming data, and video buffers. If a large-page size of 64 KB is selected, the number of translations needed for MMIO registers and local storage aliases is lower than that for the base 4-KB page size. A large-page size of 1 MB or 16 MB reduces the number of translations required for the streaming and video buffers, but it is too large for mapping the MMIO registers and local storage aliases. To improve the efficiency of the TLBs, the CBEA augments the PowerPC Architecture by providing support for multiple concurrent large-page sizes. The memory management units (MMUs)

Fig. 2.3 Memory hierarchy in the Cell Broadband Engine.
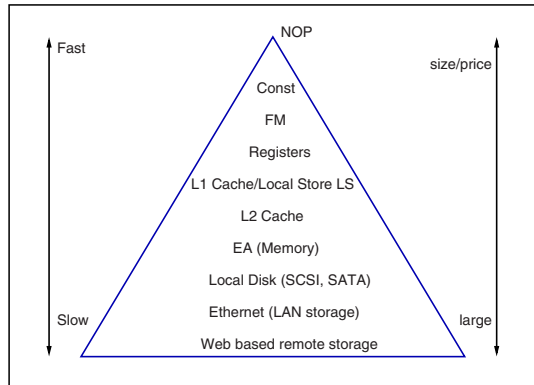
in the SPE also support the multiple concurrent large-pages extension. We discuss using GNU/Linux commands to enable large page-tables in a later chapter.

2.3 Memory arrays in the CBEA

We first consider the memory hierarchy present in the Cell architecture as shown in Figure 2.3. As we go from bottom to top, the size of the memory decreases, but its price per bit, and speed increase. At the bottom most we have remote storage, perhaps distributed across the Internet. Next up, local area network storage devices and file servers, followed by locally attached disk. The access speeds of these storage devices have almost an order of magnitude difference, and their capacities are also commiserate. Beyond disk storage, we have solid state memory inside the chip itself. These are referred to as SRAM arrays in the next section. They are divided into the Effective Address Memory (512 MB on the PS3), the L2 cache (512 kb), the L1 data cache (32 kb), the L1 instruction cache (32 kb), various queues and cache tables, and the SPU local store of 256 kb.

In the PowerPC and SPU architecture computation is performed between register arguments, and thus registers are also considered as memories, and they are near the top of the pyramid in performance. Only forwarding macro output, which is the result of the previous computation already stored in the arithmetic unit, and immediate operands can be considered faster than register access, but then you are already at the microword level, since all three access take a logical clock cycle to complete. At the top of the pyramid for consistency, is a NOP, meaning, no computation implies no memory access. Many-a-times not doing the computation which requires memory is faster than the fastest cache access.

As mentioned above in the memory hierarchy, the Cell architecture has a number of memories distributed all over the chip. According to one count, the Cell processor

contains around 270 arrays. Performance, power, and area constraints drove different designs for the memory arrays. The PPE and SPE contain the largest SRAM devices.

2.3.1 PPE Caches

The PPE has two 32-KB L1 SRAM macros that are organized as two-way associative memory. Interleaving is used with the L1 SRAM macros to improve parity-checking performance. The read latency of the L1 SRAM is three cycles: The first cycle is used by the sum address decoder (SAD) for address generation. The second cycle is used to access the array, and the third cycle is used for parity checking, data formatting, and way select. An L1 SRAM macro is internally constructed as 512 wordlines that access 32 bytes (with parity per byte) per way. Writing of the L1 is performed one cache line at a time and can be 64 bytes or 32 bytes. The writing of one 64-byte-wide cache line addresses two consecutive wordlines. Data that is 16 bytes wide can be read with parity checking on each access. The L1 caches are clocked at full clock rate of the core clock grid.

The L2 cache is a 512-KB eight-way associative cache. The L2 cache is inclusive of the L1 D-cache, but not the L1 I-cache. It supports write-back (copy-back), and allocation on store miss. The cache is constructed of four 1,024   8   140 SRAM macros. Unlike the L1 caches, the L2 cache is clocked at one-half of the clock rate of the core clock grid. Power conservation is achieved by decoding the way selected prior to accessing the array and then activating one-eighth of the L2 macro. The L2 cache has one read-port, and one write-port (but only one read or write per cycle). The L2 array can be accessed as a 140-bit or a 280-bit read. Writes to the array are pipelined and completed in two cycles. Read operations complete in three cycles for the first 140 bits of data. For 280 bits of data, the second 140 bits of data can be completed in the fourth cycle. Writes and reads can be interleaved to provide continuous data transfers to and from the L2 array. It supports ECC on data, parity on directory tags, and global and dynamic power management.

