A Brief View of the Cell Broadband Engine

Cris Capdevila
Adam Disney
Yawei Hui
Alexander Saites

02-Dec-2013

Introduction

The cell microprocessor, also known as the Cell Broadband Engine (CBE), is a Power Architecture-based microprocessor developed for high-workload and multimedia-based computing. The cell processor was designed by STI, a partnership among Sony, Toshiba, and IBM, although IBM remains the key developer of the project. Built with massive floating point operations in mind, the original processor was developed with a Power Architecture core surrounded by eight “synergistic” processors. Although the cell processor has strong computing potential, it is widely regarded as a challenging programming environment [1]; as a result, interest in the processor has waned. In this paper we discuss the characteristics of the Cell microprocessor: we start with a brief history, discuss its three principal components, introduce its instruction set architecture, and finish with a summary of its programming challenges.

A Brief History

The IBM Cell architecture is the product of a joint engineering effort by three companies, Sony, Toshiba and IBM. In 2001 the three companies established the STI (Sony Toshiba IBM) Design Center in Austin, Texas to develop a new “supercomputer on a chip.” The work involved over 400 engineers and took approximately 4 years to complete [2]. This project came about at the request of then Sony Computer Entertainment Inc. CEO Ken Kutaragi. Kutaragi was coming off the immensely successful release of the PlayStation 2, and was looking for new technologies to give the company a leg up over Microsoft in the next generation game console cycle. Unsatisfied with the microprocessors available at the time, Kutaragi challenged IBM to develop a new, faster design in time for the 2005 announcement of the PlayStation 3.

Although initially targeted at the PlayStation 3, the engineers at the STI Design Center wanted to create an architecture that would be more general purpose. Initial reports suggest that they were chasing the embedded market as well, with Cell proposed for things like televisions and home theater systems — applications that require high data throughputs. These ambitious goals were reflected in the product name they chose, the Cell Broadband Engine Architecture (CBEA), emphasizing that it was a high bandwidth processor. Ken Kutaragi, the president and CEO of Sony Computer Entertainment Inc., visualized the processor as a completely new way of handling multi-core designs: he related the system to that of a biological unit, a network of “cells” forming the building blocks of a larger system. He felt that “SCEI, IBM, and Toshiba are mapping out the future of broadband computing” [2].

To promote development on the processor, IBM held the “Cell Broadband Engine Processor University Challenge” in 2007, challenging university students to develop applications using the new CBE. The first-place students used a cluster of PS3s to perform massively-parallel image processing based on algorithms meant to model the visual processing performed by the human brain. In their paper, they stated that they were able to “[demonstrate] the potential to support the massive parallelism inherent in these algorithms by exploiting the CELL’s parallel instruction set and by further extending it to low-cost clusters built using the Sony PlayStation 3 (PS3)” [3].

In 2008, IBM produced the PowerXCell 8i, which formed the backbone of the IBM Roadrunner, the first supercomputer to break the petaflop barrier. The new design used the 65 nm process and increased the floating-point performance of the SPEs. When it was built, it had the highest MFLOPS/Watt ratio of any supercomputer in the world. The following year, versions of the PS3 were released with the 45 nm Cell Processor.

Despite the potential of the Cell processor, it was widely considered a difficult environment for programmers [1]. With the development of the PS4, Sony decided to move towards an x86 architecture, an easier environment for developers. [5] Toshiba abandoned development of HD televisions based on the Cell. Even IBM halted development of the Cell processor with 32 APUs and the Blade server line based on the processor [6], and for the most part direct use of the cell processor has ceased.

Overview: PPE, SPE, EIB

The Cell Broadband Engine Architecture was designed for distributed processing in a scalable way to facilitate a family of processors with a range of available cores and memory configurations. This allowed the Cell family to address different domains while using the same basic hardware design [7]. A Cell BE system has:

- one or more PowerPC Processor Elements (PPEs), which handles system management
- one or more Synergistic Processor Elements (SPEs), which perform the data-heavy, parallel computation
- one internal interrupt controller (IIC); and
- one Element Interconnect Bus (EIB) for connecting units within the processor.

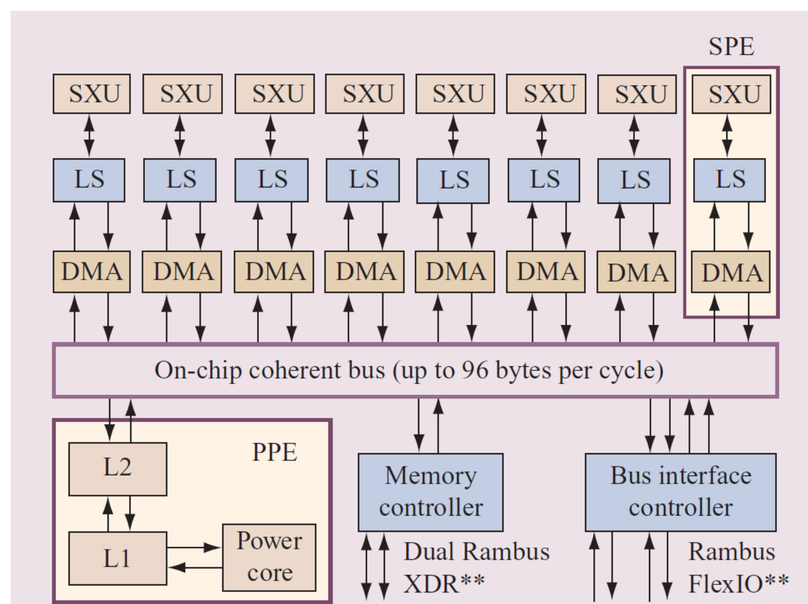

Figure 1. A general diagram of the Cell processor [8].

