Introduction to the PowerPC® Programming Model

ESC San Francisco 2004

Class 422

Paul Gramann
IBM Microelectronics
gramann@us.ibm.com

As CISC programmers transition from legacy processors to RISC architectures such as the PowerPC, they may be faced with a steep learning curve. This paper presents an overview of the PowerPC programming model for engineers and programmers new to the PowerPC architecture. It covers topics such as the PowerPC storage model, register set, instruction formats, memory management, interrupt handling, debug facilities, and provides suggestions for maintaining code compatibility across different PowerPC implementations.

For the purposes of this paper, little delineation is made between a processor's architecture, which defines the hardware resources of a processor, and the programming model, which is the programmer's view of how those hardware resources can be used.

1. HISTORY

The original PowerPC Architecture™ is a collaborative effort of IBM, Motorola, and Apple. It is derived from IBM's POWER architecture found in IBM's RS/6000® line of workstations of the early 1990s. Although initially designed for desktop applications, the PowerPC architecture's power and flexibility make it very appropriate for embedded applications. Since their introduction, PowerPC processors have grown into many embedded systems including networking, RAID controllers, set-top boxes, automotive, cellular and gaming applications.

2. ARCHITECTURE OVERVIEW

The PowerPC architecture is a Reduced Instruction Set Computer (RISC) architecture but the "RISC" moniker is a bit of a misnomer; there are over two hundred instructions defined. PowerPC is RISC in that all of the instructions are a fixed width and most execute in a single-cycle, typically performing a single operation (such as loading a register from memory, or storing a register to memory). In addition, ALU instructions only operate on register values. While PowerPC is a 64-bit architecture, this paper will focus on the 32-bit implementations that are the most widely available today.

2.1 Layered Approach

The architecture defined by the IBM, Motorola, and Apple alliance is broken up into three levels or "books".[1] By layering 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. All application code (C/C++ compiler derived) should produce user-instruction set code. This level defines optional floating-point arithmetic support and optional 64-bit mode support.

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.

2.2 Deviations from the original PowerPC Architecture

Specific implementations may also contain instructions and registers not necessarily defined in the architecture. This flexibility allows 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.

2.3 Book E – The Future

IBM and Motorola have developed a new architecture update for PowerPC that combines the original three architecture levels into one new specification called "Book E".[2] This new specification has also streamlined the definition of 64-bit implementations and eliminated 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 a user-extendible instruction space for auxiliary processors. All of these enhancements address the needs of embedded systems.

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

3. STORAGE MODEL

32-bit PowerPC has native support for byte, halfword (16-bits), and word (32-bit) data types. Processors that implement floating-point in hardware also support a doubleword (64-bit) data type. In addition, PowerPC defines string operations for multi-byte strings up to 128 bytes in length. 32-bit PowerPC implementations support a byte addressable 4 GB (2^{32}) address space. For misaligned data accesses, alignment support varies by product family, with some taking exceptions and others handling the access through multiple operations in hardware.

3.1 Endianness

Classic PowerPC is primarily big-endian, meaning that for multi-byte data types, the most-significant byte (MSB) is stored at the lowest address in memory. Support for little endian varies by implementation. Classic PowerPC had minimal support, while the 400 and 800 families provide more robust support for little endian storage.

Book E is endian-neutral, fully supporting both storage methods via an attribute in the MMU.

4. REGISTERS

Classic PowerPC registers are classified into general-purpose registers (GPRs), floating-point registers (FPRs), special-purpose registers (SPRs) and miscellaneous registers. IBM's 400 family and Book E also define an additional group of registers called device control registers (DCRs) which are peripheral registers outside of the processor core in an embedded controller implementation.

4.1 General Purpose Registers (GPRs)

Book 1 specifies that all implementations have 32 GPRs, GPR0 – GPR31. In the 32-bit PowerPC architecture each GPR is 32 bits wide. GPRs are the source and destination of all fixed-point operations and load/store operations. They also provide access to SPRs and DCRs. As their

name implies, they are all available for use in every fixed-point instruction with one exception: In certain instructions GPR0 simply means '0' and no lookup is done for GPR0's contents.

4.2 Floating-Point Registers (FPRs)

As with the GPRs, there are 32 FPRs, each being 64 bits wide. The FPRs are the source and destination of all floating-point operations and floating-point load/store operations. Similar to the GPRs, any FPR can be used during a floating-point operation. Note that the PowerPC architecture allows any architected facility or instruction to be implemented with assistance from software. Thus, processors that do not implement the PowerPC floating-point registers and instructions can emulate floating-point operations in software, usually via a compiler library.

