Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

New Patent Reveals Cell Secrets

IBM, Sony, Toshiba Develop Secure Parallel-Processing Architecture

By Tom R. Halfhill {1/3/05-01}

No microprocessor since Intel's Merced has stirred as much curiosity as the Cell processor under development by IBM Microelectronics, Sony, and Toshiba. Partly it's because Cell is destined for Sony's much anticipated PlayStation 3 videogame console, due in 2006. But Cell isn't meant just for fun and games. It's also intended for professional graphics workstations and other computing devices, which makes people wonder what kind of magic will be bottled in the chips.

Tantalizing details will trickle out in February, when IBM presents several papers about Cell at the International Solid-State Circuits Conference (ISSCC) in San Francisco. Until then, nothing beats a weighty 57-page patent issued to IBM, Sony, and Toshiba by the U.S. Patent and Trademark Office on October 29, 2004. Patent 6,809,734 describes the Cell architecture in detail, with 42 pages of illustrations.

Of course, the Cell partners applied for their patent on March 22, 2001, so it's likely some things have changed in the four years since then. However, the '734 patent is strikingly similar to an earlier U.S. patent (6,526,491), issued on February 25, 2003 to Sony alone, and to four pending U.S. patent applications by the companies. (See the "For More Information" box.) All the patents and applications describe the same new architecture, which rises from the microprocessor level to encompass complete systems and networks.

A new architecture? Isn't Cell based on the PowerPC chip? That's what many had assumed for years, and IBM has confirmed that its Power architecture (the company's umbrella term for the architecture that includes the PowerPC and Power server processors) is part of Cell. However, the '734 patent refers to "a new architecture for computers, computing devices, and computer networks" and "a new programming model for these computers, computing devices, and computer networks." *Microprocessor Report* believes the "new programming model" is a way of binding program code and data together in special bundles, perhaps as part of a new instruction-set architecture (ISA). The '734 patent describes a much larger register file and other novel architectural features not found in any PowerPC chips today. If Cell isn't a wholly new architecture, it may at least be a significant extension of PowerPC.

The most important information in the '734 and '491 patents is that Cell isn't just a single microprocessor or even a family of processors. It's a top-to-bottom architecture for a broad range of computing systems, from servers and workstations at the high end to game consoles, PDAs, digital TVs, and other consumer electronics at the low end. The name Cell derives from the architecture's "software cells," which combine program code, data, global identification numbers, and other metadata in formatted bundles. Software cells can freely migrate in search of execution resources—whether those resources are in a single chip, spread across multiple chips in a system, or distributed across multiple systems on a local or global network. With the Cell architecture, clustering and grid computing are native concepts. It's a new parallel programming model for a fast-approaching age of universal multiprocessing.

To date, most attention has focused on Cell as a whiz-bang videogame chip. It's not unrealistic to expect sales of 100 million units for the PlayStation 3 over its product lifetime, but the Cell partners appear to have even bigger ambitions in mind. The PlayStation 3 could be merely the high-volume consumer vehicle that pays for the startup costs of a considerably larger project. Indeed, the patents boldly propose Cell as a higher-performance alternative to Java virtual machines for spreading a standard programming model across the full range of client devices on the Internet. Of course, that dream assumes a homogeneous universe of Cell-based clients and servers—a highly unlikely scenario, no matter how innovative the processors are.

High Scalability and Distributed Processing

On a technical level, the '734 patent contains numerous noteworthy nuggets about the Cell architecture. Cell processors can have many different microarchitectures, depending on the processing needs of the computing devices for which they're intended, but all share the same basic characteristics. Here's an overview:

Cell processors have multiple groups of function units that can operate in parallel to execute SIMD instructions. At run time, a processor can dynamically link arbitrary numbers of function units together to form a temporary pipeline dedicated to an instruction stream. Even when this pipeline isn't busy, it prevents other instruction streams from preempting its resources. Each group of function units has its own register file, local memory, direct-memory access (DMA) controller, and dedicated banks of DRAM. Special protection mechanisms prevent different instruction streams from accessing each others' memory or processing resources. These mechanisms—implemented in both the processor and DRAM—maintain memory coherency, provide security, and enforce digital-rights management (DRM) for copyrighted content.

Other features are as follows:

- If a task needs more capacity than a single Cell processor can provide, a multiprocessor system can readily distribute the workload among its other Cell processors, optionally using optical wave guides instead of wires for chip-to-chip communications.
- If a Cell-based system is attached to a broadband network (the expected default condition), it can distribute a workload among other Cell-based systems on the network.

In either case—system-level multiprocessing or distributed network processing—the program code and data travel together in the aforementioned software cells.

- If a software cell must cross an external network to reach its destination, it can wrap itself in any standard network protocol, such as a TCP/IP packet. Figure 1, from the '734 patent, illustrates the way software cells can travel over a public network linking several different kinds of Cell-based systems.

Another interesting feature: if a more powerful Cell processor would execute a timed instruction stream (such as audio or video) faster than the stream was intended to execute on a less powerful Cell processor, the faster processor can synchronize execution to what the patent calls an "absolute timer." Alternatively, the faster processor can automatically insert NOP (no operation) instructions to consume extra CPU cycles.