2.3.2 SPU Local Store

The LS array is a 256-KB SRAM array that consumes one-third of the total SPE area. The LS macro consists of a sum address decoder, four 64-KB memory arrays, write accumulation buffers, and read accumulation buffers. The LS completes writes in four cycles and reads in six cycles. The first cycle is used by the SAD to add operands. The second cycle is used to distribute the pre-decoded indices to the four 64-KB subarrays. During the third cycle, decoding of the address is completed and the wordline is selected. The addressed subarray is addressed in the fourth cycle. For reads, the sense amplifier senses the bitline differential signal and holds the value

Fig. 2.4 SPE LS 6R-4W
timing cycle.
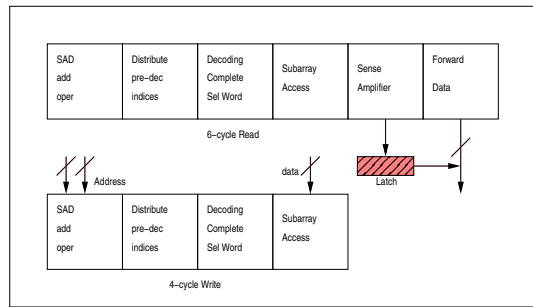
until it is captured in the read latch. In the sixth cycle, the data is forwarded to the
executions units. The LS runs at full core clock frequency. A rough sketch of the
timing is shown in Figure 2.4.

2.4 Viewing the PPE as a dual-core processor: EMT specifics

The following architected registers are duplicated for multithreading and used by
software running in any privilege state (including problem-state):

1. General-Purpose Registers (GPRs) (32 entries per thread)
2. Floating-Point Unit Registers (FPRs) (32 entries per thread)
3. Vector Registers (VRs) (32 entries per thread)
4. Condition Register (CR)
5. Count Register (CTR)
6. Link Register (LR)
7. Fixed-Point Exception Register (XER)
8. Floating-Point Status and Control Register (FPSCR)
9. Vector Status and Control Register (VSCR)
10. Decrementer (DEC)

Thread control registers which are duplicated or thread dependent

Each thread is viewed as an independent processor, complete with separate ex-
ceptions and interrupt handling. The threads can generate exceptions simultane-
ously, and the PPE supports concurrent handling of interrupts on both threads by
duplicating some registers defined by the PowerPC Architecture.

The following registers associated with exceptions and interrupt handling are
duplicated or are thread-dependent:

1. Machine State Register (MSR)
2. Machine Status Save/Restore Registers (SRR0 and SRR1)
3. Hypervisor Machine Status Save/Restore Registers (HSRR0 and HSRR1)

 4. Floating-Point Status and Control Register (FPSCR)
 5. Data Storage Interrupt Status Register (DSISR)
 6. Decrementer (DEC)
 7. Logical Partition Control Register (LPCR)
 8. Data Address Register (DAR)
 9. Data Address Breakpoint Register (DABR and DABRX)
10. Address Compare Control Register (ACCR)
11. Thread Status Register Local (TSRL)
12. Thread Status Register Remote (TSRR)

In addition, the following thread-independent registers also are associated with exceptions and interrupt handling on both threads:

1. Hypervisor Decrementer (HDEC)
2. Control Register (CTRL)
3. Hardware Implementation Dependent Registers 0 and 1 (HID0 and HID1)
4. Thread Switch Control Register (TSCR)
5. Thread Switch Time-Out Register (TTR)

The following sections describe processor registers that play a central role in controlling and monitoring multithreading activity. The tables that describe register fields have been edited for brevity and clarity (for example, the reserved fields and fields not relevant to multithreading are not described). For complete register descriptions, see the Cell Broadband Engine Registers Specification [20].

