Chapter 1

Introduction

High-performance computer systems depend on good hardware design coupled with powerful compilers and operating systems. Although announced in 1991, the PowerPC architecture represents the end product of nearly 20 years of evolution starting with work on the 801 system at IBM. From the beginning, advanced hardware and software techniques were intermingled to develop first RISC and then superscalar computer systems. This guide describes how a compiler may select and schedule code that performs to the potential of the architecture.

1.1 RISC Technologies

The time required to execute a program is the product of the path length (the number of instructions), the number of cycles per instruction, and the cycle time. These three variables interact with one another. For example, reducing the cycle time reduces the window of time in which useful work can be performed, so the execution of a complex instruction may be unable to finish. Then, the function of the complex instruction must be separated into multiple simpler instructions, increasing the path length. Identifying the optimal combination of these variables in the form of an instruction set architecture, therefore, represents a challenging problem whose solution depends on the hardware technology and the software requirements.

Historically, CISC architectures evolved in response to the limited availability of memory because complex instructions result in smaller programs. As technology improved, memory cost dropped and access times decreased, so the decode and execution of the instructions became the limiting steps in instruction processing. Work at IBM, Berkeley, and Stanford demonstrated that performance improved if the instruction set was simple and instructions required a small number of cycles to execute, preferably one cycle. The reduction in cycle time and number of cycles

needed to process an instruction were a good trade-off against the increased path length. Development along these RISC lines continued at IBM and elsewhere. The physical design of the computer was simplified in exchange for increased hardware management by compilers and operating systems.

The work at IBM led to the development of the POWER™ architecture, which implemented parallel instruction (superscalar) processing, introduced some compound instructions to reduce instruction path lengths in critical areas, incorporated floating-point as a first-class data type, and simplified the architecture as a compiler target. Multiple pipelines permitted the simultaneous execution of different instructions, effectively reducing the number of cycles required to execute each instruction. The POWER architecture refined the original RISC approach by improving the mapping of the hardware architecture to the needs of programming languages. The functionality of key instructions was increased by combining multiple operations in the same instruction: the load and store with update instructions, which perform the access and load the effective address into the base register; the floating-point multiply-add instructions; the branch-on-count instructions, which decrement the Count Register and test the contents for zero; or the rotate-mask instructions. This increased functionality significantly reduced the path length for critical areas of code, such as loops, at the expense of moderately longer pipeline stages.

The POWER instruction set architecture and the hardware implementation were developed together so that they share a common partitioning based on function, minimizing the interaction between different functions. By arranging the instruction set in this way, the compiler could better arrange the code so that there were fewer inter-instruction dependencies impeding superscalar dispatch. The role of the compiler became more important because it generated code that could extract the performance potential of this superscalar hardware.

IBM, Motorola, and Apple jointly defined the PowerPC architecture as an evolution of the POWER architecture. The modifications to the POWER architecture include:

- Clearer distinctions between the architecture and implementations.
- Simplifications and specifications to improve high-speed superscalar and single-chip performance.
- 32-bit and 64-bit architectures.
- Memory-consistency model for symmetric multiprocessing.

1.2 Compilers and Optimization

The quality of code generated by a compiler is measured in terms of its size and execution speed. The compiler must balance these factors for the particular programming environment. The quality is most profoundly affected by the choice of algorithm and data structures, choices which are the province of the individual programmer. Given the algorithm and data structures, quality depends upon a collusion between the compiler, the processor architecture, and the specific implementation to best exploit the resources of the computer system. Modern processors rely upon statistical properties of the programs and upon the ability of the compiler to transform and schedule the specification of the algorithm in a semantically equivalent way so as to improve the performance of individual programs. Today, most programming is done in a high-level language. The compilers for these languages are free to generate the best possible machine code within the constraint that the semantics of the language are preserved. This book concentrates on compilers for procedure-oriented languages, such as C or Fortran.

Optimizations are traditionally classified as machine-independent or machine-dependent. Compilers usually perform machine-independent optimizations by transforming an intermediate language version of the program into an equivalent optimized program, also expressed in the intermediate language. The choice of optimizations normally considered machine-independent and their order of application, however, may actually be machine-dependent. Most classical compiler issues, including the front-end syntactic and semantic checks, intermediate language, and most machine-independent optimizations are not covered here; they are described elsewhere in the literature. This book focuses principally on implementation-dependent optimizations specific to the PowerPC architecture.

