The PowerPC Architecture:

A Programmer's View

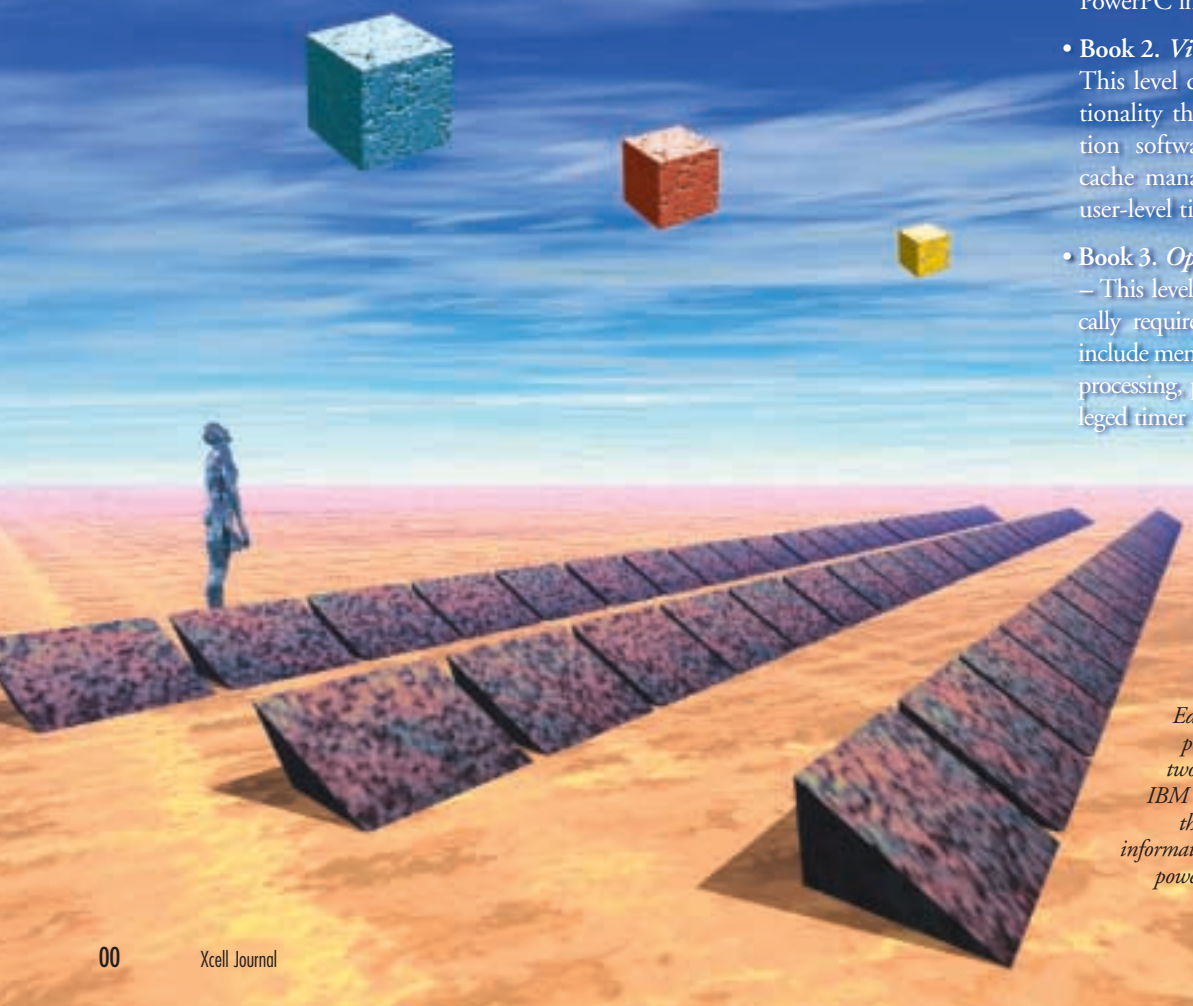
An introduction to the PowerPC programming model.

by Anthony Marsala
IBM

The PowerPC™ Architecture is a Reduced Instruction Set Computer (RISC) architecture, with over two hundred defined instructions. PowerPC is RISC in that most instructions execute in a single-cycle and typically perform a single operation (such as loading storage to a register, or storing a register to memory). This article will focus solely on 32-bit implementations, which are the most widely available today.