Duplicated resources The following arrays, queues, and structures are fully shared between threads running in any privilege state:

1. L1 instruction cache (ICache), L1 data cache (DCache), and L2 cache
2. Instruction and data effective-to-real-address translation tables (I-ERAT and D-ERAT)
3. I-ERAT and D-ERAT miss queues
4. Translation lookaside buffer (TLB)
5. Load miss queue and store queue
6. Microcode engine
7. Instruction fetch control
8. All execution units:

   a. Branch (BRU)
   b. Fixed-point integer unit (FXU)
   c. Load and store unit (LSU)
   d. Floating-point unit (FPU)
   e. Vector media extension unit (VXU)

   The following arrays and queues are duplicated for each thread.

1. Segment lookaside buffer (SLB)
2. Branch history table (BHT), with global branch history
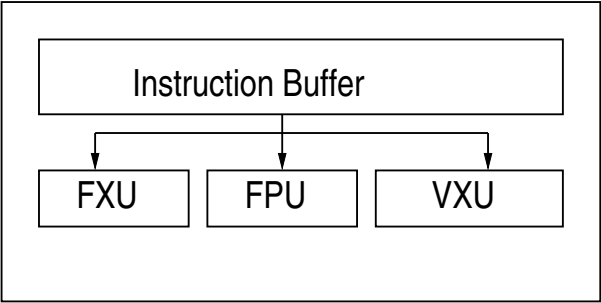3. Instruction buffer (IBuf) queue
4. Link stack queue

Fig. 2.5 Superscalar instruction execution.

Duplicating the instruction buffer allows each thread to dispatch regardless of any dispatch stall in the other thread. Duplicating the SLB is convenient for the implementation because of the nature of the PowerPC Architecture instructions that access it and because it is a relatively small array. The instruction-fetch control is shared by both threads because the instruction cache has only one read port and so fetching must alternate between threads every cycle. Each thread maintains its own BHT and global branch history (GBH) to allow independent and simultaneous branch prediction.

2.4.1 PPU Intrinsics for managing thread priorities and delays

Table 2.1 Differences between PPE and SPE from a programming point of view.

PPU Intrinsic	Description
__cctph	Set current thread priority to high
__cctpm	Set current thread priority to medium
__cctpl	Set current thread priority to low
__db[x]cyc	$x \in 8, 10, 12, 16$ delay current thread for x cycles at dispatch

If the current thread is delayed at dispatch using one of the functions (__db8cyc), whether any other thread will get executed depends on the allocation scheme, and relative thread priorities.

2.5 PowerPC Instruction Set

All PowerPC instructions are 4 bytes long and aligned on word (4-byte) boundaries. Most instructions can have up to three operands. Most computational instructions specify two source operands and one destination operand. Signed integers are represented in twos-complement form. The instructions include the following types:

1. Load and Store Instructions: these include fixed-point and floating-point load and store instructions, with byte-reverse, multiple, and string options for the fixed-point loads and stores. The fixed-point loads and stores support byte, halfword, word, and doubleword operand accesses between storage and the 32 general-purpose registers (GPRs). The floatingpoint loads and stores support word and doubleword operand accesses between storage and the 32 floating-point registers (FPRs). The byte-reverse forms have the effect of loading and storing data in little-endian order, although the CBE processor does not otherwise support little-endian order.
2. Fixed-Point Instructions: these include arithmetic, compare, logical, and rotate/shift instructions. They operate on byte, halfword, word, and doubleword operands.
3. Floating-Point Instructions: these include floating-point arithmetic, multiply-add, compare, and move instructions, as well as instructions that affect the Floating-Point Status and Control Register (FPSCR). Floating-point instructions operate on single-precision and doubleprecision floating-point operands.
4. Memory Synchronization Instructions
5. Flow Control Instructions
6. Processor Control Instructions
7. Memory and Cache Control Instructions

2.6 SPE Memory Management and EA Aliasing

All information in main storage is addressable by EAs generated by programs running on the PPE, SPEs, and I/O devices. An SPE program accesses main storage by issuing a DMA command, with the appropriate EA and LS addresses. The EA part of the DMA-transfer addresspair references main storage. Each SPEs MFC has two command queues with associated control and status registers. One queuethe MFC synergistic processor unit (SPU) command queuecan only be used by its associated SPE. The other queuethe MFC proxy command queuecan be mapped to the EA address space so that the PPE and other SPEs and devices can initiate DMA operations involving the LS of the associated SPE.