All these features make Cell ideal for gaming. Cell's scalable microarchitecture allows designers to create a variety of processors for a broad range of game systems. Cell processors can execute multiple instruction streams in parallel and run multiplayer games over broadband networks while protecting copyrighted content. The hardware-level security features in the CPU and DRAM will ward off malicious hackers and bootleggers. Cell's dynamically dedicated pipelines, distributed processing, and synchronized execution can guarantee isochronous performance on Cell-based systems with vastly different amounts of processing power: home video-game consoles, game-enabled cellphones, hand-held game machines, speedy desktop PCs, or powerful game servers. The absolute timers and automatic NOPs could also make game software written for PlayStation 3 more compatible with later, faster

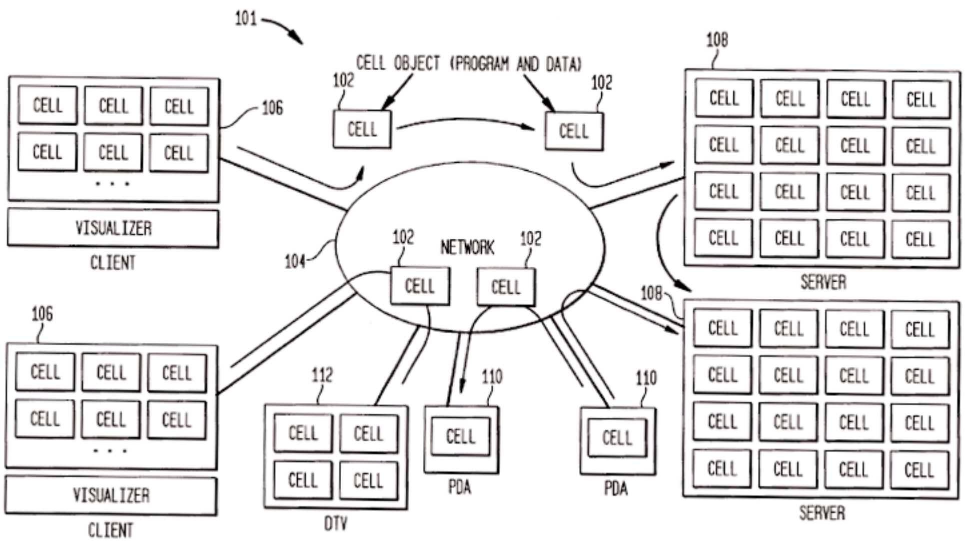

Figure 1. This figure from the '734 patent shows the way packages of program code and data, called software cells, can migrate among different Cell-based systems for distributed processing. The cells are highly mobile: they can move between Cell processors within a multiprocessor system or travel over external networks, such as LANs, public switched telephone networks (wired or wireless), or the Internet. For journeys over external networks, the cells encapsulate themselves in wrappers with standard network protocols. Each cell also has a unique identifier (similar to an Internet Protocol address) that specifies the destination for results after execution.

systems, like a PlayStation 4. Of course, all these features could be useful for nongame systems as well.

The ’734 patent describes many features as “preferred” but not required. Some descriptions seem too detailed for a general outline of a microprocessor or of a system architecture. For example, the patent discusses 1,024-bit DRAM and optical chip-to-chip interfaces, which most Cell systems probably won’t have. (The PlayStation 2 uses Rambus memory, and Sony maintains a Rambus license.) The key features appear to be the scalable microarchitecture, dynamically allocated pipelines, mobile software cells, distributed parallel processing, low-level security, and mechanisms for ensuring isochronous performance. The main invention claimed by the ’734 patent is the ability to dynamically allocate pipelines.

Examining Cell Under a Microscope

Now let’s examine the ’734 patent in more detail. Cell supports many possible implementations, as would be expected for an architecture that spans enterprise-class servers and hand-held clients. The basic building block is a processor element (PE); the least powerful Cell processor would have a single PE. The patent says a typical Cell processor might have four PEs, while a higher-end chip might have eight, including some PEs with specialized function units.

Figure 2 is a high-level block diagram of the most common type of PE, according to the patent. The PE has a processor unit (PU), a DMA controller, an I/O interface to main memory (most likely off-chip DRAM), and an array of attached processing units (APU), all connected to an on-chip bus. APUs are functionally equivalent to processor cores, because each APU has its own register file, local memory, and function units, and APUs can operate independently of each other. Although the number of APUs in a Cell processor may vary, the patent says the “preferred” PE configuration has eight APUs. Some PEs may have specialized APUs, such as pixel-processing engines or 3D-graphics shaders. The PU controls the APUs by scheduling and dispatching instructions and data.

