This chapter describes how to use assembly-language instructions to control the floating-point environment (rounding direction and exception flags) described in Chapter 4, "Environmental Controls." The current state of the floating-point environment is stored in the Floating-Point Status and Control Register and summarized in the Condition Register. This chapter describes exactly how these two registers store the environment. Then it describes the PowerPC assembler instructions you can use to test or change the environment.

Read this chapter to learn how to access and manipulate the floating-point environment in assembly language or to learn how the PowerPC architecture stores the floating-point environment.

## The Floating-Point Environment

The two special-purpose registers that reflect and control the floating-point environment are the Floating-Point Status and Control Register and the Condition Register.

### The Floating-Point Status and Control 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, whether any floating-point exceptions are enabled, and whether any floating-point exceptions have occurred. Many instructions that manipulate the FPSCR operate on 4-bit fields numbered 0 through 7. Figure 12-1 highlights some of the more useful fields in the FPSCR, and Table 12-1 shows their bit assignments. For more information on floating-point instructions, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*.

**Figure 12-1** Floating-Point Status and Control Register (FPSCR)

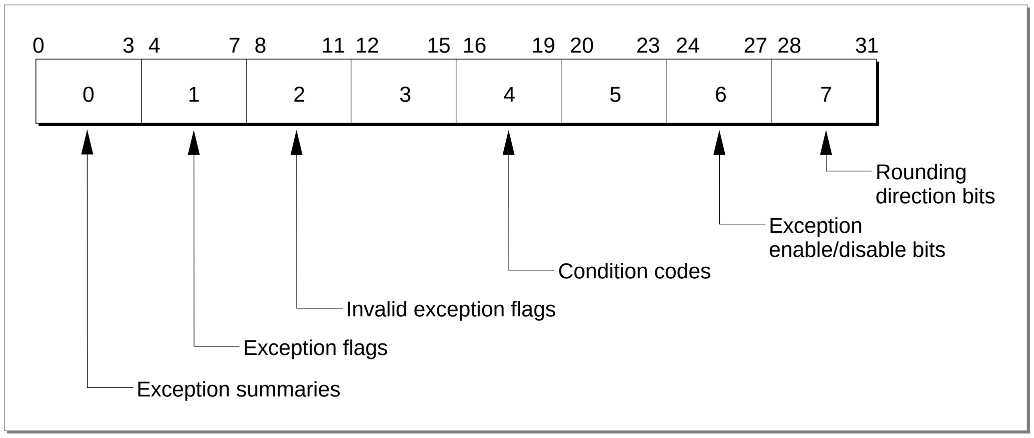


**Table 12-1** Bit assignments for FPSCR fields

| FPSCR field | Bit | Meaning if set |
|---|---|---|
| 0 | 0 | One or more of the floating-point exceptions occurred. |
| | 1 | One or more of the floating-point exceptions is enabled. |
| | 2 | One or more of the invalid exceptions occurred. |
| | 3 | An overflow exception occurred. |
| 1 | 4 | An underflow exception occurred. |
| | 5 | A divide-by-zero exception occurred. |
| | 6 | An inexact exception occurred. |
| | 7 | An invalid exception occurred because an operation other than load, store, move, select, or `mtfsf` was attempted on a signaling NaN. |
| 2 | 8 | An invalid exception occurred because $\infty - \infty$ was attempted. |
| | 9 | An invalid exception occurred because $\infty / \infty$ was attempted. |
| | 10 | An invalid exception occurred because $0 / 0$ was attempted. |
| | 11 | An invalid exception occurred because $0 \times \infty$ was attempted. |
| 3 | 12 | An invalid comparison operation was attempted. |
| | 13 | The fraction field of the result has been rounded. |
| | 14 | The fraction field of the result is inexact. |
| | 15 | Class descriptor. See “Inquiries: Class and Sign” on page 12-7. |
| 4 | 16 | Less than or less than 0. See “Inquiries: Class and Sign” on page 12-7. |
| | 17 | Greater than or greater than 0. See “Inquiries: Class and Sign” on page 12-7. |
| | 18 | Equal to or equal to 0. See “Inquiries: Class and Sign” on page 12-7. |
| | 19 | Unordered or NaN. See “Inquiries: Class and Sign” on page 12-7. |
| 5 | 20 | Reserved. |
| | 21 | An invalid exception occurred because of a software request. Not implemented in MPC601. |
| | 22 | An invalid square-root operation was attempted. Not implemented in MPC601. |
| | 23 | An invalid exception occurred because of an invalid convert-to-integer operation. |
| 6 | 24 | The invalid exceptions are enabled. |
| | 25 | The overflow exception is enabled. |
| | 26 | The underflow exception is enabled. |
| | 27 | The divide-by-zero exception is enabled. |