When virtual addressing is enabled by privileged software on the PPE, the MFC of an SPE uses its synergistic memory management (SMM) unit to translate EAs from an SPE program into RAs for access to main storage. Privileged software on

the PPE can alias an SPEs LS address space to the main-storage EA space. This aliasing allows the PPE, other SPEs, and external I/O devices to physically address an LS through a translation method supported by the PPE or SPE. SPE-to-SPE DMA often requires knowledge of SPE LS effective-address. Virtual-address translation is enabled in an SPE by setting the Relocate (R) bit in the MFC State Register. This is done by setting the map problem set area to 1 in using the SPE runtime library API.

2.7 Power Processor Element (PPE) Cache Utilization

Since in this book we don't use self-modifying code, we have no need to discuss the sync operations which are needed when instruction streams are modified by the program. But data-prefetching and *touch* operations on pre-loaded data blocks in the L2 cache can be used to improve performance, especially when streaming data. The following functions are provided as PPE intrinsics for this purpose:

Table 2.2 PPE intrinsics for data prefetching and clear.

Intrinsic	Name	Description
__sync	Sync	-
__lwsync	Light-weight sync	-
__dcbt(**void**∗)	Data cache block touch	data-prefetch hint
__dcbtst(**void**∗)	Data cache block touch for store	data-write hint
__dcbst	Data cache block store	Write-back cache data to memory
__dcbf	Data cache block flush	Flush cache block, write-back if dirty
__dcbz	Data cache block zero	set block to 0

In the above functions, the memory pointer is used in a content-addressable way; __dcbt(x) provides a hint to the processor that a data block containing x will be loaded in the near future. As the PowerPC does not write-back cache lines to main memory when they are modified, the PPE provides an intrinsic __dcbst to perform an immediate write-back to main memory. The __dcbt intrinsic acts as a data-prefetch hint only, if the requested data is not found in the L2 cache it is requested from main memory, but not forwarded to the cache. However, for the __dcbtst, data is forwarded to the L1 cache. The __dcbz instruction is treated as a store instruction from the program order point-of-view, but because the L2 cache processes the write-zero 128-bits at a time it can be significantly faster than writing 8-byte stores. The data cache flush and data cache block store instructions generally finish before the write-back to memory completes. Thus a context-synchronization instruction is inserted after these calls. For more details on the cache instruction, please see Section 6.1.5 of the Cell Broadband Engine Programmer's Handbook [19].

The __lwsync instruction can be used by the PPE when it wants to ensure transfer of data from the PPE LS (using SPE initiated DMA) is complete. The

`mfc_read_tag_status_all()` function on the SPE only guarantees that the LS is available for use, but not necessarily that the data has arrived at main store. Requiring the SPE to always issue an `mfcsync` command before sending a mailbox notification to the PPE that it has initiated transfer is very inefficient. A better and thus preferred approach is for the PPE to issue `__lwsync` after receiving mailbox notification from SPE, but before accessing any of the computational results.

2.7.1 The `eieio` instruction

The `eieio` instruction (enforce in-order execution of I/O) ensures that all main-storage accesses caused by instructions proceeding the eieio have completed, with respect to main storage, before any main-storage accesses caused by instructions following the `eieio`. An `eieio` instruction issued on one PPE thread has no architected effect on the other PPE thread. However, there is a performance effect because all load or store operations (cacheable or noncacheable) on the other thread are serialized behind the `eieio` instruction. The `eieio` instruction is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load or store combining. The first implementation of the CBEA (the CBE processor) has a fully ordered and coherent memory interface. This means that the SPE need not issue `mfcsync` or `mfceieio` commands to ensure correct ordering, simple `mfc_read_tag_status_all()` or using a barrier or fence attribute will suffice.

2.7.2 Instruction cache

Both L1 caches are dynamically shared by the two PPE threads; a cache block can be loaded by one thread and used by the other thread. The coherence block, like the cache-line size, is 128 bytes for all caches. The L1 ICache is an integral part of the PPEs instruction unit (IU), and the L1 DCache is an integral part of the PPEs load/store unit (LSU), as shown in Figure 2-2 PPE Functional Units on page 50. Accesses by the processing units to the L1 caches occur at the full frequency of the CBE core clock.