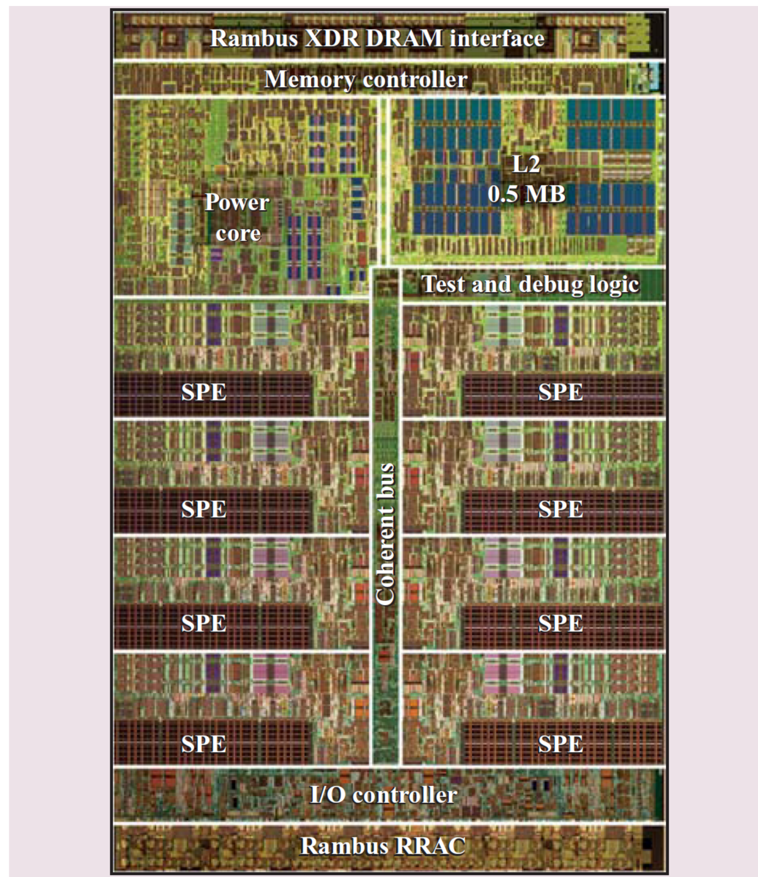

Figure 2. An actual die photo of the Cell processor [8].

The first-generation Cell processor had 1) a general purpose 64-bit RISC dual threaded, dual-issue PPE with 32K L1 cache and 512K L2 cache; 2) eight SPEs; 3) an on-chip memory controller; 4) and a controller for a configurable I/O interface. Figure 1 shows the diagram of the layout of Cell processor, and Figure 2 the actual die of the chip [8].

Performance Metrics

Metric	Performance
SPE to SPE Bandwidth	78 – 180 Gb/s
Memory Read/Write	21 Gb/s
I/O Interface	25 (inbound) – 35 Gb/s (outbound)
Optimized Matrix-Matrix Multiple	201 GFLOPs
LINPACK Optimized	155.5 GFLOPs
Single SPU MPEG2 decode	77 frames of HDTV
AES Cryptography on SPE	2.0 Gb/sec

Table 1. Performance Metrics of the Cell Broadband Engine [9].