**Table 12-1** Bit assignments for FPSCR fields (continued)

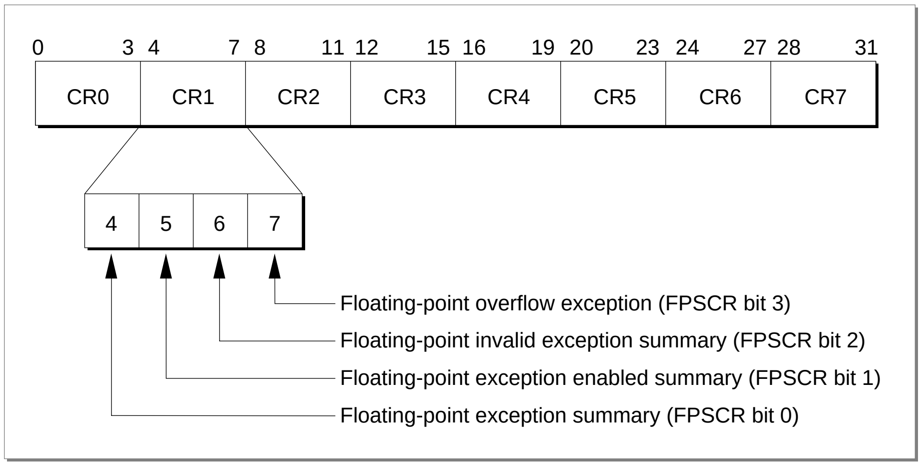
| FPSCR field | Bit | Meaning if set |
|---|---|---|
| 7 | 28 | The inexact exception is enabled. |
| | 29 | Reserved. |
| | 30 | Rounding direction. See "Setting the Rounding Direction" on page 12-9. |
| | 31 | Rounding direction. See "Setting the Rounding Direction" on page 12-9. |

**IMPORTANT**
Bit 20 or 23 of the Machine State Register must be set for the FPSCR exception enable bits to be valid. For more information, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. ▲

## The Condition Register

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 (see Figure 12-2). Field CR1 (bits 4 through 7) reflects the results of floating-point operations.

**Figure 12-2** Condition Register


0
3
4
7
8
11
12
15
16
19
20
23
24
27
28
31
CR0
CR1
CR2
CR3
CR4
CR5
CR6
CR7
4
5
6
7
Floating-point overflow exception (FPSCR bit 3)
Floating-point invalid exception summary (FPSCR bit 2)
Floating-point exception enabled summary (FPSCR bit 1)
Floating-point exception summary (FPSCR bit 0)

| Bit | Meaning |
|---|---|
| 4 | Set if bit 0 of the FPSCR is set. That is, this bit indicates whether any floating-point exception has occurred. |
| 5 | Set if bit 1 of the FPSCR is set. That is, this bit indicates whether any of the floating-point exceptions are enabled. |
| 6 | Set if bit 2 of the FPSCR is set. That is, this bit indicates whether an invalid exception has occurred for any reason. |
| 7 | Set if bit 3 of the FPSCR is set. That is, this bit indicates whether an overflow has occurred. |

If you append a dot (.) to a floating-point instruction, its status will be recorded in the Condition Register as well as in the FPSCR. If you do not append a dot, the Condition Register will not reflect the result of that instruction.

Use Condition Register fields in conditional branch instructions. Several instructions allow you to store certain FPSCR bits in fields CR2 through CR4. After using one of these instructions, you then use a conditional branch instruction of the form

*instr* *field*`,` *address*

where *field* is the Condition Register field 2 through 4 and *address* is the address to branch to if the condition is true. Table 12-2 shows some commonly used PowerPC branch instructions. Examples of how to use the conditional branch instructions appear later in this chapter. For a complete list of conditional branch instructions, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*.

