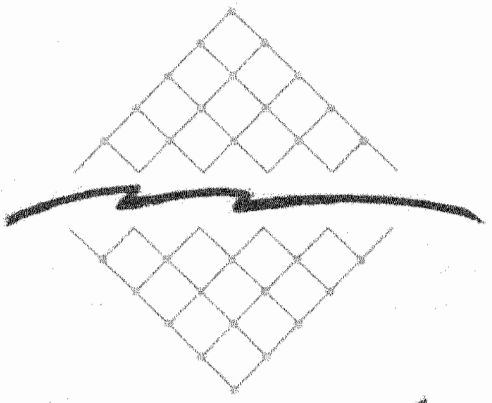
MC68881UM/AD
REV 2



MC68881/882

FLOATING-POINT COPROCESSOR USER'S MANUAL SECOND EDITION

MC68881/MC68882

FLOATING-POINT COPROCESSOR USER'S MANUAL

Second Edition

PRENTICE HALL, Englewood Cliffs, N.J. 07632

This document contains information on a new product. Specifications and information herein are subject to change without notice. Freescale reserves the right to make changes to any products herein to improve functioning or design. Although the information in this document has been carefully reviewed and is believed to be reliable, Freescale does not assume any liabiliy arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of other.

Freescale Semiconductor, Inc. general policy does not recommend the use of its components in life support applications where in a failure or malfunction of the component may directly threaten life or injury. Per Freescale Terms and Conditions of Sale, the user of Freescale components in life support applications assumes all risk of such use and indemnifies Freescale against all damages.

Printed in the United States of America

I 0 9 8 7 6 5 4 3 2 1

ISBN 0-13-567009-8

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana, S.A., Mexico
Prentice-Hall of India Private Limited, New *Delhi*
Prentice-Hall of Japan, Inc., Tokyo
Simon & Schuster Asia Pte. Ltd., Singapore
Editora Prentice-Hall do Brasil, Ltda., *Rio* de Janeiro

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

LIST OF TABLES (Continued)

PREFACE

This manual assumes that the MC68881/MC68882 is connected as a coprocessor to the MC68020/MC68030 microprocessor. If the MC68881/MC68882 is used in a system with a main processor other than the MC68020/MC68030, it is expected that the main processor emulates the M68000 Family coprocessor interface as required by the MC68881/MC68882.

This manual is divided into two major parts. The first part, sections 2 through 8, describes the programmer's model of the MC68881/MC68882 and the floating-point instruction set that it implements. This part of the manual includes a detailed description of each instruction and a section on instruction timing that can be used for program optimization and to predict floating-point arithmetic performance.

The second part of the manual, sections 9 through 13, describes the hardware interface of the MC68881/MC68882 to the main processor, and is most pertinent to system hardware designers. Bus cycle timing diagrams, interface register addressing, etc., are discussed from the viewpoint of the MC68020/MC68030 hardware conventions. A prior knowledge of the MC68020/MC68030 bus interface, particularly as it pertains to the M68000 Family coprocessor interface, is quite helpful in understanding the operation of the MC68881/MC68882 bus interface.

Throughout this manual, M68000 or M68000 Family is used to refer to the family of devices that support the **Freescale** 68000 Family architecture. A number that is preceded by MC, such as MC68020, MC68030, MC68881, or MC68882, refers to a specific part. A reference to MC68881/MC68882 or FPCP applies to either floating-point coprocessor, and a reference to MC68020/MC68030 or MPU applies to either main processor.

The sections and appendices of the manual are:

SECTION 1
GENERAL DESCRIPTION

The MC68881 and MC68882 floating-point coprocessors (FPCP) both fully implement the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI-IEEE Std 754-1985) for use with the Freescale M68000 Family of microprocessors. The coprocessors are both implemented in VLSI technology to give system designers the highest possible functionality in a physically small device. The MC68882 provides an increased level of performance in a coprocessor that is fully compatible and physically interchangeable with the MC68881.