Machine-dependent optimizations require detailed knowledge of the processor architecture, the Application Binary Interface (ABI) and the processor implementation. Detailed issues of code choice depend mostly on the architecture. Typical compilers examine the intermediate representation of the program and select semantically equivalent machine instructions. The ABI is a convention that allows programs to function in a particular programming environment, but restricts the type of code that a compiler can emit in many contexts. Two PowerPC compilers that target different operating environments may generate quite different optimized code for the same program. Machine-dependent optimizations, such as program layout, scheduling, and alignment considerations, depend on the implementation of the architec-

ture. In the case of the PowerPC architecture, there are a number of implementations, each with different constraints on these optimizations.

1.3 Assumptions

The assumptions made in this book include:

- *Familiarity with the PowerPC Architecture*—We assume that you know the PowerPC architecture as described in *The PowerPC Architecture: A Specification for a New Family of RISC Processors* (hereafter known as *The PowerPC Architecture*). We make frequent references to sections in this book.
- *Common Model*—Unless otherwise stated, we assume that you are generating code for the PowerPC Common Model implementation, which is described in Section 4.3.6 on page 117. The Common Model is a fictional PowerPC implementation whose scheduled code should perform well, though perhaps not optimally, on all PowerPC implementations. Optimizations for particular processors are mentioned where appropriate. We consider only uniprocessor systems. Multiprocessing systems lie beyond the scope of this work.
- *Compiler Environment*—We assume that you have already developed a compiler front-end with an intermediate language connection to an optimizing and code-emitting back-end, or that you are directly optimizing application programs in an assembler. This book discusses only the optimizing and code-emitting back-end that creates PowerPC object files.

Chapter 2

Overview of the PowerPC Architecture

Books I through III of *The PowerPC Architecture* describe the instruction set, virtual environment, and operating environment, respectively. The user manual for each processor specifies the implementation features of that processor. In this book, the term *PowerPC architecture* refers to the contents of Books I through III. The compiler writer is concerned principally with the contents of Book I: PowerPC User Instruction Set Architecture.

2.1 Application Environment

The application environment consists of resources accessible from the *problem state*, which is the user mode (the PR bit in the Machine State Register is set). The PowerPC architecture is a load-store architecture that defines specifications for both 32-bit and 64-bit implementations. The instruction set is partitioned into three *functional classes*: branch, fixed-point and floating-point. The registers are also partitioned into groups corresponding to these classes; that is, there are condition code and branch target registers for branches, Floating-Point Registers for floating-point operations, and General-Purpose Registers for fixed-point operations. This partition benefits superscalar implementations by reducing the interlocking necessary for dependency checking. The explicit indication of all operands in the instructions, combined with the partitioning of the PowerPC architecture into functional classes, exposes dependences to the compiler. Although instructions must be word (32-bit) aligned, data can be misaligned within certain implementation-dependent constraints. The floating-point facilities support compliance to the *IEEE 754 Standard for Binary Floating-Point Arithmetic* (IEEE 754).

2.1.1 32-Bit and 64-Bit Implementations and Modes

The PowerPC architecture includes specifications for both 32- and 64-bit implementations. In 32-bit implementations, all application registers have 32 bits, except for the 64-bit Floating-Point Registers, and effective addresses have 32 bits. In 64-bit imple-

mentations, all application registers are 64-bits long—except for the 32-bit Condition Register, FPSCR, and XER—and effective addresses have 64 bits. Figure 2-1 shows the application register sizes in 32-bit and 64-bit implementations.

Figure 2-1. Application Register Sizes

Registers	*32-Bit Implementation Size (Bits)*	*64-Bit Implementation Size (Bits)*
Condition Register	32	32
Link Register and Count Register	32	64
General-Purpose Registers	32	64
fixed-point Exception Register	32	32
Floating-Point Registers	64	64
Floating-Point Status and Control Register	32	32

Both 32-bit and 64-bit implementations support most of the instructions defined by the PowerPC architecture. The 64-bit implementations support all the application instructions supported 32-bit implementations as well as the following application instructions: load doubleword, store doubleword, load word algebraic, multiply doubleword, divide doubleword, rotate doubleword, shift doubleword, count leading zeros doubleword, sign extend word, and convert doubleword integer to a floating-point value.