The PowerPC architecture employs a layered approach, in that it is broken up into three levels or "books". By segmenting the architecture in this way, code compatibility can be maintained across implementations while leaving room for implementations to choose levels of complexity for price/performances tradeoffs. The three levels are broken up from the most general and common across implementations to the most operating system specific. The levels are:

- **Book 1.** ***User Instruction Set Architecture*** – This level defines the base set of instructions and registers that should be common to all PowerPC implementations.
- **Book 2.** ***Virtual Environment Architecture*** – This level defines additional user-level functionality that is outside the normal application software requirements. Areas include cache management, atomic operations, and user-level timer support.
- **Book 3.** ***Operating Environment Architecture*** – This level defines privileged operations typically required by an operating system. Areas include memory management, exception vector processing, privileged register access, and privileged timer access.



Editor's note: This article is reprinted with permission from IBM. It was originally a two-part series that ran in the April, 2001, IBM PowerPC Processor News. You can view the original articles, and find other useful information at: www-3.ibm.com/chips/products/powerpc/newsletter/apr2001/design-h-t.html.

Deviations from the original PowerPC Architecture offer flexibility to allow for enhancements that may come over time. In addition, IBM has defined its own Virtual Environment and Operating Environment levels for its PowerPC 400 family of embedded controllers.

Book E – A New Definition

A new PowerPC architecture update has been developed. Called "Book E", it combines the original three architecture levels into one new specification. This new specification also streamlines the definition of 64-bit implementations and eliminates non-substantive differences between IBM and Motorola implementations. The new standard maintains 100% code compatibility with Book 1 instructions and registers, while formally defining software-based memory management, a two-level interrupt hierarchy, and user-extendible instruction space for auxiliary processors. All of these enhancements address the needs of embedded systems.

To distinguish between the original architecture, the IBM embedded definitions, and Book E, the original architecture will be referred to as the "classic" architecture for the remainder of this article.

Storage Model

The 32-bit PowerPC architecture has native support for byte, halfword (16-bits), and word (32-bit) data types. Also, PowerPC implementations can handle string operations for multi-byte strings up to 128 bytes in length. The 32-bit PowerPC implementations support a 4 GB address space (2^{32}). All storage is byte addressable. For misaligned data accesses, alignment support varies by product family, with some taking exceptions and others handling the access through multiple operations in hardware.

Endianness

Classic PowerPC and the IBM PowerPC 400 family are primarily big-endian machines, meaning that for halfword and word accesses, the most-significant byte (MSB) is at the lowest address. Support for little endian varies by implementation. Classic PowerPC had minimal support, while the 400 family provides more robust support for little endian storage.

Book E is endian-neutral, as the Book E architecture fully supports both accessing method.

Registers

Classic PowerPC registers are broken into two classes: special-purpose registers (SPRs) and general-purpose registers (GPRs). IBM's PowerPC 400 family and Book E also define a third class of registers, called device control registers (DCRs), to address peripheral registers outside of the processor core in an embedded controller implementation. The three classes are explained below.

SPRs

SPRs give status and control of resources within the processor core. Table 1 shows different types of SPRs and their purpose. Where a single register exists, the SPR name is listed in parenthesis.

Supervisor vs User-Mode SPRs

When the processor is first initialized, it is in supervisor (also called privileged) mode. In this mode, all processor resources, including registers and instructions, are accessible. The processor can limit access to certain privileged registers and instructions by placing itself in user (also called problem-state) mode. This protection limits application code from being able to modify global and sensitive resources, such as the caches, memory management system, and timers. Mode switching is controlled via the Machine State Register.

- The Instruction Address Register (IAR) is known to programmers as the program counter or instruction pointer. It is the address of the current instruction. This is really a pseudo-register, as it is not directly available to the user. The IAR is primarily used by debuggers to show the next instruction to be executed.
- The Processor Version Register (PVR) is useful for code common across multiple processors that must make decisions based on a specific processor.