Power Processor Element

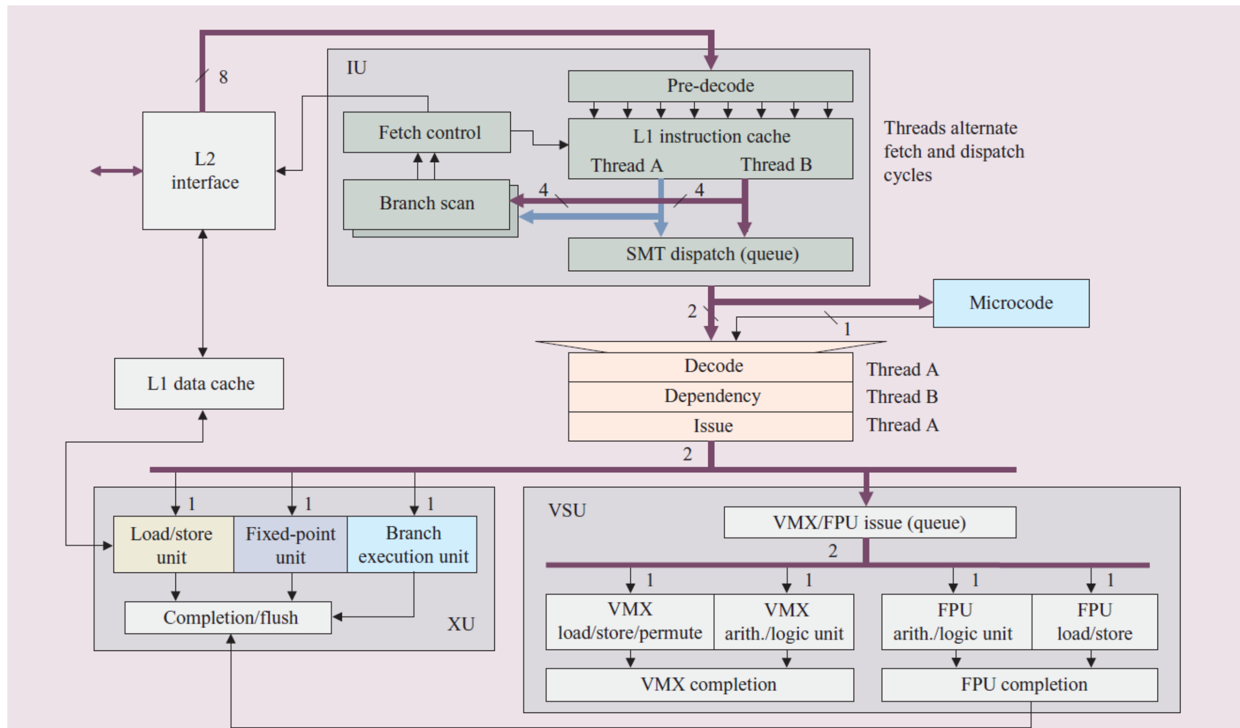

Figure 3. Diagram of the major units of a PPE [8].

The PPE (Figure 3) is a 64-bit (including all effective addresses, general registers and most special purpose registers) “Power-Architecture compliant core optimized for high frequency and power efficiency” [8]. It can run in either 64-bit or 32-bit mode with the same instruction set which controls operations such as interpretation of the effective address, setting of status bits, and branch-testing on the count register. When computing the effective address, all 64 bits of the general registers are used to produce a 64-bit result. However, only in 64-bit mode the entire result (64-bit) will be used to access data. On the other hand, only the lower 32-bit of the result will be used in 32-bit mode to access data and fetch instructions [7]. Specifically, the PPE has three units (Figure 3):

1. The instruction unit (IU) handles instruction fetch, decode, branch, issue, and completion;

2. The fixed point execution unit (XU) handles all fixed point instructions and all load/store-type instructions. It consists of a 32 by 64-bit general-purpose register file per thread, a load/store unit (LSU), a fixed-point execution unit, and a branch execution unit. The LSU consists of the L1 D-cache, a translation cache, an eight-entry miss queue, and a 16-entry store queue;
3. A vector scalar unit (VSU) handles all vector (through the vector multimedia extensions (VMX) unit) and floating-point (through the floating-point unit (FPU)) instructions. It

consists of a 32 by 64-bit register file per thread, as well as a ten-stage double-precision pipeline. The VMX contains four subunits: simple, complex, permute, and single-precision floating point, and a 32 by 128-bit vector register file per thread. All VMX instructions are 128-bit SIMD with varying element widths of 2×64-bit, 4×32-bit, 8×16-bit, 16×8-bit, and 128×1-bit.

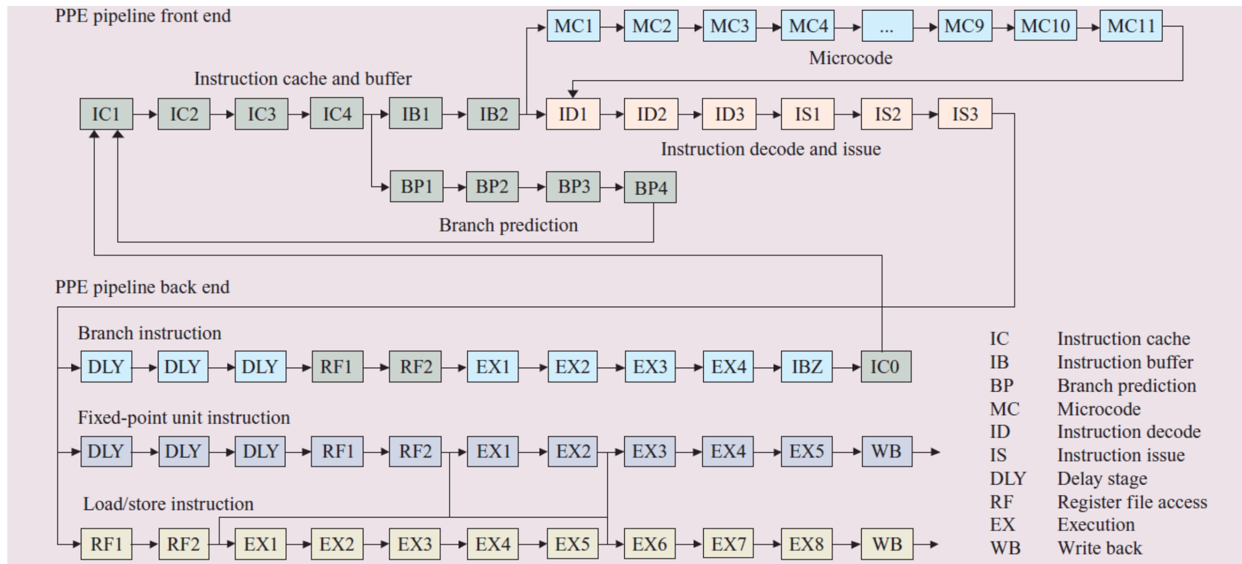

Figure 4. Structure of the pipeline of a PPE [8].