The 64-bit implementations have two modes of operation determined by the 64-bit mode (SF) bit in the Machine State Register: 64-bit mode (SF set to 1) and 32-bit mode (SF cleared to 0), for compatibility with 32-bit implementations. Application code for 32-bit implementations executes without modification on 64-bit implementations running in 32-bit mode, yielding identical results. All 64-bit implementation instructions are available in both modes. Identical instructions, however, may produce different results in 32-bit and 64-bit modes:

- *Addressing*—Although effective addresses in 64-bit implementations have 64 bits, in 32-bit mode, the high-order 32 bits are ignored during data access and set to zero during instruction fetching. This modification of the high-order bits of the address might produce an unexpected jump following the transition from 64-bit mode to 32-bit mode.
- *Status Bits*—The register result of arithmetic and logical instructions is independent of mode, but setting of status bits depends on the mode. In particular, recording, carry-bit–setting, or overflow-bit–setting instruction forms write the status

bits relative to the mode. Changing the mode in the middle of a code sequence that depends on one of these status bits can lead to unexpected results.

- *Count Register*—The entire 64-bit value in the Count Register of a 64-bit implementation is decremented, even though conditional branches in 32-bit mode only test the low-order 32 bits for zero.

2.1.2 Register Resources

The PowerPC architecture identifies each register with a functional class, and most instructions within a class use only the registers identified with that class. Only a small number of instructions transfer data between functional classes. This separation of processor functionality reduces the hardware interlocking needed for parallel execution and exposes register dependences to the compiler.

2.1.2.1 Branch

The Branch-Processing Unit includes the Condition Register, Link Register (LR) and Count Register (CTR):

- *Condition Register*—Conditional comparisons are performed by first setting a condition code in the Condition Register with a compare instruction or with a recording instruction. The condition code is then available as a value or can be tested by a branch instruction to control program flow. The 32-bit Condition Register consists of eight independent 4-bit fields grouped together for convenient save or restore during a context switch. Each field may hold status information from a comparison, arithmetic, or logical operation. The compiler can schedule Condition Register fields to avoid data hazards in the same way that it schedules General-Purpose Registers. Writes to the Condition Register occur only for instructions that explicitly request them; most operations have recording and non-recording instruction forms.
- *Link Register*—The Link Register may be used to hold the effective address of a branch target. Branch instructions with the link bit (LK) set to one copy the next instruction address into the Link Register. A Move To Special-Purpose Register instruction can copy the contents of a General-Purpose Register into the Link Register.
- *Count Register*—The Count Register may be used to hold either a loop counter or the effective address of a branch target. Some conditional-branch instruction forms decrement the Count Register and test it for a zero value. A Move To Special-Purpose Register instruction can copy the contents of a General-Purpose Register into the Count Register.

2.1.2.2 Fixed-Point

The Fixed-Point Unit includes the General-Purpose Register file and the Fixed-Point Exception Register (XER):

- *General-Purpose Registers*—Fixed-point instructions operate on the full width of the 32 General-Purpose Registers. In 64-bit implementations, the instructions are mode-independent, except that in 32-bit mode, the processor uses only the low-order 32 bits for determination of a memory address and the carry, overflow, and record status bits.
- *XER*—The XER contains the carry and overflow bits and the byte count for the move-assist instructions. Most arithmetic operations have carry-bit–setting and overflow-bit–setting instruction forms.

2.1.2.3 Floating-Point

The Floating-Point Unit includes the Floating-Point Register file and the Floating-Point Status and Control Register (FPSCR):

- *Floating-Point Registers*—The Floating-Point Register file contains thirty-two 64-bit registers. The internal format of floating-point data is the IEEE 754 double-precision format. Single-precision results are maintained internally in the double-precision format.
- *FPSCR*—The processor updates the 32-bit FPSCR after every floating-point operation to record information about the result and any associated exceptions. The status information required by IEEE 754 is included, plus some additional information to expedite exception handling.

2.1.3 **Memory Models**

Memory is considered to be a linear array of bytes indexed from 0 to 2^{32} - 1 in 32-bit implementations, and from 0 to 2^{64} - 1 in 64-bit implementations. Each byte is identified by its index, called an *address,* and each byte contains a value. For the uniprocessor systems considered in this book, one storage access occurs at a time and all accesses appear to occur in program order. The main considerations for the compiler writer are the addressing modes, alignment, and endian orientation. Although these considerations alone suffice for the correct execution of a program, code modifications that better utilize the caches and translation-lookaside buffers may improve performance (see Section 4.4 on page 133).