User-Mode SPRs

There are four SPRs available in user-mode that are important to understand:

- The Link Register (LR) is a 32-bit register that contains the address to return to at the end of a function call. Certain branch instructions can automatically load the LR to the instruction following the branch.

SPR Register Type	Access Mode	Purpose
Count (CTR)	User	Branching and Loop Control
Link (LR)	User	Subroutine Branching
Save/Restore	Supervisor	Interrupt Context Save
Debug	Supervisor	On-chip Debug Capabilities
Timers	User (read) Supervisor (write)	Timing Facilities
Interrupt Vector Prefix	Supervisor	Locates Interrupt Addresses
Exception	Supervisor	State information where exceptions occur
Storage Attribute Control	Supervisor	Controls Storage Attributes (W,I,G,LE)
Processor Version (PVR)	Supervisor	Identifies PowerPC Implementation
General Purpose (SPRGn)	Supervisor	Used by Operating Systems
Integer Exception (XER)	User	Carry Bit, Overflow, String Lengths
MMU	Supervisor	Instruction/Data Translation Control

Table 1: SPR registers

The **blr** instruction moves the program counter to the address in the LR.

- The Fixed Point Exception Register (XER) contains overflow information from fixed point arithmetic operations. It also contains carry input to arithmetic operations and the number of bytes to transfer during load and store string instructions **lswx** and **stswx**.
- The Count Register (CTR) contains a loop counter that is decremented on certain branch operations. Also, the conditional branch instruction **bcctrx** branches to the value in the CTR.
- The Condition Register (CR) is grouped into eight fields, where each field is 4 bits that signify the result of an instruction's operation: Equal (EQ), Greater Than (GT), Less Than (LT), and Summary Overflow (SO).

Machine State Register (MSR)

MSRs represent the state of the machine. It is accessed only in supervisor mode, and contains the settings for things such as memory translation, cache settings, interrupt enables, user/privileged state, and floating point availability. Exact control bits vary by implementation.

The MSR does not readily fit into the SPR/DCR/GPR classification, as it contains its own pair of instructions (**mfmsr / mtmsr**) to read and write the contents of the MSR into a GPR.

DCRs

DCRs are similar to SPRs in that they give status and control information, but DCRs are for resources outside the processor core. DCRs allow for memory-mapped I/O control without using up portions of the 32-bit memory address space.

GPRs

The User Instruction Set Architecture (Level 1) specifies that all implementations have 32 GPRs (GPR0 - GPR31). GPRs are the source and destination of all fixed-point operations and load/store operations. They also provide access to SPRs and DCRs. They are all available for use in every instruction with one exception: In certain instructions, GPR0 simply means "0" and no lookup is done for GPR0's contents.

Instructions

Table 2 lists different instruction categories, and the types of instructions that exist in that category.

Instruction Category	Base Instructions
Data Movement	load, store
Arithmetic	add, subtract, negate, multiply, divide
Logical	and, or, xor, nand, nor, xnor, sign extension, count leading zeros, andc, orc
Comparison	compare algebraic, compare logical, compare immediate
Branch	branch, branch conditional, branch to LR, branch to CTR
Condition	rand, crnor, crxnor, crxor, crandc, crorc, crnand, cror, cr move
Rotate/Shift	rotate, rotate and mask, shift left, shift right
Cache Control	invalidate, touch, zero, flush, store, dcread, icread
Interrupt Control	write to external interrupt enable bit, move to/from machine state register, return from interrupt, return from critical interrupt
Processor Management	system call, synchronize, eieio, move to/from device control registers, move to/from special purpose registers, mtcrf, mfcr, mtmsr, mfmsr
MMU Control	TLB search, TLB read, TLB write, TLB invalidate all, TLB synchronize
MAC Unit	multiply low/high halfword and accumulate/subtract

Table 2 - Instruction categories

AND	OR	Exclusive OR	Rotate and Mask	Shift	Misc.
and	or	xor	rlwimi	slw	cntlzw
and.	or.	xor.	rlwimi.	slw.	cntlzw.
andi.	ori	xori	rlwinm	sraw	
andis.	oris	xoris	rlwinm.	sraw.	extsb
			rlwnm	srawi	extsb.
nand	nor	egv	rlwnm	srawi.	
nand.	nor.	egv.		srw	extsh
				srw.	extsh.
andc	orc				
andc.	orc.				