APUs are not coprocessors, the patent insists, but they don’t conform to the PowerPC architecture, as we know it today. In the patent’s “preferred embodiment” of an APU, there are 128KB of SRAM for local memory, 128 registers (128 bits wide), four integer units (ALU), and four floating-point units (FPU). Other combinations of function units are possible. As Figure 3 shows, all eight function units in an APU appear to share the same unified register file. In contrast, existing 32-bit PowerPC chips have the usual RISC complement of 32 integer registers (32 bits wide) and 32 floating-point registers (64 bits wide). A few 64-bit PowerPC chips have 64-bit integer registers. (See *MPR 10/28/02-02*, “IBM Trims Power4, Adds AltiVec.”) Even with AltiVec extensions, the PowerPC’s vector register file is only

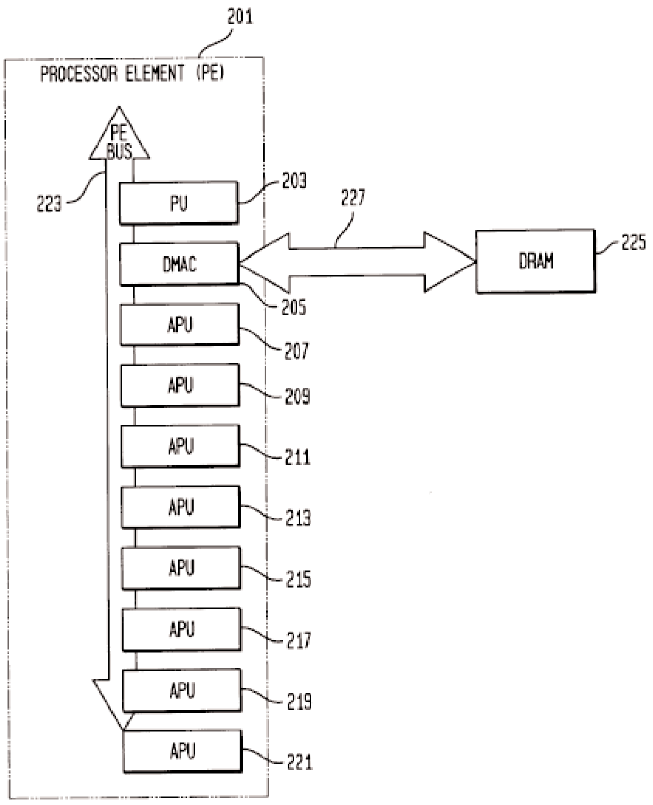

Figure 2. A processor element (PE) is the basic building block of a Cell processor. In this figure from the ’734 patent, a typical PE has a controller unit (the processor unit, or PU), a DMA controller, a DRAM interface, and eight attached processing units (APU). The highly capable APUs are functionally equivalent to processor cores in other multicore processors. Cell processors may have different numbers of PEs and various configurations of APUs.

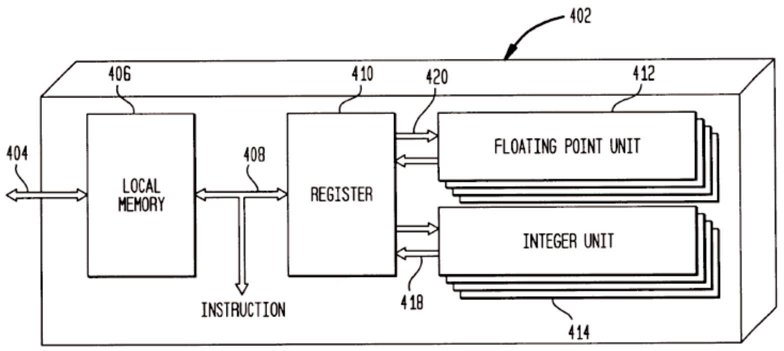

Figure 3. Inside look at an APU, the functional equivalent of a processor core. This figure, from the ‘734 patent, illustrates what the patent calls the “preferred embodiment” of an APU. It has four ALUs and four FPUs, all apparently sharing the same 128-entry, 128-bit-wide register file. Note that register porting allows each group of function units to fetch three 128-bit operands from the registers per clock cycle and return one 128-bit result per cycle. The instruction path is 256 bits wide, implying that an APU can fetch and execute eight 32-bit instructions per cycle—one for each of the eight function units. Local memory (128KB of SRAM, in this configuration) stores program code and data but isn’t a conventional fill-and-spill cache.

32 entries of 128 bits. (See *MPR 5/11/98-01*, "AltiVec Vectorizes PowerPC.") Perhaps the '734 patent describes a new extension like Intel's SSE2, which also has 128-bit registers. Certainly, a larger complement of wider registers would allow Cell processors to execute more SIMD instructions at the same time. The patent suggests that APUs are optimized for SIMD processing.

In a surprising moment of specificity, the patent estimates an APU's performance: 32 billion integer operations per second (BOPS) and 32 gigaFLOPS. Those numbers assume the "preferred embodiment" of an APU with four ALUs and four FPUs. However, the patent doesn't specify clock frequency or data types. Assuming 32-bit integer and 32-bit single-precision floating-point instructions (the most common data types for a game program), a SIMD instruction in each function unit could operate on four operands in a 128-bit register. Therefore, theoretical peak performance for the eight-way APU would be 16 integer SIMD operations and 16 floating-point SIMD operations per clock cycle.

A hypothetical Cell processor with eight of these APUs could achieve 32 BOPS and 32 gigaFLOPS at only 250MHz. Using regular instructions instead of SIMD, it could achieve the same performance at only 1GHz. And remember, that's for a Cell processor having only one PE of eight APUs. If a typical Cell processor has four PEs, as the patent suggests, multiply all those performance numbers by four. A higher-end Cell processor with eight PEs could theoretically deliver peak SIMD performance of one teraflops at a conservative clock frequency of 1GHz. A multiprocessor system could deliver supercomputer-class performance in a desktop-size box.