Comparing to other multi-issue out-of-order processors, the pipeline structure of the PPE is relatively simple with only 23 pipeline stages (Figure 4), partially due to the “short-wire” design which limits the amount of communication delay in every cycle [8]. There are six instruction fetch and cache/buffer stages. These fetch four instructions per cycle per thread into an instruction buffer from which they are issued in order (but interleaved). The IU uses a 4-KB by 2-bit branch history table with 6 bits of global history per thread to predict branching. There are six decode/issue stages that process the decoding and dependency checking before dual-issuing instructions to an execution unit. Most instruction combinations can be dual-issued, but there are a few exceptions in the case of resource conflicts: “simple vector, complex vector, vector floating-point, and scalar floating-point arithmetic can not be dual-issued with the same type of instructions” [8]. Despite this, it is possible to dual-issue these instructions “with any other form of load/store, fixed-point branch, or vector-permute instruction” [8].

After entering the back end of the pipeline, in XU the longest operation (Load/Store) takes two register file access stages, eight execution stages and one final write back stage (Figure 4). Other, shorter back end operations such as branch and fixed-point unit operations are stuffed with delayed-execution stages (which are a means to provide some benefits of out-of-order execution to avoid stalls at issue stages). This allows all of them to complete and forward their results in two cycles. A vector-scalar unit (VSU) allows vector and floating-point instructions to be issued to separate pipelines, thus allowing them to be issued out of order with respect to one

another [8].

As shown above, the PPE is designed as a dual-issue processor with in-order issue. Two simultaneous threads of execution are provided by the PPE; these interleaved instructions allow the PPE to maximize its use of instruction slots, increase efficiency, and reduce pipeline depth [8]. By duplicating all architected states (including all architected registers and most of special-purpose registers), the programmer can view the processor as a “two-way multiprocessor with shared data flow” [8]. Caches and queues are usually shared for both threads [8].

The PPE’s cache hierarchy consists of two L1 32-KB caches (one for instructions, one for data), along with a 512-KB unified (instruction and data) L2 cache. The L1 instruction cache is organized as a two-way associative memory with 512 wordlines that access 32 bytes per way. It is SRAM based and clocked at the full clock rate with a three-cycle read latency. These cycles are spent generating the address; accessing the array; and parity checking, data formatting, and way selecting [9]. Writes are performed one cache line at a time in 32-byte wordlines, but they may be 64 bytes wide (in which case two consecutive wordlines are addressed) [9].

The L2 cache is clocked at only half the rate and includes the L1 data cache. It’s built with eight SRAM macros and organized as a 512-KB eight-way associative memory. It can only perform one read or write per cycle, though writes complete in two cycles and reads complete in either three or four depending on the size (140-bit or 280-bit). It supports write-back and allocation on store miss, along with “ECC on data, parity on directory tags, and global and dynamic power management” [9]. In particular, due to its organization, only ⅛ of the L2 cache need be activated during a read or write, since the way selected is decoded before accessing the array. Thus, the L2 has high power conservation while still allowing reads and writes to be interleaved, provide continuous data transfers to and from the L2 [9].

Synergistic Processing Elements

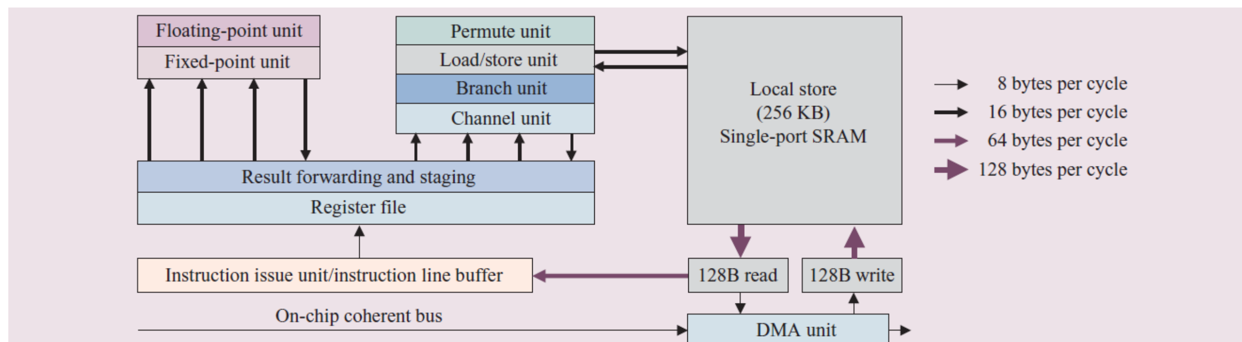

Figure 5. Diagram of the major units of an SPE [8].