**Table 12-2** Branch instructions using the Condition Register

| Instruction | Description |
|---|---|
| `bta` *bit*, *address* | Branch to *address* if condition is true (*bit* = 1) |
| `blt` *field*, *address* | Branch to *address* if less than (bit 0 of *field* = 1) |
| `ble` *field*, *address* | Branch to *address* if less than or equal (bit 0 of *field* = 1 or bit 2 = 1) |
| `beq` *field*, *address* | Branch to *address* if equal (bit 2 of *field* = 1) |
| `bge` *field*, *address* | Branch to *address* if greater than or equal (bit 1 of *field* = 1 or bit 2 = 1) |
| `bgt` *field*, *address* | Branch to *address* if greater than (bit 1 of *field* = 1) |
| `bnl` *field*, *address* | Branch to *address* if not less than (bit 0 of *field* = 0) |
| `bne` *field*, *address* | Branch to *address* if not equal (bit 2 of *field* = 0) |
| `bng` *field*, *address* | Branch to *address* if not greater than (bit 1 of *field* = 0) |
| `bun` *field*, *address* | Branch to *address* if unordered (bit 3 of *field* = 1) |
| `bnu` *field*, *address* | Branch to *address* if not unordered (bit 3 of *field* = 0) |

## Inquiries: Class and Sign

As stated in Chapter 2, “Floating-Point Data Formats,” the result of a floating-point operation is either a normalized number, a denormalized number, a zero, a NaN, or an Infinity. This section describes how the class and sign of a floating-point number can be determined in PowerPC assembly language.

### Floating-Point Result Flags and Condition Codes

FPSCR bits 15 through 19 are the floating-point result flags. Bit 15 is in FPSCR field 3, and bits 16 through 19 are in FPSCR field 4. For many instructions, FPSCR bits 15 through 19 specify the class and sign of the instruction’s result. For comparison instructions, bits 16 through 19 store the result of the comparison.

| Bit | Meaning |
|---|---|
| 15 | The class descriptor. If this bit is set, the result is either a quiet NaN or a denormalized number, depending on the settings of bits 16 through 19. |
| 16 | < or < 0. For comparison operations, this bit is set if the first operand is less than the second operand. For other operations, this bit is set if the result is negative (< 0). |
| 17 | > or > 0. For comparison operations, this bit is set if the first operand is greater than the second operand. For other operations, this bit is set if the result is positive (> 0). |
| 18 | = or = 0. For comparison operations, this bit is set if the first operand is equal to the second operand. For other operations, this bit is set if the result is 0 (= 0). |
| 19 | Unordered or NaN. For comparison operations, this bit is set if either of the operands is a NaN. For other operations, this bit is set if the result is a NaN or an Infinity, depending on the value of bit 15. |

Table 12-3 shows how bits 15 through 19 are interpreted, depending on whether the previous instruction was a comparison operation or not.

**Table 12-3** Values for FPSCR bits 15 through 19

| Bits 15–19 | Result for comparisons | Result for other operations |
|---|---|---|
| 00001 | Unordered | Not applicable |
| 00010 | == (equal to) | +0 |
| 00100 | > (greater than) | Positive normalized number |
| 00101 | Not applicable | +∞ |
| 01000 | < (less than) | Negative normalized number |

**Table 12-3** Values for FPSCR bits 15 through 19 (continued)

*continued*

| Bits 15–19 | Result for comparisons | Result for other operations |
|---|---|---|
| 01001 | Not applicable | –∞ |
| 10001 | Unordered | Quiet NaN |
| 10010 | == (equal to) | –0 |
| 10100 | > (greater than) | Positive denormalized number |
| 11000 | < (less than) | Negative denormalized number |

## Example: Determining Class

To determine the class of a floating-point operation, copy the FPSCR bits to the Condition Register and then branch on the Condition Register field, as shown in Listing 12-1. To copy FPSCR bits to the Condition Register, use the `mcrfs` instruction, which has the form

```
mcrfs  DST, SRC
```

where *DST* is a 4-bit Condition Register field and *SRC* is an FPSCR field.