Data movement will be critical in a chip that has such vast processing resources. According to the '734 patent, the preferred configuration of a PE has a 1,024-bit-wide local bus. The bus may be implemented as a conventional on-chip bus (e.g., IBM's CoreConnect), a crossbar switch, or a packet-switched network. All the PE local buses tie into a global on-chip bus with interfaces to main memory, but the patent doesn't describe the memory interface. The patent does note that a PU or APU could perform its own memory transactions without help from a global bus interface unit.

Software Cells Carry Code and Data

Each PE has a PU controller for running trusted software, such as the operating system. In addition to dispatching instructions and data to local APUs, the PU apparently plays a role in routing software cells to other PEs for distributed processing—whether the other PEs are on the same chip, on another Cell processor in the same system, or located remotely on a network. Presumably, the PU dispatches a software cell to another PE when the local PE lacks enough processing resources to handle the workload at that moment.

There must be some mechanism allowing a PU to judge when it's faster to outsource the execution of a software cell to another PE instead of waiting for resources to become available in the local PE, but the '734 patent doesn't discuss it. To make distributed processing across a network practical, such a mechanism would also need an understanding of network transfer latencies. Because those latencies can vary greatly, dispatching software cells outside a system would seem impractical for tasks requiring isochronous or deterministic processing.

Inside the software cells are "apulets," or APU applets. Unlike Java applets and other miniapplications, apulets aren't necessarily self-contained programs. Apulets may have only enough instructions and data to execute part of a program. They appear to be more like the serialized objects in object-oriented programming languages—packaged subroutines of instructions and related data. PUs control apulet execution by issuing commands through the local DMA controller. Those commands, called APU remote procedure calls (ARPC), load an apulet from main memory into the fast local memory of an APU. Another ARPC, called a kick command, initiates execution. Apulets always execute in an APU's local memory, not directly from main memory.

Apulets are highly independent. Each has its own program counter and stack frame. A special header in the software cell carries critical information about the apulet, including a unique global ID, the type of APU that can execute the apulet, the minimum number of APUs required to execute the apulet in a timely fashion, and the amount of memory the apulet needs. If the apulet requires sequential execution in multiple APUs—as might be the case for streaming-media apulets—the header also carries the global ID of the most recent APU to execute the apulet. Cell processors generate global IDs at runtime by adding a date/time stamp to the ID of the PE or APU where the apulet originated. If the apulet must traverse a network, an outer wrapper contains the network addresses (e.g., TCP/IP addresses) of the source and destination PEs and APUs. There's also a "reply-to" address—the network address of a PE or APU that can answer queries about the apulet and receive results of the apulet's remote execution.

Apulets would seem to be packages of program code and data that have no data dependencies with other apulets, or least with apulets in other software cells. Otherwise, it's difficult to understand how a system could synchronize their execution, especially if mutually dependent apulets were dispersed across a large network with indeterminate transfer latencies. Imagine an apulet running on your PDA that depends on a result coming from another apulet running on a computer in Norway.

Instead, Cell compilers will probably concentrate data dependencies within a single apulet or a group of apulets in a software cell. If an apulet must execute sequentially on multiple APUs—and especially if timing is critical, as with streaming media—the compiler will probably tag the apulet with a limiter restricting its mobility. Lacking such limits, the apulet might execute too remotely to satisfy the timing requirements of the application.

Playing In the Sandbox

Java applets—unlike full-fledged Java applications—run in a protected region of memory, called a security sandbox, that prevents them from inflicting intentional or unintentional damage on other parts of the host system. For instance, a Java applet normally can't access memory, mass storage, peripherals, or network connections outside its sandbox, unless someone changes the security settings. The '734 patent describes similar sandboxes even more flexible than Java's, because they can be any size—large enough to enclose multiple programs or small enough to isolate only part of a program.

Each PE in a Cell processor has its own block of shared DRAM, but that doesn't mean any APU in the PE can access all the shared memory at will. Mechanisms in both the processor and the DRAM restrict each APU or a group of APUs to its own memory sandbox. Sandboxes have obvious security benefits, because they limit the damage a malicious or buggy program can wreak on a system. But sandboxes also provide hardware-level support for protecting copyrighted content with DRM.

For example, with today's Windows PCs, it's possible to capture a screen-resolution image of a copyrighted photograph by pressing the PrtScn (Print Screen) key. Windows copies the image onto the clipboard, from which one can paste the photograph into any image editor. Likewise, there are utility programs for capturing audio and video streams from the Internet and saving them as files, which can be burned onto audio CDs or DVDs. All these functions rely on the ability of the operating system or utility to freely access the video frame buffer or other regions of memory where the content temporarily resides. Sandboxed memory prevents this kind of copying, because only the program authorized to use the content can access those regions of memory. And because Cell implements the access controls at the hardware level, rogue programs cannot easily subvert them.

Figure 4, from the '734 patent, shows what the patent calls the "preferred embodiment" of shared memory attached to each PE of a Cell processor. Each of the eight APUs in this PE has exclusive access to eight banks of DRAM. Each bank is 1MB, so each APU has 8MB, and the total shared memory is 64MB. The patent envisions the smallest addressable location of DRAM as 1,024 bits, although it also suggests an alternative configuration interleaving 512-bit blocks of memory between two banks for faster access. Of course, the memory configuration of a hand-held game machine would differ considerably from that of a server, so the addressable memory locations and banks can be of any size. Likewise, a sandbox assigned to an APU can be any size.