Intended primarily for use as coprocessors to the MC68020/MC68030 32-bit microprocessor unit (MPU), the MC68881 and MC68882 provide a logical extension to the main processing unit integer data processing capabilities. These coprocessors provide a very high performance floating-point arithmetic unit and a set of floating-point data registers utilized in a manner that is analogous to the use of the integer data registers. The MC68881/MC68882 instruction set, a natural extension of all earlier members of the M68000 Family, supports all of the addressing modes of the host MPU. Due to the flexible bus interface of the M68000 Family, the MC68881 or MC68882 can be used with any of the MPU devices of the family and may also be used as a peripheral to other processors.

The major features of the MC68881 and MC68882 are:

- Eight general-purpose floating-point data registers, each supporting a full 80-bit extended precision real data format (a 64-bit mantissa plus a sign bit, and a 15-bit signed exponent).
- A 67-bit arithmetic unit to allow very fast calculations, with intermediate precision greater than the extended precision format.
- A 67-bit barrel shifter for high-speed shifting operations (for normalizing, etc.).
- Forty-six instructions, including 35 arithmetic operations.
- Full conformance to the ANSI-IEEE 754-1985 standard, including all requirements and suggestions.
- Support of functions not defined by the IEEE standard, including a full set of trigonometric and transcendental functions.
- Seven data formats: byte, word, and long word integers; single, double, and extended precision real numbers; and packed binary coded decimal string real numbers.
- Twenty-two constants available in the on-chip ROM, including π, e, and powers of 10.
- Virtual memory/machine operations.
- Efficient mechanisms for exception processing, context switches, and interrupt handling.
- Fully concurrent instruction execution with the main processor.
- Use with any host processor, on an 8-, 16-, or 32-bit data bus.

In addition to these features, the MC68882 provides:

- Concurrent execution of multiple floating-point instructions.
- Special-purpose hardware for high-speed conversion of binary real memory operands to/from the internal extended format.
- Simultaneous access to the floating-point registers by the MC68882's conversion and arithmetic processing units.
- Reduced coprocessor interface overhead to increase throughput.

1.1 THE COPROCESSOR CONCEPT

The FPCP functions as a coprocessor in systems where the MC68020 or MC68030 is the main processor via the M68000 coprocessor interface. It functions as a peripheral processor in systems where the main processor is the MC68000, MC68008, or MC68010.

The FPCP utilizes the M68000 Family coprocessor interface to provide a logical extension of the MPU registers and instruction set in a manner that is transparent to the programmer. The programmer perceives the MPU and FPCP execution model as if both devices were implemented on one chip.

A fundamental goal of the M68000 Family coprocessor interface is to provide the programmer with an execution model based upon sequential instruction execution by the MPU and the FPCP. For optimum performance, however, the coprocessor interface allows floating-point instructions to execute concurrently with MPU integer instructions. Concurrent instruction execution is further extended by the MC68882, which can execute multiple floating-point instructions simultaneously. However, the coprocessor interface and the FPCP are designed to maintain a strictly sequential instruction execution model from the programmer's viewpoint.

The FPCP is a non-DMA type coprocessor that uses a subset of the general-purpose coprocessor interface supported by the MPU. Features of the interface implemented in the FPCP are as follows:

- The main processor and the FPCP communicate via standard M68000 bus cycles.
- Communication between the main processor and the FPCP is not dependent upon the architecture of the individual devices (e.g., instruction pipes or caches, addressing modes).
- The main processor and the FPCP can operate at different clock speeds.
- The FPCP instructions support all addressing modes provided by the main processor.
- All effective addresses calculations and data transfers performed by the main processor at the request of the coprocessor.
- Overlapped (concurrent) instruction execution enhances throughput while maintaining the programmer's model of sequential instruction execution.
- Coprocessor detection of an exception that requires a trap to be taken is serviced by the main processor at the request of the FPCP.
- Support of virtual memory/virtual machine systems is provided via the FSAVE and FRESTORE instructions.

- Up to eight coprocessors can reside in a system simultaneously.
- Multiple coprocessors of the same type are allowed.
- Systems can use software emulation of the FPCP without reassembling or relinking user software.