   The PPE uses speculative instruction prefetching (including branch targets) for the L1 ICache and instruction prefetching for the L2 cache. Because the PPE can fetch up to four instructions per cycle and can execute up to two instructions per cycle, the IU will often have several instructions queued in the instruction buffers. Letting the IU speculatively fetch ahead of execution means that L1 ICache misses can be detected early and fetched while the processor remains busy with instructions in the instruction buffers. In the event of an L1 ICache miss, a request for the required line is sent to the L2. In addition, the L1 ICache is also checked to see if it contains the next sequential cache line. If it does not, a prefetch request is made to the L2 to bring this next line into the L2 (but not into the L1, to avoid L1 ICache

pollution). This prefetch occurs only if the original cache miss is committed (that is, all older instructions must have passed the execution-pipeline pointcalled the flush pointat which point their results can be written back to architectural registers). This is especially beneficial when a program jumps to a new or infrequently used section of code, or following a task switch, because prefetching the next sequential line into the L2 hides a portion of the mainstorage access latency.

When an L1 ICache miss occurs, a request is made to the L2 for the cache line. This is called a demand fetch. To improve performance, the first beat1 of data returned by the L2 contains the fetch group with the address that was requested, and so returns the data critical-sector first. To reduce the latency of an L1 miss, this critical sector of the cache line is sent directly to the instruction pipeline, instead of first writing it into the L1 cache and then rereading it (this is termed bypassing the cache). Each PPE thread can have up to one instruction demand-fetch and one instruction prefetch outstanding to the L2 at a time. This means that in multithread mode, up to four total instruction requests can be pending simultaneously. In multi-threading mode, an L1 ICache miss for one thread does not disturb the other thread.

2.7.3 Data cache miss

Loads that miss the L1 DCache enter a load-miss queue for processing by the L2 cache and main storage. Data is returned from the L2 in 32-byte beats on four consecutive cycles. The first cycle contains the critical section of data, which is sent directly to the register file. The DCache is occupied for two consecutive cycles while the reload is written to the DCache, half a line at a time. The DCache tag array is then updated on the next cycle, and all instruction issue is stalled for these three cycles. In addition, no instructions can be recycled during this time. The load-miss queue entry can then be used seven cycles after the last beat of data returns from the L2 to handle another request.

Instructions that are dependent on a load are issued speculatively, assuming a load hit. If it is later determined that the load missed the L1 DCache, any instructions that are dependent on the load are flushed, refetched, and held at dispatch until the load data has been returned. This behavior allows the PPE to send multiple overlapping loads to the L2 without stalling if they are independent. In multithreading mode, this behavior allows load misses from one thread to occur without stalling the other thread. In general, write-after-write (WAW) hazards do not cause penalties in the PPE.

2.8 Understanding the PPE Pipeline

Although the PPE serves as the control-pane for our applications, we have deliberately not used any of the vector functionality of the PPE, instead relying on the

SPUs. But as we promised in the Preface, understanding the PPE architecture in detail also carries over to the other PowerPC based consoles and embedded systems out there. In this section we shall look at the PPE pipeline in some detail.

The PPE is a load-store RISC architecture. We have already presented the system design aspects of the instruction issue, pipelining and caches. In this section we familiarize ourselves with the PPE instruction set and assembly listings. Though the main focus of this book is on SPU optimization, knowledge of the PPE instruction scheduling and pipelining can come in handy when extracting the last clock-cycle worth of optimization on the Cell. Moreover, as we have stated earlier, the PowerPC ecosystem is thriving, especially in the embedded systems space, and the Cell in the PS3 provides an exciting and simple machine to learn PowerPC programming.

Consider the code fragment shown in Listing 2.1.

```
typedef float matrix_t[100][100];
void mult(int size,int row,int col,
matrix_t A, matrix_t B,matrix_t C) {
  int pos;
  C[row][col] = 0;
  for(pos = 0; pos < size; ++pos)
    C[row][col] += A[row][pos] * B[pos][col];}
```

Listing 2.1 Function to perform matrix multiply inner loop

When compiled on the PPE using `gcc -O2` we get:

```
;C[row][col]+=A[row][pos]*B[pos][col];
lfsx 0,9,4
addi 9,9,4
lfs 12,0(7)
addi 7,7,400
lfsx 13,5,11
fmadds 0,0,12,13
stfsx 0,5,11
bdnz .L4
blr
```

Listing 2.2 PPE Assembly listing for matrix multiply.