The patent describes two mechanisms for controlling memory access: full/empty (F/E) bits and sandbox control keys. F/E bits maintain memory coherency; control keys enforce access privileges. Each addressable memory location (1,024 bits of DRAM in the "preferred embodiment") has an entry in a lookup table stored in memory. If the F/E bit for a memory location is set, the data at that location is current. If the F/E bit for a memory location is cleared, the data is currently in use by an APU, so it's unavailable to other APUs. Additional entries in the lookup table can store the ID of the APU requesting the data and the address in the APU's local memory, where the data should be copied when it's available. If an APU requests data from a memory location whose F/E bit is set, the lookup table stores the APU's ID and the local memory address until the F/E bit is cleared. This mechanism maintains data coherency in the PE's shared memory.

Another lookup table in DRAM stores a "busy bit" for each memory address in an APU's local memory. If the busy bit for a particular memory location is set, that location is available only for specific data retrieved from DRAM. If the busy bit is cleared, the associated memory location in local storage can hold any data fetched from DRAM. This mechanism is similar to cache locking—it allows a program to reserve blocks of local memory for specific purposes. For example, an APU could reserve enough local memory to decode a media stream, ensuring that other data fetched from memory won't crowd into the "busy" memory block.

Controlling Access to Sandboxes

According to the '734 patent, the PU and DMA controller of each PE jointly supervise access to the sandboxes assigned to

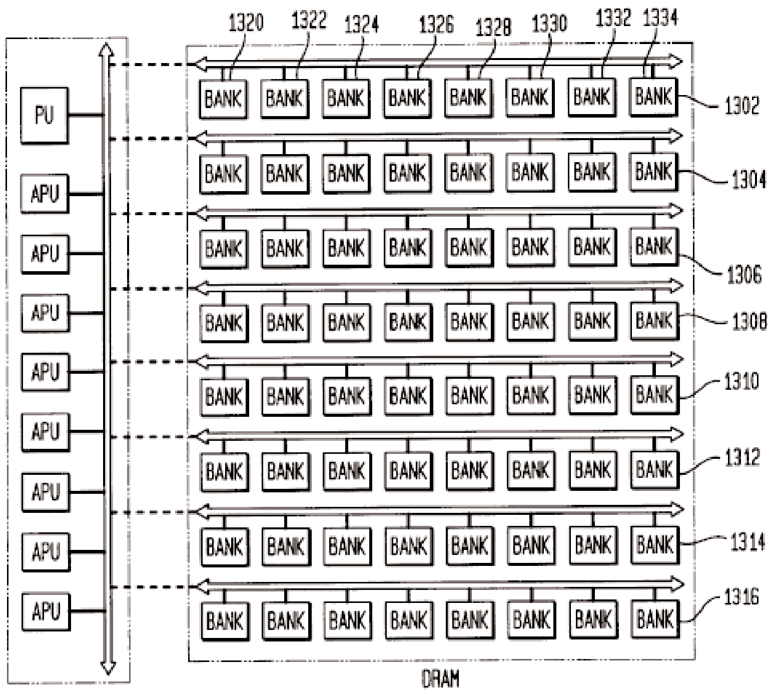

Figure 4. The shared DRAM attached to each PE of a Cell processor is divided into protected regions of memory called sandboxes. Each APU has exclusive access to its own sandbox, which strengthens system security and supports DRM for copyrighted data. Sandboxes can be any size—part of a bank of DRAM, or multiple banks of DRAM. The '734 patent suggests that shared memory for a typical PE might have 64 one-megabyte banks of 1,024-bit addressable DRAM, but actual configurations will vary, according to the type of system. Source: U.S. patent 6,809,734

the PE. The PU builds and maintains a control table with entries for each APU ID, a control key for each APU, and a key mask for each control key. For efficiency, the PU will probably store the control table in fast SRAM inside the DMA controller, not in the PU itself, because every main-memory access requires a control-table lookup.

In DRAM, another lookup table stores a memory-access key for every addressable location in main memory. This sandbox table is separate from the previously described DRAM lookup table that stores F/E bits for memory coherency. When an APU issues a memory command to the DMA controller, the controller compares the APU's key in the control-lookup table to the memory-access key in the sandbox table. If the keys match, the APU can access the target memory location. Otherwise, the processor returns a memory exception, which the operating system can handle appropriately.

For added flexibility, the key masks stored in the control table allow wild-card matching. When a bit in the mask is set, the corresponding bit position in the access keys can be either 0 or 1. For example, if the sandbox key for a particular memory location is 1010, and the key mask is 0001, an APU can access that memory location with a key of 1010 or 1011. If the key mask is 0000, the APU's key must match the sandbox key exactly.

