This chapter introduces the numeric implementation in PowerPC assembly language. It describes the basics of the floating-point architecture, showing what floating-point data formats and registers are available, what numeric operations are available in assembly language, and what load and store instructions you must use before you can perform assembly-language numeric operations. An example application using assembly-language numeric operations is shown at the end of this chapter.

Read this chapter to learn how to use the numeric assembly-language instructions described in Chapters 12 through 14.

PowerPC Floating-Point Architecture

This section describes those pieces of the PowerPC architecture used in floating-point operations, which include

- floating-point data formats
- floating-point registers
- floating-point special-purpose registers
- the Machine State Register

Floating-Point Data Formats

The PowerPC architecture supports only the single and double floating-point data formats. These formats can represent normalized numbers, denormalized numbers, zeros, NaNs, and Infinities, and are interpreted exactly as described in Chapter 2, “Floating-Point Data Formats.” The double-double data format is implemented in software and therefore is not a valid format in PowerPC hardware.

The PowerPC hardware is double-based. This means that when you load a single-format number into a register, it is automatically converted to double format. In addition, all arithmetic operations are performed on double-format numbers unless they are specifically forced to be performed on single-format numbers.

Floating-Point Registers

The PowerPC architecture contains thirty-two 64-bit floating-point registers labeled F0 through F31 (or FP0 through FP31). Because the registers are 64 bits long, they store values using the double data format.

Floating-Point Special-Purpose Registers

The two special-purpose registers that affect floating-point operations are the Floating-Point Status and Control Register and the Condition Register.

The **Floating-Point Status and Control Register (FPSCR)** is a 32-bit register that stores the current state of the floating-point environment. It specifies the current rounding direction and notes whether any floating-point exceptions are enabled and whether any floating-point exceptions have occurred.

The **Condition Register** is a 32-bit register that stores the current state of the entire PowerPC processor. It is grouped into eight 4-bit fields labeled CR0 through CR7. Field CR1 reflects the results of floating-point operations. You may also specify one of the Condition Register fields as a place to store the result of a floating-point comparison operation or the result of a floating-point environment manipulation operation.

The FPSCR and the Condition Register are discussed more fully in Chapter 12, "Assembly-Language Environmental Controls."

The Machine State Register

The **Machine State Register** is a 32-bit supervisor-level register that reflects the current state of the entire PowerPC processor. It differs from the Condition Register in that it is accessible only by supervisor-level software and in that it stores the processor state in a different way. The Machine State Register contains 3 bits that control floating-point computations:

- Bit 18 specifies whether the floating-point instructions are available. If bit 18 is 0, the processor cannot execute floating-point instructions.
- Bits 20 and 23 specify whether floating-point exceptions are enabled. If both of these bits are 0, floating-point instructions will not raise any floating-point exceptions. If either of these bits is set, instructions can raise floating-point exceptions.

Floating-Point Instructions

Most floating-point operations are performed by the PowerPC floating-point processor. Floating-point arithmetic, conversion, comparison, and other operations are supported through assembler instructions. The only basic arithmetic operations supported are add, subtract, multiply, divide, and round-to-integer. In addition to instructions that perform the basic numeric operations, PowerPC assembly language provides instructions that can perform both a multiply and an add or subtract with at most a single roundoff error (called *multiply-add instructions*) and instructions that manipulate the sign bit of a number. All PowerPC floating-point assembler instructions conform to the IEEE standard.

All floating-point instructions (other than load instructions) operate on data located in the floating-point registers. The data must be loaded into a floating-point register before any operation can be performed.

Even though the floating-point registers are double format, the data can be in either single or double format. The instruction mnemonic specifies whether the data in the floating-point register is interpreted as single or double format. For example, `fadd` means add two double-format numbers, and `fadds` means add two single-format numbers.

Load and Store Instructions

Before you perform any floating-point computation, you must load a value into a floating-point register. To do this, use one of the load instructions. Load instructions load either single or double floating-point numbers from memory into floating-point registers. Store instructions take the contents of a floating-point register and store them in memory.

Load and store instructions take one of two forms depending on which address mode is used. The first form is

instr *FPR*, *D(GPR)*

instr	Specifies which type of load or store is to be performed.
FPR	A floating-point register, which is either the source or the destination for the operation, depending on whether it is a load or a store.
D	A 16-bit signed integer value.
GPR	A general-purpose register or the value 0.

The *D*(*GPR*) part of the instruction determines the memory address involved. If *GPR* is not 0, it is interpreted as a general-purpose register and the contents of register *GPR* are added to the value *D* to produce the memory address. If *GPR* is 0, it is interpreted as the value 0 rather than as register GPR0, so 0 is added to *D* to produce the memory address.

Load instructions of this form are interpreted as *FPR* ← (*D* + (*GPR*)), which means that the instruction loads into *FPR* the contents of the memory address obtained by adding *D* to the contents of *GPR* (unless *GPR* is 0).

Store instructions of this form are interpreted as *D* + (*GPR*) ← (*FPR*), which means that the instruction stores the contents of *FPR* at the memory address obtained by adding *D* to the contents of *GPR* (unless *GPR* is 0).

The second form for load and store operations uses a different address mode:

instr *FPR*, *GPR1*, *GPR2*

instr	Specifies which type of load or store is to be performed.
FPR	A floating-point register, which is either the source or the destination for the operation, depending on whether it is a load or a store.
GPR1	A general purpose register or the value 0.
GPR2	A general-purpose register.