1.2 HARDWARE OVERVIEW

The MC68881 and MC68882 are high-performance floating-point devices designed to interface with the MC68020 or MC68030 as coprocessors. These coprocessors fully support the MPU virtual machine architecture and are implemented in HCMOS, Freescale's low-power, small-geometry process. This process allows CMOS and HMOS (high-density NMOS) gates to be combined on the same device. CMOS structures are used where speed and low power are required, and HMOS structures are used where minimum silicon area is desired. As a result, HCMOS technology provides the combined advantages of high-performance and low-power consumption without enlarging the die size.

In systems using the MC68000, MC68008, or MC68010 as the main processor, the MC68881 or MC68882 functions as a peripheral processor. The configuration of the FPCP as a peripheral processor or coprocessor can be completely transparent to user software (i.e., the same object code can be executed in either configuration).

The architecture of the FPCP appears to the user as a logical extension of the M68000 Family architecture. Because of the coupling of the coprocessor interface, the MPU programmer can view the FPCP registers as though the registers were resident in the MPU. Thus, an MPU and FPCP device pair appears to be one processor that supports seven floating-point and integer data types with eight integer data registers, eight address registers, and eight floating-point data registers.

The FPCP programming model is shown in Figures 1-1 through 1-6 and consists of the following:

- Eight 80-bit floating-point data registers (FP7–FP0). These registers are analogous to the integer data registers (D7–D0) and are completely general purpose (i.e., any instruction may use any register).
- A 32-bit control register that contains enable bits for each class of exception trap, and mode bits to set the user-selectable rounding and precision modes.
- A 32-bit status register that contains floating-point condition codes, quotient bits, and exception status information.
- A 32-bit instruction address register that contains the main processor memory address (virtual) of the last floating-point instruction that was executed. This address is used in exception handling to locate the instruction that caused the exception.

The connection between the MPU and the FPCP is a simple extension of the M68000 bus interface. The FPCP is connected as a coprocessor to the MPU, and a chip-select signal (decoded from the MPU function codes and address bus) selects the FPCP. Figure 1-7 illustrates the coprocessor/MPU configuration.

As shown in Figure 1-8, the MC68881 is internally divided into two processing elements; the bus interface unit (BIU) and the arithmetic processing unit (APU). The BIU communicates