Table 3 - Power PC logical instructions

Deciphering an Instruction

For 32-bit implementations, all instructions are 32 bits (4 bytes) in length. Bit numberings for PowerPC are opposite of most other definitions; bit 0 is the most significant bit, and bit 31 is the least significant bit. Instructions are first decoded by the upper 6 bits, in a field called the primary opcode. The remaining 26 bits contain operands and/or reserved fields. Operands can be registers or immediate values.

Arithmetic Instructions

Many instructions exist for performing arithmetic operations, including add, subtract, negation, compare, multiply and divide. Many forms exist for immediate values, overflow detection, and carry in and out. Multiply and divide

instruction performance varies among implementations, as these are typically multi-cycle instructions.

Logical Instructions

Table 3 lists PowerPC logical instructions. Looking at the AND instruction, The "i" form means that a 16-bit immediate is used for the AND, the "is" form means that a 16-bit immediate is used in the upper 16-bits of the AND. For all "." forms, the CR[CR0] is updated as previously described. PowerPC has the ability to perform a 32-bit rotate-and-combine with a mask in a single cycle. In the miscellaneous column are instructions to count the leading zeros in a register, and sign extension instructions.

Load/Store Instructions

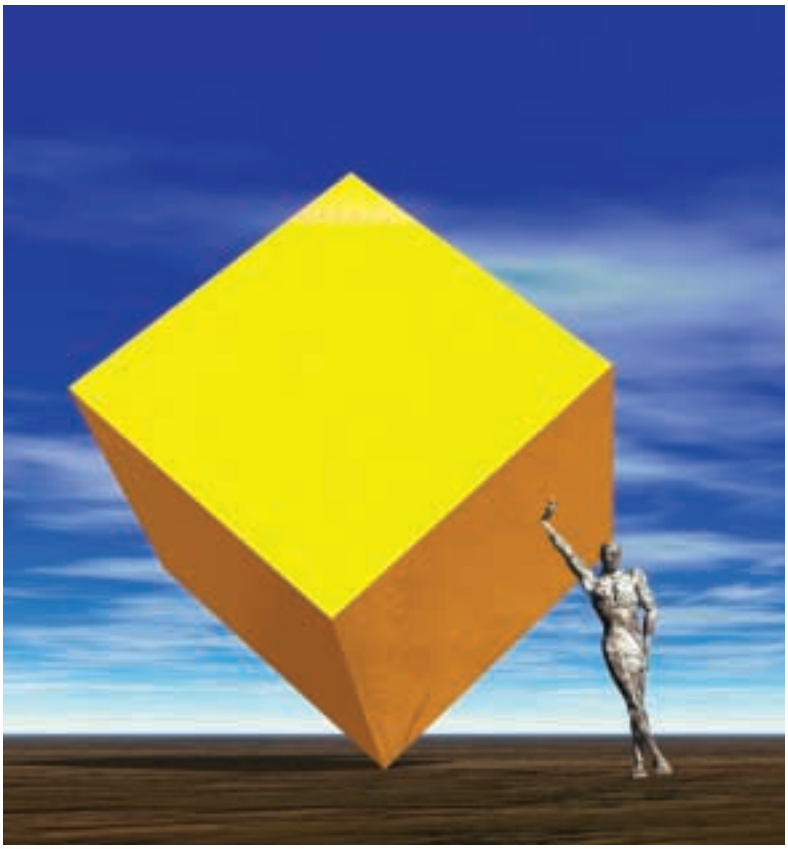
All loads/stores are performed using the GPRs. Instructions exist for byte, halfword, and word sizes. Special instructions include:

- Multiple-word load/stores (**lmw / stmw**), which can operate on up to 31, 32-bit words
- String instructions, which can operate on up to 128-byte strings
- Memory Synchronization instructions **lwarx** (Load Word and Reserve Indexed) and **stwcx.** (Store Word Conditional Index) are used to implement memory synchronization. lwarx performs a load and sets a reservation bit internal to the processor and hidden from the programming model. The associated store instruction **stwcx.** performs a conditional store only if the reservation bit is set and thereafter clears the reservation bit. CR[CR0]EQ is set to the state of the reservation bit at the start of the instruction so that software can determine if the write was successful.