**Listing 12-1** Determining the class of an assembler instruction result

```
   fadd  f0,f1,f2    # sets FPSCR bits 15–19 from f0
   mcrfs 2,3         # copy FPSCR bits 12–15 to CR2
   mcrfs 3,4         # copy FPSCR bits 16–19 to CR3
   # CR bits 11 – 15 are class and sign of f0
   bun   3,inf       # if bit 3 of CR3 is 1, result is
                     # Infinity or NaN
   beq   3,zero      # if bit 2 of CR3 is 1, result is zero
   blt   3,norm      # if bit 0 or 1 of CR3 is 1,
   bgt   3,norm      # result is a normalized or
                     # denormalized number

inf:
   bta   11,NaN      # if bit 11 is set, result = quiet NaN
                     # else result is an Infinity
```

```
norm:
   bta   11,denorm   # if bit 11 is set, result is denorm
                     # else result is norm

zero:
   # return class of zero

denorm:
   # return class of denormalized number
```

The `fadd` instruction, which adds two floating-point numbers, is one of the many floating-point instructions that set FPSCR bits 15 through 19 to the class and sign of its result. To read these FPSCR bits, Listing 12-1 copies them to the Condition Register using the `mcrfs` instruction. This instruction operates on 4-bit fields. Bits 15 through 19 are contained in two fields (3 and 4), so two separate `mcrfs` instructions are required to copy all pertinent bits to the Condition Register. Once the bits are copied, Condition Register fields 2 and 3 contain FPSCR fields 3 and 4, which means that Condition Register bits 11 through 15 reflect FPSCR bits 15 through 19. Next, the branch instructions test the values in the Condition Register and determine what type of result the `fadd` instruction had.

## Setting the Rounding Direction

Bits 30 and 31 of the FPSCR specify the current rounding direction, as shown in Table 12-4. The section “Rounding Direction Modes” in Chapter 4, “Environmental Controls,” describes what the different rounding directions do.

**Table 12-4** Rounding direction bits in the FPSCR

| **Mode** | **Bit 30** | **Bit 31** |
|---|---|---|
| To nearest (default) | 0 | 0 |
| Toward zero | 0 | 1 |
| Upward | 1 | 0 |
| Downward | 1 | 1 |

Bits 30 and 31 are in FPSCR field 7.

To set the rounding direction, use the `mtfsfi` instruction. It has the form

```
mtfsfi    DST, n
```

where *DST* is a 4-bit FPSCR field and *n* is an integer value to be copied into *DST*. Here are some examples.

```
mtfsfi   7,0   # set rounding direction to to-nearest
mtfsfi   7,1   # set rounding direction to toward-zero
mtfsfi   7,2   # set rounding direction to upward
mtfsfi   7,3   # set rounding direction to downward
```

## Floating-Point Exceptions

The assembly-language numeric implementation contains the same five floating-point exception flags that are described in the IEEE standard. This section describes how to enable, disable, set, clear, and test these exception flags.

### Exception Bits in the FPSCR

Table 12-5 summarizes the FPSCR bits that control floating-point exceptions. For each bit, it shows which FPSCR field contains that bit. Note that all of these bits, unless otherwise specified, are **sticky;** that is, once set, they stay set until you specifically clear them. For information on exactly what happens when a floating-point exception occurs, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*.

**Table 12-5** Floating-point exception bits in the FPSCR

| Exception | FPSCR field | Bit | Comment |
|---|---|---|---|
| All | 0 | 0 | Exception summary; set if any floating-point exception has occurred |
| | 0 | 1* | Exception enable summary; set if any floating-point exception is enabled |
| Invalid | 0 | 2* | Invalid exception summary; bits 7 through 12 or 21 through 23 tell why the exception occurred |
| | 1 | 7 | Signaling NaN |
| | 2 | 8 | $\infty - \infty$ |
| | 2 | 9 | $\infty / \infty$ |
| | 2 | 10 | $0/0$ |
| | 2 | 11 | $0 \times \infty$ |

**Table 12-5** Floating-point exception bits in the FPSCR (continued)

| Exception | FPSCR field | Bit | Comment |
|---|---|---|---|
| | 3 | 12 | Comparison operation produced invalid |
| | 5 | 21 | Software request produced invalid[†] |
| | 5 | 22 | Square root produced invalid[†] |
| | 5 | 23 | Convert-to-integer operation produced invalid |
| | 6 | 24 | Invalid exception enable/disable |
| Overflow | 0 | 3 | Overflow flag |
| | 6 | 25 | Overflow enable/disable |
| Underflow | 1 | 4 | Underflow flag |
| | 6 | 26 | Underflow enable/disable |
| Divide-by-zero | 1 | 5 | Divide-by-zero flag |
| | 6 | 27 | Divide-by-zero enable/disable |
| Inexact | 1 | 6 | Inexact flag |
| | 3 | 13[*] | Fraction rounded |
| | 3 | 14[*] | Fraction inexact |
| | 6 | 28 | Inexact enable/disable |

[*] This field is not sticky; it applies only for the last instruction executed.
[†] Not implemented in MPC601.