The SPEs (Figure 5) are dual-issue SIMD RISC cores with all instructions encoded in 32-bit fixed-length instruction formats. It consists of three main units:

1. the synergistic processor unit (SPU), which possesses a 128 by 128-bit SIMD register file used by both floating-point and integer instructions. This “facilitates efficient instruction scheduling and [...] enables important optimization such as loop unrolling” [10];

2. the local store (LS), a special block of memory local to an individual SPE. It’s arranged as a 256 KB non-cached memory and is the largest component in the SPE. It stores both instructions and data used by the SPU [8];

3. the memory interface controller (MFC), which facilitates the communication between the SPU/LS and the main memory and other processors in the Cell. MFC commands are issued by the SPUs or the PPEs and carried out by the direct memory access (DMA) controller, which actually moves the data.

The SPUs have a focus on vector-based data. They are meant to perform the computationally heavy (and hopefully parallel) instructions for any particular program. All the instructions for the SPU are SIMD and use the 128-bit data as 2×64-bit double precision, 4×32-bit single precision, 8×16-bit integers, 16×8-bit integers and of course 128×1-bit [8,10]. By using the LS, many memory transactions can be in-process without requiring “deep speculation that drives high degrees of inefficiency on other processors” [10]. This significantly decreases the overhead for memory access, but puts the burden of data access on the programmer and compiler, who must determine data prefetches at compile-time.

Access to the LS outside of its SPE is managed by the MFC via a DMA controller. Each SPE can use its MFC to issue up to 16 asynchronous, coherent DMA commands to transfer data

and instructions between the LS and system memory. The PPE and SPEs share the same translation protocol and protections (via the page and segment tables), so the OS can manage the system memory across the SPEs easily. The LS has two read/write ports which supports 128-bit wide DMA transfers, and a single command can transfer up to 16 KB of sequential data [10]. Transfer requests are tagged (in software) with information about the storage type being accessed. They are then queued in the MFC so that the EIB can efficiently prioritize transfers. Since this access is only available every eight cycles, a given DMA transfer takes 16 cycles to put all data on the EIB. The extra clock cycles are left for data load/store and instruction prefetch [10]. The SPU prioritizes access to the LS as: DMA transfers, data load/store, and instruction pretech.

Contention for memory resources is of course an issue with such a highly parallel structure, and the MFC naturally has several ways of managing this issue. This includes a multisource synchronization via atomic memory updates; however, these require constant polling of the memory location by the SPU, which is inefficient. As a result, a more typical locking mechanism involves setting a reservation on the memory location of the lock. A program can then request an SPU event or interrupt when the memory “reservation” is lost (i.e., another processor modifies the lock [10]). Although this does not guarantee a lock value has changed, it does allow the SPU to only check the lock value when it has changed. This method can be inefficient when there is high contention for a lock [10].

The SPU has other mechanism for synchronization and message passing. Perhaps most interesting among them is the mailbox facility. The mailbox is designed to assist in process-to-process communication and synchronization by providing “a simple, unidirectional communication mechanism typically used by the PPE to send short commands to the SPE and to receive status in return” [10]. There are three mailboxes: inbound, outbound, and outbound interrupt, and each are able to hold one or more entries. The SPU uses channel instructions to read the inbound mailbox or send to the outbound mailbox. These instructions block if the inbound mailbox is empty or outbound mailbox is full. Writing to the outbound box causes an interrupt that is routed to the PPE for processing [10]. The typical use-case involves the PPE issuing commands to the SPU. The SPU carries out the commands and issues a response to the PPE before reading the next instruction in the mailbox.