Synchronization Instructions

Commonly misunderstood PowerPC instructions are those that perform synchronization. These instructions include:

- Enforce In/Order Execution of I/O (eieio) – This instruction is for data accesses to guarantee that loads and stores complete with respect to one another. Since PowerPC defines a weakly ordered storage model in which loads and stores can complete out of order, this instruction exists to guarantee ordering where necessary.



- Synchronize (sync) – This instruction guarantees that the preceding instructions complete before the sync completes. This instruction is useful for guaranteeing load/store access completion. For example, a sync may be used when writing memory mapped I/O registers to a slow device before making further access to the device.
- Instruction Synchronize (isync) – This instruction provides ordering for all effects of all instructions executed by the processor. It is used to synchronize the instruction context, such as memory translation, endianness, cache coherency, etc. Instruction pipelines are flushed when an isync is performed, and the next instruction is fetched in the new context. This instruction is useful for self-modifying code.

Memory Management

Memory management is used to translate logical (effective) addresses to physical (real) addresses. Memory management units (MMUs) are also used to control storage attributes, such as cacheability, cache write-though/write-back mode, memory coherency, and guardedness. There are two primary approaches; one defined by PowerPC classic in the 600/700 family of processors and another used by the 400 family and Book E specification. In both cases, the architecture defines a unified MMU, which has traditionally been implemented as independent instruction and data MMUs, enabled via the MSR [IR,DR] bits, respectively. Below is an overview of the two approaches.

PowerPC Classic MMU

The PowerPC Classic MMU was designed primarily for demand page operating systems such as UNIX or MacOS. There are two translation mechanisms, one for block address translation, and another for page tables. Block address translation is performed using eight pairs (upper and lower) of address translation registers, four for instruction addresses (IBATU/L 0-3), and four for data accesses (DBATU/L 0-3). The BAT registers define page sizes ranging from 128KB to 16MB.

For systems requiring more translations than are found in the allocated BAT registers, page table translation is provided. A 32-bit effective address is translated to a 52-bit virtual address, and is then translated into a physical address. One of 16 segment registers (SR0-

SR15) provide virtual address and protection information. Page Table Entries (PTEs) provided physical address and page protection information. The architecture allows for implementations to provide translation-lookaside buffers (TLBs) to speed the translation process, but does not define them. The page-tables are typically programmed by the operating system and their discussion is beyond the scope of this article.

400 Family/Book E MMU

The Book E carries on the idea of a flexible MMU structure for embedded systems. Page sizes are programmable; a page can be large (up to a terabyte in the Book E architecture) to simplify software and minimize the number of entries, or as small as 1KB, to avoid wasting memory space. In addition to normal protection and translation mechanisms, endianness is defined by a page attribute. TLB misses result in an exception; it is under software control to handle the page miss algorithm. A TLB search instruction, **tlbsx**, assists in searching the entire TLB array in a single cycle.

Interrupts

The PowerPC architecture provides a minimal hardware scheme for saving state on interrupts. The only registers that are saved are the IAR and MSR. Interrupt enable bits are disabled for the interrupt type that occurred in order to prevent a second interrupt from occurring before saving the context. Software must save all necessary registers – these typically include all user-mode registers and possibly certain supervisor mode SPRs. Exception-state saving is typically performed by an operating system, but note that for small exception vectors, time can be saved by only saving registers that would otherwise be corrupted. Operating systems must take a more universal approach and save all registers that may be necessary, even if some wind up not being touched by a particular exception handler.