## Signaling and Clearing Floating-Point Exceptions

To signal or clear a floating-point exception explicitly, set or clear its bit in the FPSCR. For example, the following instructions signal an overflow exception and then clear that exception:

```
mtfsb1   3     # sets FPSCR bit 3 to 1, signaling overflow
mtfsb0   3     # clears FPSCR bit 3, so no overflow
```

These two instructions operate on individual FPSCR bits rather than on 4-bit FPSCR fields. The instruction `mtfsb1` sets the specified bit in the FPSCR to 1. The `mtfsb1` instruction shown here sets bit 3, which is the overflow exception flag; therefore this instruction signals that an overflow has occurred. Similarly, the `mtfsb0` instruction sets the specified FPSCR bit to 0 and therefore clears the overflow exception.

## Enabling and Disabling Floating-Point Exceptions

To enable or disable a floating-point exception, set or clear its enable bit in the FPSCR.

**Note**
Disabling a floating-point exception does not mean that its flag will never be set. For the exact meaning of disabling a particular floating-point exception, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. ◆

For example, the following instructions enable and then disable the overflow exception:

```
mtfsb1   25    # sets FPSCR bit 25; overflow enabled
mtfsb0   25    # clears FPSCR bit 25; overflow disabled
```

You can also use the following commands to enable and disable all floating-point exceptions at once:

```
mtfsfi   6,0      # disables all floating-point exceptions
mtfsfi   6,15     # enables all floating-point exceptions
```

As you can see from Table 12-1 on page 12-4, FPSCR field 6 contains all of the floating-point exception enable switches, so to enable or disable all floating-point exceptions at once, you need to set or clear this field. The `mtfsfi` instruction (described on page 12-10) copies a 16-bit signed integer value into an FPSCR field; so the first instruction shown here disables all floating-point exceptions by clearing all bits in field 6, and the second instruction enables all floating-point exceptions by setting all bits in field 6.

**IMPORTANT**
For the FPSCR exception enable bits to be valid, bit 20 or 23 of the Machine State Register must be set. For more information, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. ▲

## Testing for Floating-Point Exceptions

If you would like to see whether an exception occurred, test the Condition Register. Listing 12-2 checks the Condition Register to see if an exception has occurred and, if so, branches to a routine that determines the type of exception. It uses the `fadd.` form of the floating-point add instruction to copy the exception summary bits to Condition Register field 1. If the add instruction causes an exception, this example uses the `mcrfs` instruction (described on page 12-8) to copy the FPSCR fields containing floating-point exception flags to Condition Register fields 2 through 5 and then uses branch instructions to see which type of exception has occurred.

**Listing 12-2** Testing for occurrence of floating-point exceptions

```
   fadd. f0,f1,f2 # f1 + f2 = f0. CR1 contains except.summary
   bta   4,error  # if bit 0 of CR1 is set, go to error
                  # bit 0 is set if any exception occurs
   .              # if clear, continue operation
   .
   .
error:
   mcrfs 2,1   # copy FPSCR bits 4–7 to CR field 2
               # now CR1 and CR2 (bits 6 through 10)
               # contain all exception bits from FPSCR
   bta   6,invalid   # CR bit 6 signals invalid
   bta   7,overflow  # CR bit 7 signals overflow
   bta   8,underflow # CR bit 8 signals underflow
   bta   9,divbyzero # CR bit 9 signals divide-by-zero
   bta   10,inexact  # CR bit 10 signals inexact

invalid:
   mcrfs 2,2   # copy FPSCR bits 8–11 to CR field 2
   mcrfs 3,3   # copy FPSCR bits 12–15 to CR field 3
   mcrfs 4,5   # copy FPSCR bits 20–23 to CR field 4
               # invalid bits are now CR bits 11–16 and bit 23

   # now do exception handling based on which invalid bit
   # is set

overflow:
   # do exception handling for overflow exception

underflow:
   # do exception handling for underflow exception

divbyzero:
   #do exception handling for the divide-by-zero exception

inexact:
   # do exception handling for the inexact exception
```

## Saving and Restoring the Floating-Point Environment

To save and restore the state of the entire floating-point environment, use the `mffs` and `mtfsf` instructions.