2.1.3.1 Memory Addressing

The PowerPC architecture implements three addressing modes for instructions and three for data. The address of either an instruction or a multiple-byte data value is its lowest-numbered byte. This address points to the most-significant end in big-endian mode, and the least-significant end in little-endian mode.

Instructions

Branches are the only instructions that specify the address of the next instruction; all others rely on incrementing a program counter. A branch instruction indicates the effective address of the target in one of the following ways:

- *Branch Not Taken*—The byte address of the next instruction is the byte address of the current instruction plus 4.
- *Absolute*—Branch instructions to absolute addresses (indicated by setting the AA bit in the instruction encoding) transfer control to the word address given in an immediate field of the branch instruction. The sign-extended value in the 24-bit or 14-bit immediate field is scaled by 4 to become the byte address of the next instruction. The high-order 32 bits of the address are cleared in the 32-bit mode of a 64-bit implementation. An unconditional branch to an absolute address, which has a 24-bit immediate field, transfers control to a byte address in the range 0x0 to 0x01FF_FFFC or 0xFE00_8000 to 0xFFFF_FFFC. A conditional branch to an absolute address, which has a 14-bit immediate field, transfers control to a byte address in the range 0x0 to 0x7FFC or 0xFFFF_8000 to 0xFFFF_FFFC.
- *Relative*—Branch instructions to relative addresses (indicated by clearing the AA bit in the instruction encoding) transfer control to the word address given by the sum of the immediate field of the branch instruction and the word address of the branch instruction itself. The sign-extended value in the 24-bit or 14-bit immediate field is scaled by 4 and then added to the current byte instruction address to become the byte address of the next instruction. The high-order 32 bits of the address are cleared in the 32-bit mode of a 64-bit implementation.
- *Link Register or Count Register*—The Branch Conditional to Link Register and Branch Conditional to Count Register instructions transfer control to the effective byte address of the branch target specified in the Link Register or Count Register, respectively. The low-order two bits are ignored because all PowerPC instructions must be word aligned. In a 64-bit implementation, the high-order 32 bits of the target address are cleared in 32-bit mode. The Link Register and Count Registers are written or read using the *mtspr* and *mfspr* instructions, respectively.

Data

All PowerPC load and store instructions specify an address register, which is indicated in the RA field of the instruction. If RA is 0, the value zero is used instead of the contents of R0. The effective byte address in memory for a data value is calculated relative to the base register in one of three ways:

- *Register + Displacement*—The displacement forms of the load and store instructions add a displacement specified by the sign-extended 16-bit immediate field of the instruction to the contents of RA (or 0 for R0).

- *Register* + *Register*—The indexed forms of the load and store instructions add the contents of the index register, which is a General-Purpose Register, to the contents of RA (or 0 for R0).
- *Register*—The Load String Immediate and Store String Immediate instructions directly use the contents of RA (or 0 for R0).

The update forms reload the register with the computed address, unless RA is 0 or RA is the target register of the load.

Arithmetic for address computation is unsigned and ignores any carry out of bit 0. In 32-bit mode of a 64-bit implementation, the processor ignores the high-order 32-bits, but includes them when the address is loaded into a General-Purpose Register, such as during a load or store with update.

2.1.3.2 Endian Orientation

The address of a multi-byte value in memory can refer to the most-significant end (big-endian) or the least-significant end (little-endian). By default, the PowerPC architecture assumes that multi-byte values have a big-endian orientation in memory, but values stored in little-endian orientation may be accessed by setting the Little-Endian (LE) bit in the Machine State Register. In PowerPC Little-Endian mode, the memory image is not true little-endian, but rather the ordering obtained by the address modification scheme specified in Appendix D of Book I of *The PowerPC Architecture*. In Little-Endian mode, load multiple, store multiple, load string, and store string operations generate an Alignment interrupt. Other little-endian misaligned load and store operations may also generate an Alignment interrupt, depending on the implementation. In most cases, the load and store with byte reversal instructions offer the simplest way to convert data from one endian orientation to the other in either endian mode.

2.1.3.3 Alignment

The alignment of instruction and storage operands affects the result and performance of instruction fetching and storage accesses, respectively.

Instructions

PowerPC instructions must be aligned on word (32-bit) boundaries. There is no way to generate an instruction address that is not divisible by 4.