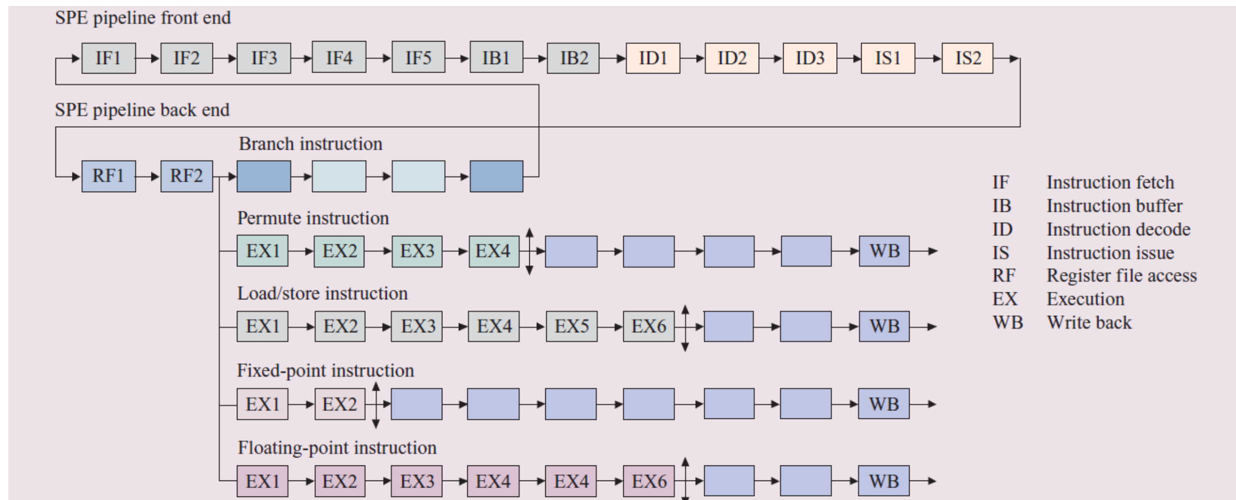

Figure 6. Structure of the pipeline of an SPE [8].

Figure 6 shows the basic structure of the SPE pipeline. Two instructions can be issued each cycle for the SPE -- one for fixed- and floating-point operations and the other for load/stores. The load/store operation cycle can also be used for byte permutation and branching [8]. Most instructions are fully pipelined, but fixed-point instructions take two cycles, and single-precision FP and load instructions take six [8]. The programmer "hints" at branches to notify hardware of an upcoming branch address and target to allow pre-fetching to limit stalls.

As mentioned above most of the instructions operate on 128-bit wide data. The SPE core can perform single precision floating point operations, integer arithmetic, boolean logic (including branching and comparisons), loads, and stores. Up to 32 instructions are buffered, and the SPE uses an “Even/Odd Pipe” to determine scheduling; two instructions are retrieved from the buffer and checked for dependences. If possible, they are executed in parallel; otherwise, they are issued in-order. The SPU only access its LS and register file and has no scalar unit.

Element Interconnect Bus

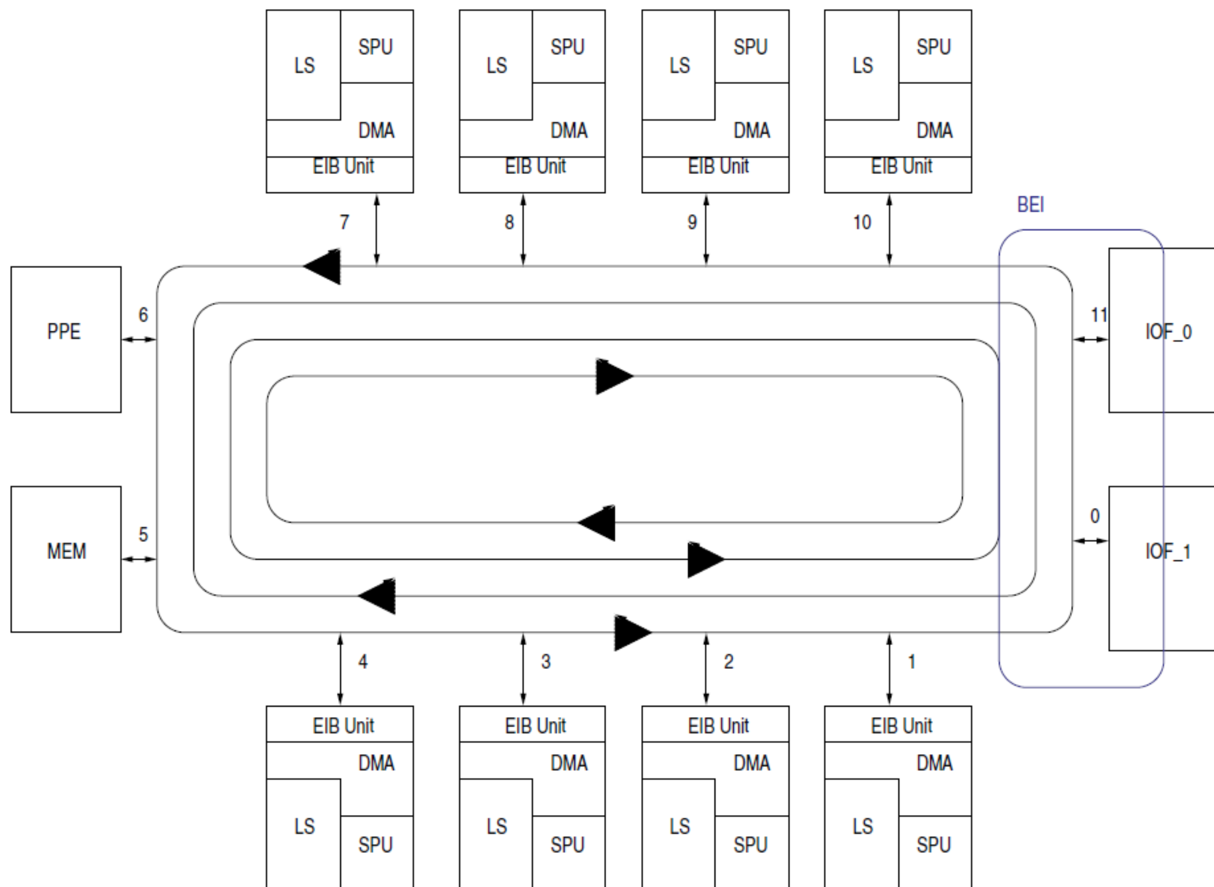

Figure 7. Cell Diagram with Unit ID [9].