PowerPC Classic Exception Vector Processing

A single interrupt hierarchy is defined. When an interrupt occurs, Save/Restore Register 0 (SRR0) is loaded with the address of where processing should resume after the exception, and the machine state register is saved to SRR1. SRR0 may be loaded with the current IAR or in some cases the next instruction. Interrupt vectors are located at either a high address (0xFFFn_nnnn if MSR[IP=1] or low address (0x000n_nnnn if MSR[IP=0]), depending on the instruction prefix bit in the MSR. The interrupt type determines the lower 5 bits of the vector. When processing is completed, an **rfi** instruction is executed to restore the IAR and MSR to the saved values in SRR0 and SRR1.

400 Family and Book E Exception Vector Processing

Both the IBM 400 family and Book E define a two-level interrupt hierarchy: a non-critical interrupt class, and a critical interrupt class. The non-critical class registers work as previously described for PowerPC classic. For critical interrupts, the IAR and MSR are saved to separate registers (SRR2 & SRR3, respectively for the 400 family, and CSSR0 & CSSR1 for Book E). When a critical exception is completed, an rfci instruction is executed to properly restore the machine. By having a dual-level interrupt scheme, non-critical interrupts can be more easily debugged. More than two sets of interrupt vectors are possible – for the 400 family, the upper 16 bits of the exception vector is contained in the Exception Vector Prefix Register (EVPR). For Book E, all 16 exceptions can have the upper half of the exception vector mapped to a different location through the use of 16 Interrupt Vector Prefix Registers (IVPR0-15).

Stack

The PowerPC architecture has no notion of a stack for local storage. There are no push or pop instructions and no dedicated stack pointer register defined by the architecture. However, there is a software standard used for C/C++ programs called the Embedded Application Binary Interface (EABI) which defines register and memory conventions for a stack. The EABI reserves GPR1 for a stack pointer, GPR3-GPR7 for function argument passing and GPR3 for function return values. Assembly language programs wishing to interface to C/C++ code must follow the same standards to preserve the conventions.

Caches

The PowerPC architecture contains cache management instructions for both user-level and supervisor-level cache accesses. Cache management instructions are found in Table 4 below.

Instruction	Mode	Implementation	Function
dcbf	User	All	Flush Data Cache Line
dcbi	Supervisor	All	Invalidate Data Cache Line
dcbst	User	All	Store Data Cache Line
dcbt	User	All	Touch Data Cache Line (for load)
dcbtst	User	All	Touch Data Cache Line (for store)
dcbz	User	All	Zero Data Cache Line
dccci	Supervisor	IBM 4xx	Data Cache Congruence Class Invalidate
icbi	User	All	Invalidate Instruction Cache Line
icbt	User	4xx / Book E	Touch Instruction Cache Line
iccci	Supervisor	IBM 4xx	Instruction Cache Congruence Class Invalidate

Table 4 - Cache management instructions

Care should be taken when porting cache manipulation code to a different PowerPC implementation. Although cache instructions may be common across different implementations, cache organization and size may likely change. For example, code that makes assumptions about the cache size to perform a flush may need to be modified for other cache sizes. Also, cache initialization may vary between implementations. Some provide hardware to automatically clear cache tags, while others require software looping to invalidate cache tags.

Self-Modifying Code

While it is not a recommended practice to write self-modifying code, sometimes it is absolutely necessary. The following sequence shows the instructions used to perform a code modification:

1. Store modified instruction.
2. Issue **dcbst** instruction to force new instruction to main store.
3. Issue **sync** instruction to ensure DCBST is completed.
4. Issue **icbi** instruction to invalidate instruction cache line.
5. Issue **isync** instruction to clear instruction pipeline.
6. It is now OK to execute the modified instruction.

Timers

Most implementations have provided a 64-bit timebase that is readable via two 32-bit registers. The amount the timer increments varies across families, as well as the SPR numbers and instructions to access the timebase. Therefore, care should be taken when porting timer code across implementations. Additional timers may also vary, but most provide at least one kind of decrementing programmable timer.

Book E Timers

Both the IBM 400 family and Book E define the following timers in addition to the timebase: a 32-bit programmable decrementer (DEC in Book E, PIT for the 400 family) with an auto-reload capability, a fixed-interval timer (FIT), and a watchdog timer (WDT) for system hang conditions.