The `mffs` instruction saves the FPSCR to a floating-point register. It has the form

```
mffs    DST
```

where *DST* is the floating-point register into which the FPSCR should be copied. For example, the instruction

```
mffs  f0
```

saves the current state of the FPSCR register in bits 32 through 63 of floating-point register F0. Bits 0 through 31 of register F0 are set to 1's.

To restore a floating-point environment that you have previously saved, use the `mtfsf` instruction. This instruction copies a 4-bit field from a floating-point register into an FPSCR field. It has the form

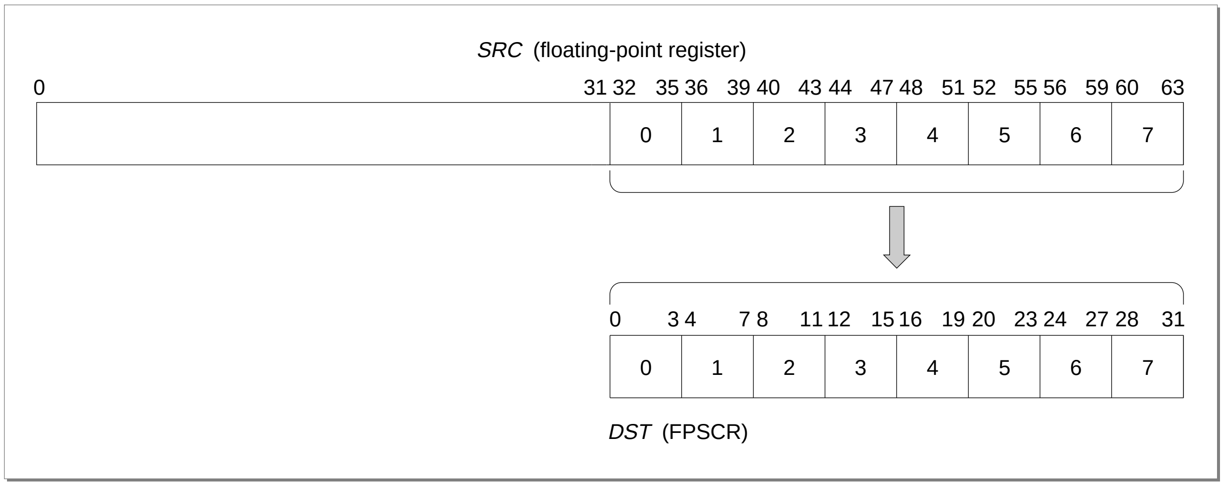
```
mtfsf      DST, SRC
```

where *DST* is a 4-bit FPSCR field and *SRC* is the floating-point register from which the field should be copied. The instruction assumes that the last half of the floating-point register *SRC* contains an FPSCR value. Thus, if you specify

```
mtfsf    3,f0
```

bits 44 through 47 of register F0 are copied into FPSCR field 3, bits 12 through 15. Figure 12-3 shows how the FPSCR fields map to a floating-point register.

**Figure 12-3** *SRC* and *DST* fields for `mtfsf` instruction



Listing 12-3 saves the floating-point environment and then restores it.

**Listing 12-3** Saving and restoring the floating-point environment

```
mffs  f10      # FPSCR copied into register f10

# other floating-point computations occur here

mtfsf 0,f10    # restore bits 0 and 3
mtfsf 1,f10    # restore bits 4 through 7
mtfsf 2,f10    # restore bits 8 through 11
mtfsf 3,f10    # restore bits 12 through 15
mtfsf 4,f10    # restore bits 16 through 19
mtfsf 5,f10    # restore bits 20 through 23
mtfsf 6,f10    # restore bits 24 through 27
mtfsf 7,f10    # restore bits 28 through 31
               # entire FPSCR now restored
```