The Element Interconnect Bus (EIB) connects the PPE(s), SPE(s), and off-chip memory controllers and I/O devices. The EIB has a token ring-like layout, with four 16 byte wide data rings, two running clockwise, and two running counter-clockwise. The processing elements each have one on-ramp and one off-ramp to the bus, enabling simultaneous send/receives. Each message can be up to 8 cycles long, or 128 bytes (the length of a single PPE cache line). The interconnected units use DMA commands to send data to each other along the bus, with up to three messages able to be transmitted at a time, as long as they don't overlap. [9]

A processor element wishing to use the data ring must request access from the EIB's data ring arbiter. In order to reduce memory related stalls, the arbiter gives the highest priority to DMA requests from memory controllers and treats all other requests as equal. The requester can use any of the four data rings. Which one it gets assigned to is decided by the arbiter. Once the arbiter decides that a requester can send data over the ring, it gives that requester a token. Once the requester has sent its data and the receiver has received it, the token is returned to the arbiter to be re-used.[9]

Data transfer latencies depend on the distance from, or number of hops, from the sender to the receiver. Clearly, transfers to adjacent units are fastest, and the farther around the ring a message has to go, the longer it takes [11]. The arbiter will not force transfers more than half way across the ring, instead it will wait until the shorter path (going in the opposite direction) is available. [9]

Instruction Set Architecture

The PPE instruction set (PPEIS) is designed as a superset of the PowerPC Architecture instruction set (with a few additions and changes), which adds new vector/SIMD multimedia extensions (VSME) and their associated C/C++ intrinsics [11]. The length of instructions in PPEIS is 4 bytes and most PPEIS instructions can have up to three operands with two source operands and one destination operand. The PPEIS instructions can be categorized in three types [9]:

1. computational, which include fixed-point instructions operating on byte, halfword, word, and double word (e.g. arithmetic, compare, logical, and rotate/shift) and floating-Point Instructions operating on single-precision and double precision floating-points (e.g. floating-point arithmetic, multiply-add, compare, and move);

2. load/store, which include both fixed-point and floating-point load/store instructions. The fixed-point loads and stores support byte, halfword, word, and doubleword operand accesses between storage and the 32 general purpose registers (GPRs). The floating point loads and stores support word and doubleword operand accesses between storage and the 32 floating-point registers (FPRs);

3. system management/control, like memory synchronization, memory flow and processor control, and memory/cache control.

It also has a unique set of commands for managing DMA transfer, external events, interprocessor messaging, and other functions. The SPEIS draws a similarity between itself and the PPE VSME because they both operate on SIMD vectors. But under the hood they are quite different instruction sets which need different compilers for processing programs designated to PPE and SPEs [11].

Unit	Instructions	Execution pipe	Depth	Latency
Simple fixed	Word arithmetic, logicals	Even	2	2
Simple fixed	Shifts/rotates	Even	3	4
Single precision	MAC	Even	6	6
Single precision	Int MAC	Even	7	7
Byte	Byte avg,sum, pop count	Even	3	4
Permute	QW shift/rotates	Odd	3	4
Local storage	Load and store	Odd	6	6
Channel	Channel read and write	Odd	5	6
Branch	Branches	Odd	3	1–18

Table 2. Overview of the SPEIS and its associated pipelines and functional units [9].

In Table 2 is a description of the SPE instructions and their associated pipelines and functional units. Like PPE, SPEs are multi-issue in-order processors which means they issue and complete all instructions in program order and do not reorder or rename their instructions. The SPE features a dual-issue odd-even pipeline (as mentioned in previous section) and has nine execution units lined up to service the dual issue pipeline. Even though out-of-order issue is not supported within the hardware, SPE does allow the compiler to reorder the instruction sequence in order to achieve high dual-issue efficiency, as long as the compile maintains the correct program behavior [9].

Detailed descriptions about the even and old-issued instructions and their associated pipeline characteristics are given in Table 3 and Table 4. If two consecutive instructions have different parities (represented by even or odd addresses) which match the parities of the pipelines, the SPE Instruction pair can be dual-issued. Otherwise, a single-issue will be performed without “no-op padding”, which results in the instructions executing in-order. This instruction issue mechanism simplifies resource and dependency checking [9].

Even Pipeline	Latency	Stalls
Single-precision floating-point	6	0
Integer multiplies, spu_convtf, spu_convts, **fi**	7	0
Immediate loads, logical operations, integer add/sub, select bits	2	0
Double-precision floating-point	7	6
Element rotates and shifts	4	0
Byte operations (pop count, abs. diff., avg, sum)	4	0

Table 3. Detailed description of the even-issued pipeline [9].

Odd Pipeline	Latency	Stalls
Shuffle bytes, quadword rotates/shifts	4	0
Gather, mask, generate insertion control	4	0
Estimate	4	0
Loads	6	0
Branches	4	0
Channel operations, SPRs	6	0

Table 4. Detailed description of the odd-issued pipeline [9].