4.3 Special Purpose Registers (SPRs) and Miscellaneous Registers

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

Register Type	Access Mode	Purpose
Program Flow Control	User	Branching, subroutine, loop control
Fixed-point Exception (XER)	User	Carry bit, overflow, string lengths
Floating-Point Status and Control (FPSCR)	User	Controls floating-point exceptions and records floating-point operation status
Processor Version (PVR)	Supervisor	Identifies PowerPC implementation
Machine State (MSR)	Supervisor	Global operational mode controls
MMU	Supervisor	Instruction/data translation control
Storage Attribute Control	Supervisor	Controls storage attributes (W,I,G,LE)
General Purpose (SPRGn)	Supervisor[1]	Used by operating systems
Interrupt / Exception	Supervisor	Interrupt context save, vector generation
Timers	User (read) Supervisor (write)	Timing facilities
Debug	Supervisor	On-chip debug capabilities

Table 1. SPR and Miscellaneous Registers

4.3.1 Supervisor versus 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, described in Section 4.3.4.

[1] Some implementations define SPRG registers that have user-mode access.

4.3.2 User-mode Special Purpose Registers

There are a few special purpose and miscellaneous registers that are available in user-mode that are important to understand:

4.3.2.1 Link Register (LR)

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

4.3.2.2 Count Register (CTR)

The count register contains a loop counter that is decremented on certain branch operations. Also, the conditional branch instruction **bcctr**x branches to the value in the CTR.

4.3.2.3 Condition Register (CR)

The condition register is a miscellaneous register that is grouped into eight fields, where each field is 4 bits that signify the following result of an instruction's operation:

- Less Than (LT) or Floating-Point Less Than (FL)
- Greater Than (GT) or Floating-Point Greater Than (FG)
- Equal (EQ) or Floating-Point Equal (FE)
- Summary Overflow (SO) or Floating-Point Unordered (FU)

There are condition logical instructions that perform Boolean logic on individual bits in the CR. In addition, the conditional branch instructions test specific bits in the CR when decided whether to branch or not.

4.3.2.4 Fixed Point Exception Register (XER)

The fixed-point exception register contains overflow information from fixed-point arithmetic operations. It also contains carry input to arithmetic operations and the number of bytes to transfer during the load and store string instructions **lswx** and **stswx**.

4.3.2.5 Floating-Point Status and Control Register (FPSCR)

The floating-point status and control register contains all floating-point exception status bits, exception summary bits, exception enable bits and rounding control bits.

4.3.2.6 Instruction Address Register (IAR)

The IAR is more readily known to programmers as the program counter or instruction pointer and represents the address of the current instruction. The IAR is really a pseudo-register, as it is not directly available to the user. The IAR is primarily used by debuggers to set and show the next instruction to be executed.

4.3.3 Processor Version Register (PVR)

PowerPC defines that a unique number be given to every PowerPC implementation. This number is contained in the supervisor read-only PVR. This register is useful for code common across multiple processors that must make decisions based on a specific processor.

4.3.4 Machine State Register (MSR)

The MSR represents 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.

4.4 DCRs

DCRs provide status and control of chip resources outside of the processor core. This register class is unique to the IBM 400 family and Book E. It allows designers to access I/O devices without using memory addresses within the normal 4GB address space and keeps potentially low latency accesses from disrupting traffic on the higher bandwidth processor busses.

5. INSTRUCTIONS

5.1 Instruction Categories

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
MAC Unit	multiply low/high halfword and accumulate/subtract
Logical	and, or, xor, nand, nor, xnor, sign extension, count leading zeros, andc, orc
Rotate/Shift	rotate, rotate and mask, shift left, shift right
Comparison	compare algebraic, compare logical, compare immediate
Floating-Point	Load, store, move, arithmetic, comparison
Condition Logical	rand, crnor, crxnor, crxor, crandc, crorc, crnand, cror, cr move
Branch	branch, branch conditional, branch to LR, branch to CTR
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

Table 2. Instruction Categories

5.2 Deciphering an instruction

All PowerPC 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. The upper 6 bits of every instruction define a field called the primary opcode. The remaining 26 bits contain operands and/or reserved fields. Operands can be registers or immediate values.