Table 2.3 PowerPC PPE instructions used in matrix multiply inner loop.

Instruction	Description
lfsx	load floating-point single (indexed xform)
addi	Add immediate
fmadds	single-precision floating point multiply add
stfsx	store floating point single (indexed xform)
bdnz	branch conditional
blr	branch to link register

We also compiled the same function on an Opteron system using `gcc -O2`, the output assembly is given below:

```
     ;C[row][col]+=A[row][pos]*B[pos][col];
     movss (%rcx,%r8,4),%xmm0
     incl  %r9d
     mulss (%rax), %xmm0
5    incq  %r8
     addq  400, %rax
     cmpl  %r9d, %edi
     addss (%r10,%rdx,4), %xmm0
     movss %xmm0, (%r10,%rdx,4)
10   jne .L4
     rep ; ret
```

Listing 2.3 Opteron assembly listing for matrix multiplication.

Consider the floating-point intensive code shown in Listing 2.4.

```
void VMULT( float* X, float* Y, float *Z,
            float t, float r,
            unsigned long int N) {
  unsigned long int i=0;
5   for(i=0;i<N;++i)
    X[i] = Y[i] * (X[i]*t + Z[i]*r);
}
```

Listing 2.4 Floating-point computation on PPE.

The above code was compiled using `gcc -O2 -funroll-loops` and the generated assembly is given below in Listing 2.5.

```
          .file   "vmult.c"
  # rs6000/powerpc options: -msdata=data -G 8
          .globl VMULT
          .type   VMULT, @function
5   VMULT:
   .LFB2:
   .LVL0:
          mr. 0,6  #, N
          mtctr 0  # N,
10        beqlr 0
   .LVL1:
          li 9,0   # ivtmp.33,
   .L4:
          lfsx 0,9,5      #* Z, tmp136
15        lfsx 12,9,3     #* X, tmp134
          fmuls 0,2,0     # tmp135, r, tmp136
          lfsx 13,9,4     #* Y, tmp138
          fmadds 12,1,12,0        # tmp137, t, tmp134, tmp135
          fmuls 13,13,12  # tmp139, tmp138, tmp137
20        stfsx 13,9,3    #* X, tmp139
          addi 9,9,4      # ivtmp.33, ivtmp.33,
          bdnz .L4        #
   .LVL2:
          blr
25  .LFE2:
          .size   VMULT,.-VMULT
```

Listing 2.5 PPE floating-point instruction analysis.

A slightly more detailed assembly listing can be generated as well (only relevant portions are shown):

```
   2:vmult.c       ****   unsigned long int i=0;
   3:vmult.c       ****   for(i=0;i<N;++i)
```

```
    54                                    .loc 1 3 0
    55 0000 7CC03379                      mr. 0,6   #, N
    56 0004 7C0903A6                      mtctr 0   # N,
    57 0008 4D820020                      beqlr 0
    58                        .LVL1:
    59 000c 39200000                      li 9,0    # ivtmp.33,
    60                                    .p2align 4,,15
    61                        .L4:
     4:vmult.c       ****      X[i] = Y[i] * (X[i]*t + Z[i]*r);
    63 0010 lfsx 0,9,5       #* Z, tmp136
    64 0014 fsx 12,9,3       #* X, tmp134
    65 0018 muls 0,2,0       # tmp135, r, tmp136
    66 001c fsx 13,9,4       #* Y, tmp138
    67 0020 madds 12,1,12,0 # tmp137, t, tmp134, tmp135
    68 0024 muls 13,13,12   # tmp139, tmp138, tmp137
    69 0028 tfsx 13,9,3      #* X, tmp139
    71 002c ddi 9,9,4        # ivtmp.33, ivtmp.33,
    72 0030 dnz .L4          #
    75 0034 lr
```

Given the PowerPC assembly language and architecture books [15] it is straightforward to follow the assembly language. Another useful reference is the Programming Environments Manual [113] for 32-bit implementations of PowerPC architecture by Freescale Semiconductors. Since the PPE has a single floating-point unit there is no superscalar issue of floating point instructions.