If the final Cell architecture conforms to the descriptions in the '734 patent, it's hard to avoid the conclusion that Cell processors will have an extraordinarily secure but cumbersome memory model. For each main-memory access, the processor would have to consult four lookup tables: the memory-coherency table, to see if the F/E bit for the target memory location is set or cleared; the "busy bit" table, to see if the destination in local memory has permission to read data from the target location in main memory; the key-control table, to find the control key and mask for the target memory location; and the sandbox table, to match the control key and mask with the sandbox key. Three of those tables are in DRAM, which implies slow off-chip memory references; the other table is in the DMA controller's SRAM. In some cases, the delays caused by the table lookups might eat more clock cycles than reading or writing the actual data. The patent hints that some keys might unlock multiple memory locations or sandboxes, perhaps granting blanket permission for a rapid series of accesses, within certain bounds.

In addition to the overhead of the memory model, there's the extra baggage that software cells must carry during their journeys, either on chip or off chip. Apulets are the payload, but a software cell must also carry various ID numbers, program counters, and other state information for each apulet. This baggage is separate from any wrappers required for higher-level bus or network protocols. Figure 5 shows the format of a software cell.

The patent doesn't specify sizes for all the data structures in a software cell. Payloads, of course, will vary widely, depending on their function. Overhead is probably a few hundred bytes or perhaps a few kilobytes, depending on how much state information travels with a cell; the patent refers briefly to stack frames and other data structures. To allow distributed processing anywhere on a network, a cell would have to carry enough state information to be self-sufficient. Cell processors will probably have special logic for parsing the headers and metadata in software cells, much as network processors have special logic for parsing packet headers.

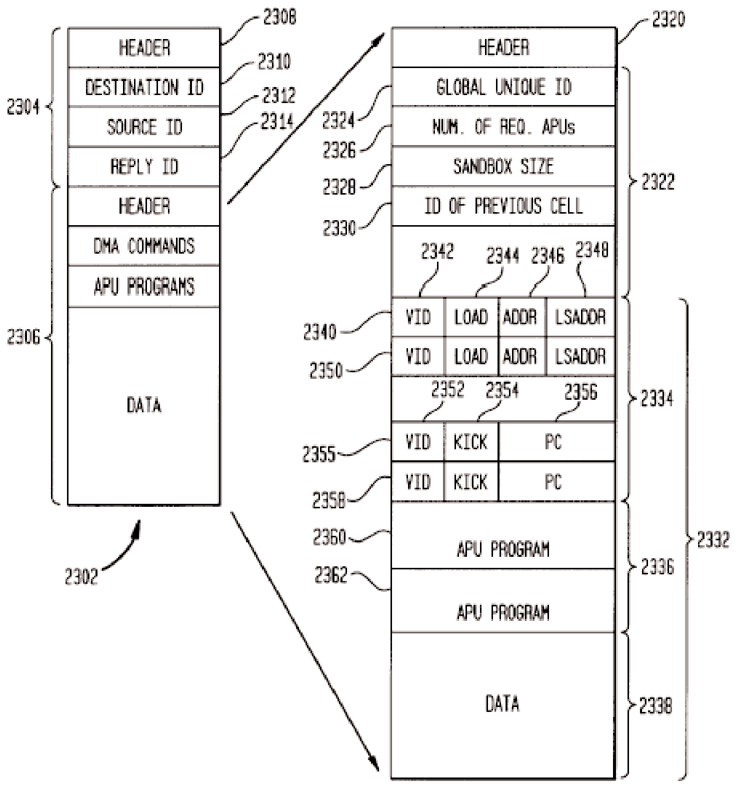

Figure 5. The Cell architecture's "software cells" have a complex format, as this figure from the '734 patent illustrates. At left is a higher-level view of the cell with a packet wrapper (denoted by bracket 2304), which allows it to travel anywhere on a network of Cell systems. At right is an expanded view of the cell's inner packet (denoted by bracket 2306). The inner packet contains a header, the cell's unique global ID, the number of APUs required to execute the apulets in the cell's payload, the size of the security sandbox allocated for the apulets, the global ID of the previous cell in this code sequence, a group of virtual IDs and memory addresses associated with APU local memory, "kick" commands to initiate apulet execution on appropriate APUs, program counters for the apulets, and finally the apulets themselves, with their program instructions and data.

Dynamically Reconfigurable Pipelines

The '734 patent claims the invention of dynamically reconfigurable execution pipelines. Hypothetically, a Cell processor can rearrange these pipelines at run time in response to software demands. Other processors have a fixed number of fixed-length pipelines, although the number of pipe stages can vary according to the type of data being processed. (For example, FPU pipelines are usually deeper than ALU pipelines, because they require more logic.) From our reading of the ’734 patent, it appears a Cell processor can dynamically allocate one or more APUs to form a temporary pipeline dedicated to executing a specific instruction stream. That instruction stream may be a program or part of a program.

Curiously, the patent describes a mechanism for automatically forwarding data from one APU to another by using the DMA controller and main memory. This is a clue that the reconfigurable pipelines are larger in scope than the instruction pipelines commonly associated with microprocessors. Normally, a processor uses latched registers to forward data from one pipe stage to another. Forwarding data through memory—especially off-chip main memory—would be a ridiculously slow detour. Therefore, we think the reconfigurable pipelines the patent describes aren’t the same as conventional microprocessor pipelines. More likely, they are ad hoc groups of APUs (with their function units pipelined in the conventional sense), organized on the fly to handle certain tasks.