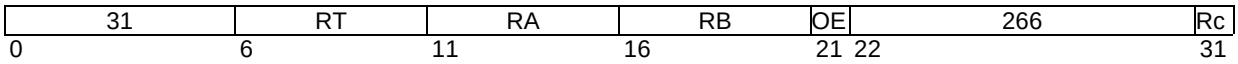
5.2.1 Add example

Consider the case of the **add** instruction, which is the first instruction alphabetically. A description of the **add** instruction may look like the following:

add

Add

add	RT, RA, RB	OE= 0, Rc= 0
add.	RT, RA, RB	OE= 0, Rc= 1
addo	RT, RA, RB	OE= 1, Rc= 0
addo.	RT, RA, RB	OE= 1, Rc= 1



(RT) ← (RA) + (RB)

The sum of the contents of register RA and the contents of register RB is placed into register RT.

Registers Altered

- RT
- CR[CR0]LT, GT, EQ, SO if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

In this sample page from a PowerPC processor's user manual, the instruction mnemonic is first listed, "add". Underneath it is a description of its function (in this case, it's the same as the mnemonic, "Add"). The four lines that follow are different variations of the same instruction. In this case, the variations come from two bits, the OE or overflow bit, and the Rc, or record bit. Below the instructions is a picture of the instruction layout. "31" is the decimal notation for the primary opcode in bits 0:5 of the instruction. For this primary opcode, a secondary opcode is also used, "266". RT is the destination register (the register format always lists the destination register followed by the source registers) and RA and RB are the source operand registers. The next line shows the pseudo-code for the operation. In this case, the sum of RA and RB is placed into RT. Under "Registers Altered", note that the record bit form updates CR0 and that the overflow form updates XER. Finally, it is noted that the instruction is part of the Book 1, or User Level Instruction Set, and is therefore common to all PowerPC implementations.

5.3 Load/Store Instructions

All integer or fixed-point loads/stores are performed using the GPRs. Instructions exist for byte, halfword, and word sizes. In addition, 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** and **stwcx.**

5.4 Fixed-Point Arithmetic Instructions

Many instructions exist for performing fixed-point 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. The IBM 400 family also includes several single cycle multiply accumulate instructions to assist in traditional DSP computations.

5.5 Logical Instructions

Table 3 lists logical operations for PowerPC. Looking at the **and** instruction, the "i" form means that a 16-bit immediate value is used for the instruction and the "is" form means that a 16-bit immediate value is used in the upper 16-bits of the instruction. For all "." forms, the CR[CR0] field 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 that count the leading zeros in a register and perform sign extensions.

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	eqv	rlwnm.	srawi.	
nand.	nor.	eqv.		srw	extsh
				srw.	extsh.
andc	orc				
andc.	orc.				

Table 3. PowerPC Logical Instructions

5.6 Floating-Point Arithmetic Instructions

Similar to the fixed-point instructions, the PowerPC architecture includes floating point instructions to perform arithmetic on the values contained in the floating-point registers. Floating-point operations include addition, subtraction, multiplication, division, square root, rounding, conversion, comparison as well as combinations of these operations. As with the fixed-point instructions, floating-point load and store instructions are used to copy floating-point values from memory into the floating-point registers and vice-versa.

5.7 Memory Synchronization Instructions

PowerPC does not include an atomic read-modify-write instruction. Instead, a pair of instructions, **lwarx** and **stwcx.**, are used to implement memory synchronization. **lwarx** (Load Word and Reserve Indexed) performs a load and sets a reservation bit internal to the processor and hidden from the programming model. The associated store instruction, **stwcx.** (Store Word Conditional Index), 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. The following code example shows how the **lwarx**/**stwcx.** pair can be used to perform a semaphore write.

```
loop: lwarx       # read semaphore from memory, set reservation bit
      "alter"     # change memory location as needed
      stwcx.      # attempt to store semaphore, reset reservation bit if
                  successful
      bne loop    # retry if interrupted by an asynchronous process
```

5.8 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. Book E has renamed this instruction **mbar** to imply that this instruction creates a barrier between memory accesses.

- 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 to ensure that the write completes before continuing to the next instruction. Book E has renamed this instruction **msync** to imply that this instruction synchronizes instructions with respect to a memory access.

- 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, as described in Section 8.1.1.

6. 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.[3] The EABI reserves GPR1 for a stack pointer. Compilers use the **stwu** instruction to store the contents of the GPR1 stack pointer and update it when creating a new stack frame. In the EABI GPR3-GPR10 are used for fixed-point function argument passing and GPR3 and GPR4 are used for function return values. Floating-point arguments are passed in FPR1-FPR8 and floating-point return values are stored in

FPR1. Assembly language programs wishing to interface to C/C++ code must follow the same standards to preserve the conventions.

7. INTERRUPTS

The 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, for small exception handlers, only saving registers that would otherwise be corrupted can save time. 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.