We exercise this code using a simple test wrapper as:

```
extern void VMULT( float*,float*,float*,float,float,unsigned long int);
int main( int argc, char* argv[] ) {
  long time_diff = 0;
  unsigned long int i=0;
  struct timespec t1,t2;
  Setup();
  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t1 );
  for(i=0;i<1000;++i)
    VMULT(X,Y,Z,r,t,N);
  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t2 );
  time_diff = t2.tv_nsec = t1.tv_nsec;
  printf("\n [%f:%f] it took %ld ns.\n", X[0],X[MAX-1],time_diff);
  return (EXIT_SUCCESS);
}
```

Listing 2.6 Timing PPE floating point performance.

On the PS3, this code performs in 1437120 nano-seconds, and on an Opteron 242 it takes 1876521 nano-seconds. The PS3 PPE is running at 3.2 GHz while the Opteron is running at 1.8 GHz. The Opteron has a dual-issue floating-point pipeline. The assembly code for the Opteron is shown as well:

```
.globl VMULT
        .type   VMULT, @function
VMULT:
.LFB2:
        .file 1 "vmult.c"
        .loc 1 1 0
.LVL0:
        .loc 1 3 0
        testq   %rcx, %rcx       # N
        .loc 1 1 0
        movaps  %xmm0, %xmm3     # t, t
        movaps  %xmm1, %xmm2     # r, r
        .loc 1 3 0
```

```
            je      .L6      #,
15  .LVL1:
            xorl    %eax, %eax      # i
    .LVL2:
            .p2align 4,,7
    .L4:
20          .loc 1 4 0
            movaps  %xmm3, %xmm0    # t, tmp70
            movaps  %xmm2, %xmm1    # r, tmp71
            mulss   (%rdi,%rax,4), %xmm0    #* X, tmp70
            mulss   (%rdx,%rax,4), %xmm1    #* Z, tmp71
25          addss   %xmm1, %xmm0    # tmp71, tmp70
            mulss   (%rsi,%rax,4), %xmm0    #* Y, tmp70
            movss   %xmm0, (%rdi,%rax,4)    # tmp70,* X
            .loc 1 3 0
            incq    %rax    # i
30          cmpq    %rax, %rcx      # i, N
            jne     .L4      #,
    .LVL3:
    .L6:
    .LVL4:
35          .loc 1 5 0
            rep ; ret
    .LFE2:
            .size   VMULT, .-VMULT
```

Listing 2.7 Analysis of floating-point instructions on Opteron.

2.8.1 Discuss assembly language

Flynn [47] provides an excellent quantitative analysis of the many choices which are presented to superscalar pipeline architects. Considering the PPE as a pure load-store architecture with well defined pipeline behavior, we can understand the generated code and the actions of the compiler. This knowledge can guide us when we are writing code for the PPE.

2.9 Measuring and Profiling Application Performance

The Cell Broadband Engine (CBE) processor provides extensive performance-monitoring facilities that assist performance analysis, as well as provide application-optimized and system-optimization features that include:

1. Debugging, analyzing, and optimizing processor-architecture features
2. Profiling the behavior of the memory hierarchy and the interaction of multiple address spaces, as well as tuning system and application algorithms to optimize scheduling, partitioning, and structuring for tasks and data
3. Real-time application-tuning by monitoring bandwidth use and other resource-management behavior
4. Tuning of numerically intensive floating-point applications
5. Tuning of fixed-point applications

The facilities give clear visibility to the details of instruction execution, loads and stores, the behavior of caches throughout the CBE processor, and the entire virtual-memory architecture.

2.10 Conclusion

Since the PPE is PowerPC compliant, much of what is applicable to PowerPC (eg., PowerPC 970MP) is valid and germane to PPE. Thus, we have been able to describe only a fraction of the facilities provided by the PPE. Other excellent references for PowerPC information are the PowerPC Architecture Books [15], and the PowerPC Compiler Writer's Guide [13]. GNU/GCC toolchain for the PowerPC is readily available and investigations of the produced assembly are useful to learn the instruction set of PPE. In the sequel of this book we shall rely on the PPE to provide GNU/Linux facilities, file-system interaction, memory-management and SPE runtime management. We have not presented any VMX code for SIMD processing on the PPE as this topic is relatively mature (due to widespread acceptance of AltiVec), and in our opinion the SPUs provide significantly elegant infrastructure to design SIMD and parallel programs. The SPE, its design, ISA and pipeline is the focus of the next chapter.