In another departure from convention, Cell can dedicate a pipeline to a single task, locking out all other tasks. Other microprocessors always make all their processing resources available to whatever task is actively executing. For instance, a three-way superscalar processor always allows the task in the active context to use all three pipelines, along with their associated function units and registers. If a higher-priority task needs CPU time, an interrupt triggers a context switch, which flushes the current task out of the pipelines and transfers control to the new task. The new task enjoys the same exclusive access to the pipelines and resources that the previous task did. A recent variation of this model is simultaneous multithreading, such as Intel’s Hyper-Threading, which allows instruction streams from two or more contexts to simultaneously share a pipeline.

Cell processors appear to follow a radically different model. At run time, in response to software demands, the processor can dedicate one or more APUs to a particular task. This “pipeline” is the high-level *über* pipeline of pipelined function units (the APUs). The ’734 patent says the pipeline remains dedicated to its task even when it’s not busy. During idle periods, the pipeline enters what the patent calls a “reserved state,” preventing other tasks from preempting its resources. This model guarantees adequate processing resources will always be available to a particular task at a moment’s notice.

MPR is unaware of any other microprocessor that can dedicate its processing resources to a particular task in this way. The closest example might be cache locking, which fences off part of an L1 cache to prevent another task (or another routine within the same task) from flushing out vital instructions and data. However, other processors cannot reserve their pipelines or function units for a given task, as Cell can.

In the past, locking down part of a processor in this way was considered wasteful and inefficient, because those resources would be unavailable to other tasks, even when the resources weren’t busy. Cell is designed for an era in which microprocessors are so rich in resources, they can afford to dedicate some of their wealth to running the most important tasks.

Absolute Timers and Spontaneous NOPs

Several features of the Cell architecture are designed to guarantee performance to programs that need it. Among those features are the reconfigurable pipelines, the ability to dedicate processing resources to specific tasks, and the option of distributing software cells to other Cell processors. These are particularly useful features for a processor intended to handle streaming media over broadband networks. The ’734 patent specifically mentions MPEG, ATRAC (Sony’s proprietary

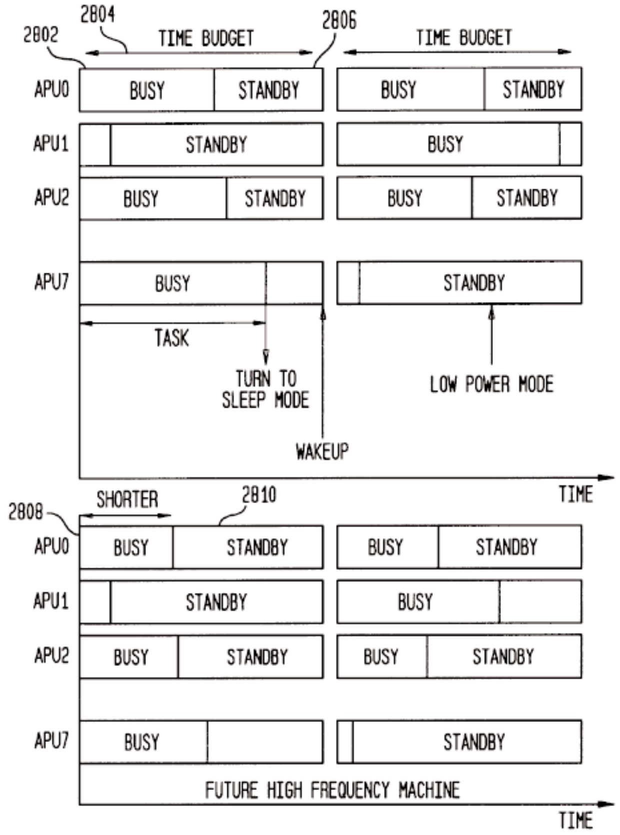

Figure 6. This figure from the ’734 patent illustrates the way a slower Cell processor (top) and a faster Cell processor (bottom) can allocate “time budgets” in their APUs to execute time-critical tasks in the same amount of real time. For example, APU0 in the faster processor (indicated by callout number 2808) spends less time in busy mode and more time in standby mode than does APU0 in the slower processor (indicated by callout number 2802). An absolute timer running faster than the processor’s main clock provides a consistent reference signal for synchronization. As noted at the bottom of the figure, this feature would allow future Cell processors to run time-critical software written for slower Cell processors—especially useful for keeping games playable on future generations of game machines.

For More Information

U.S. patents 6,809,734 and 6,526,491 are available from the U.S. Patent and Trademark Office website. Enter the patent number in the search field on this web page:

http://164.195.100.11/netahtml/srchnum.htm

Four similar U.S. patent applications filed by the Cell partners are pending. The applications are number 20030229765, "Memory Protection System and Method for Computer Architecture for Broadband Networks," filed on December 11, 2003; number 20020156993, "Processing Modules for Computer Architecture for Broadband Networks," filed on October 24, 2002; number 20020138707, "System and Method for Data Synchronization for a Computer Architecture for Broadband Networks," filed on September 26, 2002; and number 20020138637, "Computer Architecture and Software Cells for Broadband Networks," filed on September 26, 2002.