Data

Although the best performance results from the use of aligned accesses, the PowerPC architecture is unusual among RISC architectures in that it permits misaligned data accesses. Different PowerPC implementations respond differently to misaligned accesses. The processor hardware may handle the access or may generate an Alignment interrupt. The Alignment interrupt handler may handle the access or indicate that a program error

has occurred. Load-and-reserve and store-conditional instructions to misaligned effective addresses are considered program errors. Alignment interrupt handling may require on the order of hundreds of cycles, so every effort should be made to avoid misaligned memory values.

In Big-Endian mode, the PowerPC architecture requires implementations to handle automatically misaligned integer halfword and word accesses, word-aligned integer doubleword accesses, and word-aligned floating-point accesses. Other accesses may or may not generate an Alignment interrupt depending on the implementation.

In Little-Endian mode, the PowerPC architecture does not require implementation hardware to handle any misaligned accesses automatically, so any misaligned access may generate an Alignment interrupt. Load multiple, store multiple, load string, and store string instructions always generate an Alignment interrupt in Little-Endian mode.

A misaligned access, a load multiple access, store multiple access, a load string access, or a store string access that crosses a page, Block Address Translation (BAT) block, or segment boundary in an ordinary segment may be restarted by the implementation or the operating system. Restarting the operation may load or store some bytes at the target location for a second time. To ensure that the access is not restarted, the data should be placed in either a BAT or a direct-store segment, both of which do not permit a restarted access.

2.1.4 Floating-Point

The PowerPC floating-point formats, operations, interrupts, and special-value handling conform to IEEE 754. The remainder operation and some conversion operations required by IEEE 754 must be implemented in software (in the run-time library).

A Floating-Point Register may contain four different data types: single-precision floating-point, double-precision floating-point, 32-bit integer, and 64-bit integer. The integer data types can be stored to memory or converted to a floating-point value for computation. The *frsp* instruction rounds double-precision values to single-precision. The precision of the result of an operation is encoded in the instruction. Single-precision operations should act only on single-precision operands.

The floating-point operating environment for applications is determined by bit settings in the Floating-Point Status and Control Register (FPSCR) and the Machine State Register (MSR). Figure 2-2 shows the bit fields and their functions. Floating-point interrupts may be disabled by clearing FE0 and FE1. If either FE0 or FE1 is set, individual IEEE 754 exception types are enabled with the bits in the FPSCR indicated in Figure 2-2.

The non-IEEE mode implemented by some implementations may be used to obtain deterministic performance (avoiding traps and interrupts) in certain applications. See Section 3.3.7.1 on page 79 for further details.

Figure 2-2. Floating-Point Application Control Fields

Register	*Field **	*Name*	*Function*
FPSCR	24	VE	*Floating-Point Invalid Operation Exception Enable* 0 Invalid operation exception handled with the IEEE 754 default response. 1 Invalid operation exception causes a Program interrupt.
	25	OE	*Floating-Point Overflow Exception Enable* 0 Overflow exception handled with the IEEE 754 default response. 1 Overflow exception causes a Program interrupt.
	26	UE	*Floating-Point Underflow Exception Enable* 0 Underflow exception handled with the IEEE 754 default response. 1 Underflow exception causes a Program interrupt.
	27	ZE	*Floating-Point Zero-Divide Exception Enable* 0 Zero divide exception handled with the IEEE 754 default response. 1 Zero divide exception causes a Program interrupt.
	28	XE	*Floating-Point Inexact Exception Enable* 0 Inexact exception handled with the IEEE 754 default response. 1 Inexact exception causes a Program interrupt.
	29	NI	*Floating-Point Non-IEEE Mode* 0 The processor executes in an IEEE 754 compatible manner. 1 The processor produces some results that do not conform with IEEE 754.
** 64-bit and 32-bit refer to the type of implementation.*			

Figure 2-2. Floating-Point Application Control Fields (continued)

Register	*Field **	*Name*	*Function*
	30:31	RN	*Floating-Point Rounding Control* 00 Round to Nearest 01 Round toward 0 10 Round toward +∞ 11 Round toward -∞
MSR	64-bit: 52 32-bit: 20 64-bit: 55 32-bit: 23	FE0 FE1	*Floating-Point Exception Modes 0 and 1* 00 Ignore exceptions mode. Floating-point exceptions do not cause interrupts. 01 Imprecise nonrecoverable mode. The processor interrupts the program at some point beyond the instruction that caused the enabled exception, and the interrupt handler may not be able to identify this instruction. 10 Imprecise recoverable mode. The processor interrupts the program at some point beyond the instruction that caused the enabled exception, but the interrupt handler can identify this instruction. 11 Precise mode. The program interrupt is generated precisely at the floating-point instruction that caused the enabled exception.
** 64-bit and 32-bit refer to the type of implementation.*			