GPR1 and *GPR2* determine the memory address involved. If *GPR1* is not 0, it is interpreted as a general-purpose register, and the contents of register *GPR1* are added to the contents of register *GPR2* to produce the memory address. If *GPR1* is 0, it is interpreted as the value 0 rather than as register GPR0, so 0 is added to the contents of register *GPR2* to produce the memory address.

Load instructions of this form are interpreted as *FPR* ← ((*GPR1*) + (*GPR2*)) unless *GPR1* is 0.

Store instructions of this form are interpreted as (*GPR1*) + (*GPR2*) ← (*FPR*) unless *GPR1* is 0.

Table 11-1 lists and describes the PowerPC load and store instructions. There are two load and two store instructions for each address mode. One version simply performs the load or store, and the other version puts the effective memory address into the general-purpose register specified in the instruction (shown as R*n* in the table).

Each of the load and store instructions has a single and a double form, making a total of eight load and eight store instructions. If the single form of a load instruction is used, the number is converted to double format before the load is performed. If the single form of a store instruction is used, the number is converted to single format before it is stored. See Chapter 13, "Assembly-Language Numeric Conversions," for more information about conversions performed during load and store operations.

None of the load and store instructions raise floating-point exceptions or make special cases of zeros, NaNs, or Infinities.

Table 11-1 Load and store floating-point instructions

Address mode	Instruction syntax	Operation
d(R*n*)	`lfd` *DST*, *n*(*GPR*)	Load double format
	`stfd` *SRC*, *n*(*GPR*)	Store double format
	`lfs` *DST*, *n*(*GPR*)	Load single format
	`stfs` *SRC*, *n*(*GPR*)	Store single format
	`lfdu` *DST*, *n*(*GPR*)	Load double format and update
	`stfdu` *SRC*, *n*(*GPR*)	Store double format and update
	`lfsu` *DST*, *n*(*GPR*)	Load single format and update
	`stfsu` *SRC*, *n*(*GPR*)	Store single format and update

Table 11-1 Load and store floating-point instructions (continued)

Address mode	Instruction syntax	Operation
R*n*, R*m*	`lfdx` *DST*, *GPR1*, *GPR2*	Load double format indexed
	`stfdx` *SRC*, *GPR1*, *GPR2*	Store double format indexed
	`lfsx` *DST*, *GPR1*,*GPR2*	Load single format indexed
	`stfsx` *SRC*, *GPR1*, *GPR2*	Store single format indexed
	`lfdux` *DST*, *GPR1*, *GPR2*	Load double format and update indexed
	`stfdux` *SRC*, *GPR1*, *GPR2*	Store double format and update indexed
	`lfsux` *DST*, *GPR1*, *GPR2*	Load single format and update indexed
	`stfsux` *SRC*, *GPR1*, *GPR2*	Store single format and update indexed

Numerics Example Using PowerPC Assembly Language

Listing 11-1 is a code example that shows when the PowerPC assembly-language numeric features might be useful. The instructions used in this example are described in the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. This example evaluates the polynomial

$x^3 + 2x^2 - 5$

It illustrates the evaluation of a polynomial

$c_0 x^n + c_1 x^{n-1} + \ldots + c_n$

using Horner's recurrence

$r \leftarrow c_0$

$r \leftarrow (r \times x) + c_j \quad \text{for } j = 1 \text{ to } n$

On entry, general-purpose register GPR0 contains the degree n (<256) of the polynomial, and floating-point register F1 points to a function argument x. The coefficient table consists of $n + 1$ double-format coefficients, starting with c_0. In this particular polynomial, $n = 3$, $c_0 = 1$, $c_1 = 2$, $c_2 = 0$, and $c_3 = -5$.

Listing 11-1 Polynomial evaluation

```
r0:   equ   0                # general-purpose register 0
r5:   equ   5                # general-purpose register 5
f0:   equ   0                # floating-point register 0
f1:   equ   1                # floating-point register 1
f2:   equ   2                # floating-point register 2
CTR:  equ   9                # Count Register for loops

extern   polyeval{DS}        # export the routine descriptor
extern   .polyeval           # export the entry point
# put the code in a program control section
csect    polyeval{PR}

#high-level languages prepend a period to function names
.polyeval:
   lwz      r0,0(r5)         # r0 = degree
   lfd      f0,4(r5)         # f0 = leading coefficient, c0
   addic    r5,r5,4          # r5 = address of leading coeff. &c0
   mtspr    CTR,r0           # CTR = r0
loop:
   lfdu     f2,8(r5)         # f2 = next coefficient
                             # update r5 = r5 + 8
   fmadd    f0,f0,f1,f2      # f0 = f0 * f1 + f2; ...
                             # res = res * x + c[j]
   bdnz     loop             # CTR = CTR - 1, branch if CTR ≠ 0
   fmr      f1,f0            # f1 = f0
   blr                       # return through the Link Register
   nop


#
# Set up the table of contents.  It must include at least the
# exported routines. It may also contain global data or pointers
# to data.
#
polyeval_TOC:  tc    polyeval{tc}, polyeval{PR}
```

```
#
# Build a transition vector for all exported routines so they can
# be accessed through an inter-TOC call.
#
 csect   polyeval{DS}      # it's in a separate control section
 dc.l    .polyeval         # contains the entry point
 dc.l    0                 # loader will fill in correct TOC
                           # pointer
 dc.l    0                 # save space for environment pointer
```