Programming Challenges

The Cell architecture has developed a reputation of being difficult to program. This is at least partially due to the difficulty in writing concurrent software in general. This is really the case for all modern processors and GPUs. However, due to the limited power of the Cell PPE, the architecture demands efficient distribution of computations among all SPEs in order to get good performance. In 2005, most application programmers were still writing single threaded, sequential code.

Aside from that, there were some difficulties particular to Cell. One issue is that the processor most commonly seen (the one shipped in the PlayStation 3) had only 256k of local storage for both instructions and data dedicated to each SPE. This is a fairly small amount of room to work with, so problems need to be decomposed into chunks small enough to fit. Moreover, moving data in and out of the local storage areas from main memory had to be done manually by the programmer, unlike a cache for example, where the contents are managed by the machine. Management of this memory is crucial since code running on any given SPE only has immediate access to data sitting in that SPE's local storage.

The PPE was fairly easy to work with. It resembled the standard desktop microprocessors that developers were used to. The SPEs, on the other hand, were more difficult. The lack of caches, purely in-order execution, and lack of branch prediction made it difficult to write fast code in the conventional manner. Everything had to be managed by the programmer, with little help provided by the processor itself. Ideally you wanted each SPE running simple, sequential code that did simple operations on large vectors of numbers. Too much logic meant a lot of time stalled in missed branches. Too many pointers to non-contiguous chunks of memory meant time lost doing DMA into and out of the local storage. Unfortunately, not all programs can be broken down into simple, parallel sets of vector operations. For those that couldn't, execution would generally happen mostly on the (underpowered for its time) PPE.

Developers also had to do a lot of work that they had traditionally relied on the compiler to do for them. Loop unrolling and function inlining, for example, had to be done manually. This could have a large impact on execution time since larger basic blocks were easier for the compiler to schedule. Unfortunately, this also made it harder to manage the limited amount of storage space that had to hold both the code and the data. Similarly, larger blocks of data were more efficient to transfer in and out of local storage, but they couldn't be too large to fit in what was left of the SPE's 256k. In order to take advantage of the SPE dual pipelines, the programmer would have to ensure that his code was free of hazards since conflicting instructions would cause an entire pipeline to stall.

The Cell processor confronted programmers with all the “new” problems of writing concurrent software, as well as a set of hardware constraints that made dealing with these problems even more difficult. On top of that, the new, smaller units of code that were to be run on the system's

SPEs had to be written in a far more direct, manual way than most developers were used to. A lot of the help programmers have traditionally gotten from both the hardware and tools (caches, dynamic scheduling, compiler optimizations, reasonably feature rich debuggers) was also unavailable to them. Altogether, this caused a lot of the early bad press Cell got and helped shaped its reputation.

References

[1] Chris Gottbrath. (2008), "Overcoming the Challenges of Developing Programs for the Cell Processor" [Online]. Available: http://www.roguewave.com/documents.aspx?entryid=468&command=core_download

[2] Jim Kahle et al. (2012, March 7) “The Cell Broadband Engine” [Online]. Available: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/cellengine/

[3] Jayram Moorkanikara Nageswaran et al. (2007) “Accelerating brain circuit simulations of object recognition with a Sony PlayStation 3”, *Int. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems,* 2007.

[4] Daniele Scarpazza et al (2007), “Programming the Cell Processor” [Online]. Available: http://www.drdobbs.com/parallel/programming-the-cell-processor/197801624?pgno=1

[5] Rashid Sayed (2013), “Mark Cerny Explains Why The Cell Processor Was Not Included In The PS4”, *GamingBolt*, 2013 June 28 [Online]. Available: http://gamingbolt.com/mark-cerny-explains-why-they-did-not-included-cell-in-the-ps4

[6] Timothy Prickett Morgan (2011), “IBM to snuff last Cell blade server”, *The Register*, 2011 June 28 [Online]. Available: http://www.theregister.co.uk/2011/06/28/ibm_kills_qs22_blade/

[7] IBM, (2007) “Cell Broadband Engine Architecture”, 2007 November 11 [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA

[8] J. A. Kahle et al. (2007), “Introduction to the Cell Multiprocessor”, *IBM J. Res. & Dev.*, vol. 49, no. 4-5, pp. 589, July/Sept, 2005. Available: http://users.ece.utexas.edu/~adnan/grad-vlsi-05/cell-ibm-journal.pdf

[9] S. Koranne (2009), “Practical Computing on the Cell Broadband Engine”, *Springer Science & Business Media*, 2009.

[10] Johns & Brokenshire (2007), “Introduction to the Cell Broadband Engine Architecture”, *IBM J. Res. & Dev.*, vol. 51, no. 5, Sept 2007. Available: http://www.signallake.com/innovation/johns.pdf

[11] A. Averalo et al. (2008), “Programming the Cell Broadband Engine Architecture: Examples and Best Practices”, 2008 [Online]. Available: http://ldc.usb.ve/docs/ProgrammingCell.pdf