IBM is scheduled to present several papers about Cell at the International Solid-State Circuits Conference (ISSCC) in San Francisco on February 6–10, 2005. The advance program is available from the ISSCC website:

www.isscc.org/isscc/2005/ap/

IBM's website has a brief description of the Cell project:

www-1.ibm.com/businesscenter/venturedevelopment/us/en/featurearticle/gcl_xmlid/8649/nav_id/emerging

Sony's website has an announcement of Cell-based workstations:

www.us.playstation.com/pressreleases.aspx?id=208&op=print

audio codec), and TCP/IP. Audio and video streams flowing over networks need isochronous execution to avoid hiccups, and gamers expect smooth graphics.

Two additional features of Cell also guarantee certain levels of performance: absolute timers and automatically generated NOPs. Interestingly, the patent describes both features in the context of *reducing* performance. Because different Cell processors can have vastly different capabilities, they need a way to execute programs written for slower Cell processors without running the programs too quickly to be usable. This is particularly important for games. It may also allow future generations of Cell-based game machines and other systems to run software written for earlier generations.

Absolute timers sound much like real-time clocks, because they provide a clock reference independent of the clock signal driving the processor's logic. One difference, according to the patent, is that absolute timers provide a reference signal faster than the processor's fastest clock. A program can use the accelerated heartbeat of an absolute timer to create "time budgets" for tasks having critical execution times. APUs can synchronize execution to the absolute timer by allocating clock cycles to busy modes and standby modes in any combination that satisfies the time budget. Furthermore, each APU in a PE can synchronize execution independently of other APUs. Figure 6 shows how two different Cell processors can juggle their time budgets to guarantee the same level of performance to time-critical tasks.

The '734 patent describes a novel alternative method for synchronizing execution if absolute timers aren't available, or perhaps if the software doesn't specify time budgets. In particular, the patent says this alternative method is useful for coordinating parallel execution on a Cell processor running at a faster clock speed than the processor for which the programmer wrote the software. At run time, the PU controller or a designated APU can analyze an instruction stream and automatically insert NOP instructions. The NOPs consume enough CPU cycles to reduce execution speed to the desired rate.

Of course, programmers have been writing do-nothing timing loops for years, but this is the first time we've seen a microprocessor that can automatically modify an instruction stream at run time to deliberately waste clock cycles. The closest comparison might be the branch-delay slots in many ISAs, which sometimes mandate a NOP immediately after a branch instruction, just to give the processor enough time to calculate the branch-target address. But in those cases, it's the programmer or compiler that pads the code with NOPs at design time—quite different from a processor that inserts its own NOPs at runtime. The Cell architecture introduces a whole new meaning to the term "self-modifying code."

Don't Jump to Conclusions

It's worthwhile repeating that IBM, Sony, and Toshiba filed the '734 patent almost four years ago, and much may have changed since then. Also, the patent's numerous references to a "preferred embodiment" of a Cell processor may be little more than a CPU architect's pipe dream. *MPR* considers the patent an outline of a general architecture, not a blueprint of a specific microprocessor or microarchitecture.

Nevertheless, the patent contains more information about Cell than everything else leaked in the past four years put together. The unifying theme appears to be the software cells, with their payloads of apulets and freedom to roam far and wide. It's an innovation that now seems inevitable. With search engines like Google, it's sometimes faster to find something on a web server in another hemisphere than to locate a file on one's own hard drive. Cell is designed to harness the ubiquity and speed of broadband networks for everyday distributed processing. Wide-area clustering and grid computing, which today require special software and a great deal of planning, will probably be second nature to Cell systems.

Another vital part of Cell is security, implemented at the lowest hardware level. This, too, is inevitable. It's absurd that a moderately knowledgeable teenage hacker can disrupt the networks of multinational corporations, costing millions of dollars in lost productivity. Only an architecture secure from the ground up can offer the protection sorely needed. Everyone is working on the security problem, so nothing about Cell's solution is particularly surprising—but it's certainly welcome. Less welcome, for some users, will be the equally tight DRM inherent in any system-wide security scheme. Copyrighted content will be more difficult to share, but this, too, seems inevitable.

A common vein running through the Cell architecture is the lavish use of resources. The smallest Cell processor the patent envisions would have one PE with eight APUs, each with four ALUs and four FPUs. That's like an eight-core multiprocessor chip with 64 superscalar pipelines. A "typical" Cell chip with four PEs would be like a 32-core, 256-way processor. And the patent mentions much larger Cell processors for workstations and servers.

Consider Cell's practice of temporarily reserving pipelines for certain tasks. It's the opposite of Hyper-Threading, which strives to squeeze out every drop of performance by always sharing the processor's resources. Think about the overhead of Cell's security and memory-access models. Look at the extra baggage of metadata carried by every software cell. And don't forget the time-budget wait states and spontaneously generated NOPs that burn unwanted CPU cycles as if they were excess calories. The Cell architecture described in the '734 patent is designed on a grand scale for an age of big transistor budgets. If it is to have any future in low-power handheld systems, that extravagance must be pared back.

At Fall Processor Forum in October, a running joke throughout the conference was a question posed repeatedly to chip designers: "What will you do with a billion transistors?" Everyone had a different answer. Now we have strong hints of the answer from IBM, Sony, and Toshiba: Cell. ◇