7.1 PowerPC Classic Exception Vector Processing

PowerPC classic defines a single interrupt hierarchy. When an interrupt occurs, Save/Restore Register 0 (SRR0) is loaded with the instruction address where processing should resume after the exception and the machine state register is saved in SRR1. SRR0 may be loaded with the current IAR or in some cases the next instruction depending on the interrupt type. The interrupt vectors are located at either a high address (0xFFFn_nnnn if MSR[IP=1] or a low address (0x000n_nnnn if MSR[IP=0]) depending on the instruction prefix bit in the MSR. The interrupt type determines the lower 20 bits of the exception vector (n_nnnn). When processing is completed, an **rfi** instruction is executed to restore the IAR and MSR to the saved values in SRR0 and SRR1.

7.2 400 Family and Book E Exception Vector Processing

Both the IBM 400 family and Book E also define a critical interrupt class. 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 restore the interrupted machine context. In addition, Book E defines a machine-check interrupt class that uses the MCSRR0 and MCSRR1 registers and the **rfmci** instruction to save and restore the machine context. This multi-level interrupt scheme with its separate save-restore registers allows critical class interrupts to safely preempt non—critical class interrupts.

In contrast to PowerPC classic, the upper 16 bits of all the exception vectors are specified in the Exception Vector Prefix Register (EVPR) for the IBM 400 family and in the Interrupt Vector Prefix Register (EVPR) for Book E. For the IBM 400 family the interrupt type defines the lower 16 bits of each exception vector, similar to classic PowerPC. In Book E, these lower address bits are specified through the use of the Interrupt Vector Offset Registers (IVPR0-15).

8. CACHES

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

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[2]	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 differ. For example, code that makes assumptions about the cache size to perform a flush may need to be modified for other cache sizes. Cache initialization may also vary between implementations. Some provide hardware to automatically clear cache tags while others require software looping to invalidate cache tags.

8.1.1 Self-Modifying Code

While it is not a recommended practice to write self-modifying code, sometimes it is 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
```

9. TIMERS

Most implementations have provided a 64-bit incrementing timebase register that is readable via two 32-bit registers. The timebase increment frequency is implementation dependant as are the SPR numbers and instructions used to access the timebase. Therefore, care should be taken when porting timer code across implementations. Additional timers also vary, but most implementations provide at least one kind of decrementing programmable timer.

9.1 Book E Timers

In addition to the timebase, both the IBM 400 family and Book E define 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) to recover from system hang conditions.

---

[2] icbt is privileged in IBM PowerPC 401™ and PowerPC 403™ implementations

10. 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 600/700/800 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.

10.1 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 (BAT) is performed using several pairs (upper and lower) of address translation registers for both instruction addresses (IBATU/L 0-n) and data accesses (DBATU/L 0-n). The number of BAT registers (n) is implementation dependent. 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 with fixed 4 Kbyte sized pages. A 32-bit effective address is translated to a 52-bit virtual address, and is then translated into a physical address. One of sixteen segment registers (SR0-SR15) provides virtual address and partial page protection information. The Page Table Entries (PTEs) provide physical address and the remaining 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 paper.

10.2 400 Family/Book E MMU

The IBM 400 family and Book E provides a more flexible MMU structure for embedded systems. Page sizes are programmable; a page can be as large as a terabyte to simplify software and minimize the number of entries or as small as 1KB to avoid wasting memory space. A process ID rather than a segment is create the virtual address. In addition to normal protection and translation mechanisms, endianness is defined by a page attribute. TLB misses result in an exception and software is used to handle the page miss algorithm. A TLB search instruction, **tlbsx**, is provided to quickly search the entire TLB array for an entry with a specific virtual address.

In Book E, translation is always enabled and the concept of a translation space has been added to differentiate the translation used in interrupt handlers verses non-interrupt code.

11. DEBUG FACILITIES

Debug facilities vary greatly between implementations. The original 600 family parts had only one instruction address breakpoint. 700 family parts have added a single data address breakpoint. 400 and 800 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.

12. 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.

---

[1] *PowerPC Programming Environments Manual* at http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2/$file/6xx_pem.pdf

[2] *Book E : Enhanced PowerPC Architecture* specification at http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600682CC7/$file/booke_rm.pdf

[3] *PowerPC Embedded Application Binary Interface, Version 1.*0, IBM and Motorola, January 10, 1995. Available from ESOFTA at http://www.esofta.com/pdfs/ppceabi.pdf

Printed in the United States of America January 16, 2004.

IBM Microelectronics Division
2070 Route 52, Bldg 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at
http://www.ibm.com

The IBM Microelectronics Division home page can be found at
http://www.ibm.com/chips