2.2 Instruction Set

All instructions are 32 bits in length. Most computational instructions specify two source register operands and a destination register operand. Only load and store instructions access memory. Furthermore, most instructions access only the registers of the same functional class. Branch instructions permit control transfers either unconditionally, or conditionally based on the test of a bit in the Condition Register. The branch targets can be immediate values given in the branches or the contents of the Link or Count Register. The fixed-point instructions include the storage access, arithmetic, compare, logical, rotate and shift, and move to/from system register instructions. The floating-point instructions include storage access, move, arithmetic, rounding and conversion, compare, and FPSCR instructions.

2.2.1 Optional Instructions

The PowerPC architecture includes a set of optional instructions:

- *General-Purpose Group—fsqrt* and *fsqrts*.
- *Graphics Group—stfiwx*, *fres*, *frsqrte*, and *fsel*.

If an implementation supports any instruction in a group, it must support all of the instructions in the group. Check the documentation for a specific implementation to determine which, if any, of the groups are supported.

2.2.2 Preferred Instruction Forms

Some instructions have a preferred form, which may execute significantly faster than other forms. Instructions having preferred forms include:

- *Load and Store Multiple*—Load multiple and store multiple instructions load or store the sequence of successive registers from the first target or source register through R31. In the preferred form, the combination of the effective address and the first target or source register should align the low-order byte of R31 to a 128-bit (quadword) boundary.
- *Load and Store String*—Load string and store string instructions load or store the sequence of bytes from the first target or source register through successive registers until all the bytes are transferred. In the preferred form, the target or source register is R5 and the last accessed register is R12 or lower.
- *No-Op*—The preferred form of the no-op is *ori* 0,0,0.

2.2.3 Communication Between Functional Classes

A special group of instructions manage the communication between the resources of different functional classes. No branch instructions can use resources of the non-branch classes. The communication always occurs through an fixed-point or floating-point instruction. The execution of these instructions may cause substantial implementation-dependent delays because both execution units must be available simultaneously as both are involved in the execution of the instruction.

2.2.3.1 Fixed-Point and Branch Resources

The fixed-point instructions manage the following transfers between fixed-point and branch registers:

- *General-Purpose Register to Condition Register*—Move To Condition Register Fields (*mtcrf*)
- *Condition Register to General-Purpose Register*—Move From Condition Register (*mfcr*)
- *General-Purpose Register to Link Register*—Move To Link Register (*mtlr*)
- *Link Register to General-Purpose Register*—Move From Link Register (*mflr*)
- *General-Purpose Register to Count Register*—Move To Count Register (*mtctr*)
- *Count Register to General-Purpose Register*—Move From Count Register (*mfctr*)

- *XER to Condition Register*—Move to Condition Register Field from XER (*mcrxr*)
- *Fixed-Point Record Instruction to CR0*—An fixed-point arithmetic or logical instruction with the record bit set loads CR0 with condition codes representing the comparison of the result to 0 and the Summary Overflow bit.
- *Fixed-point Comparison Instruction to CRx*—An fixed-point comparison instruction loads the CR field specified in the instruction with condition codes representing the result of the comparison and the Summary Overflow bit.

2.2.3.2 Fixed-Point and Floating-Point Resources

No direct connection exists between General-Purpose Registers and Floating-Point Registers. A transfer between these registers must be done by storing the register value in memory from one functional class and then loading the value into a register of the other class.

2.2.3.3 Floating-Point and Branch Resources

The floating-point instructions manage the following transfers between Floating-Point Registers and Branch Unit registers:

- *FPSCR to CR*—Move to Condition Register field from FPSCR (*mcrfs*)
- *Floating-Point Record Instruction to CR1*—A floating-point arithmetic instruction with the record bit enabled loads CR1 with condition codes indicating whether an exception occurred.
- *Floating-Point Comparison to CRx*—A floating-point comparison instruction loads the CR field specified in the instruction with condition codes representing the result of the comparison.