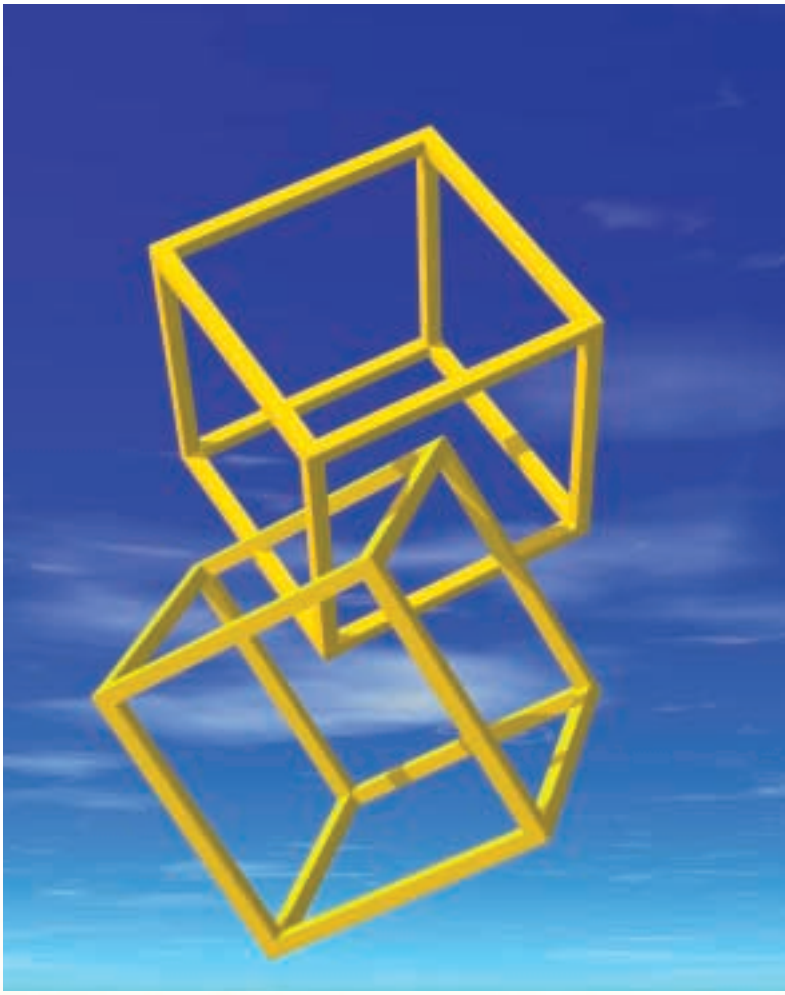
Debug Facilities

Debug facilities vary greatly between implementations. Original PowerPC 600 family parts had only one instruction address breakpoint. PowerPC 700 family parts have added a single data address breakpoint. PowerPC 400 family parts have much more robust debug capabilities, including multiple instruction address breakpoints, data address breakpoints, and data value compares. Other features may include breakpoint sequencing, counters, ranges, and trace capabilities.

Maintaining Code Compatibility

PowerPC users who expect to program for more than one implementation typically ask for tips on maintaining code compatibility. The following are some suggestions to help minimize porting problems:

- Use C code whenever possible. Today's C compilers can produce code that is comparable in performance to hand-assembly coding in many cases. C code, being Book I code, will guarantee code portability.
- Also, try not to embed processor-specific assembly instructions in C, as they'll be harder to find. Separate processor-specific code that is known to contain device dependent registers or instructions. These are typically things like boot up sequences and device drivers, but also may include floating point code (including long long types). Keep them well documented as to assumptions and dependencies.
- Use the PVR, but only when appropriate. Common code across minor variations of implementations is good, and the PVR can be used for decision making. But in the case where major modifications are necessary (for example, 7xx versus 4xx MMU code), separate code bases are recommended.



Summary

This completes an introduction to the PowerPC programming model. IBM hopes you have found this of value, and that it adds to the success of your development programs. For more information, go to: *www-3.ibm.com/chips/products/powerpc/newsletter/*