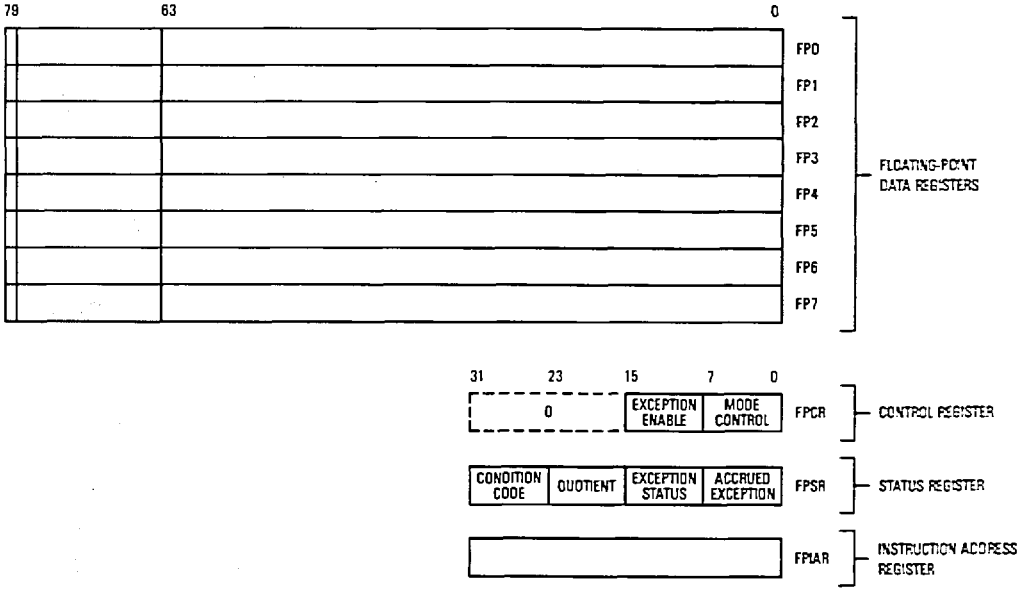


Figure 1-1. MC68881/MC68882 Programming Model

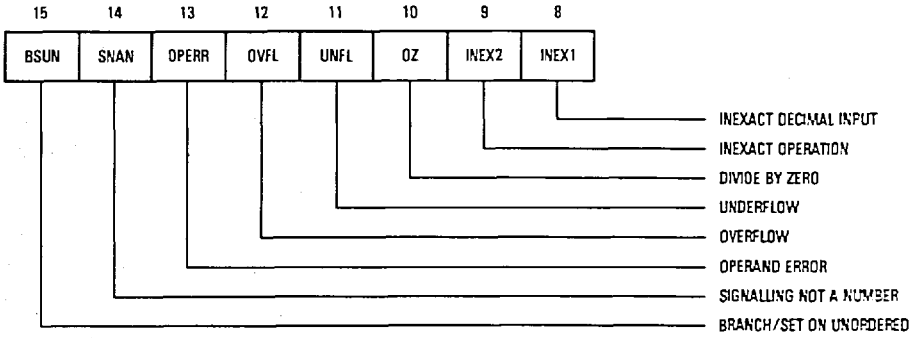


Figure 1-2. Exception Status/Enable Byte

with the MPU, and the APU executes all MC68881 instructions. Though the BIU monitors the state of the APU closely, it operates independently of the APU. The APU operates on the command word and operands that the BIU passes to it. In return, the APU reports its internal status to the BIU.

The BIU contains the coprocessor interface registers (CIRs). In addition to these registers, the CIR register select and $\overline{\text{DSACK}}$ timing control logic is contained in the BIU. Finally, the status flags used to monitor the status of communications with the main processor are contained in the BIU.

The eight 80-bit floating-point data registers (FP7–FP0) and the 32-bit control, status, and instruction address registers (FPCR, FPSR, and FPIAR) are located in the APU. In addition

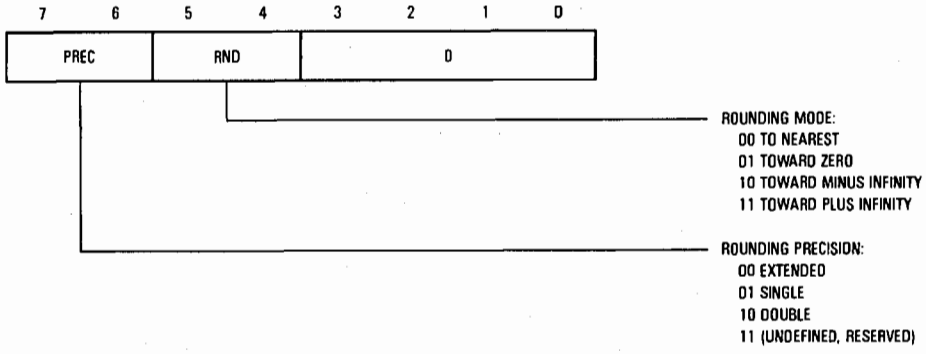


Figure 1-3. Mode Control Byte

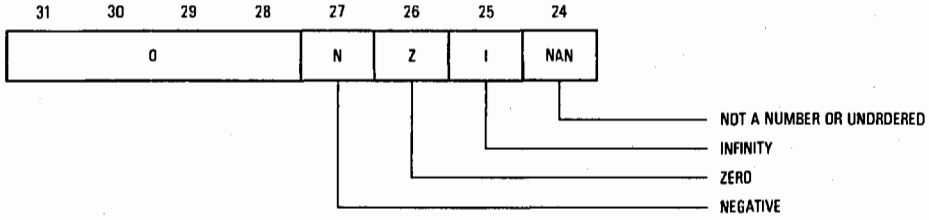


Figure 1-4. Condition Code Byte

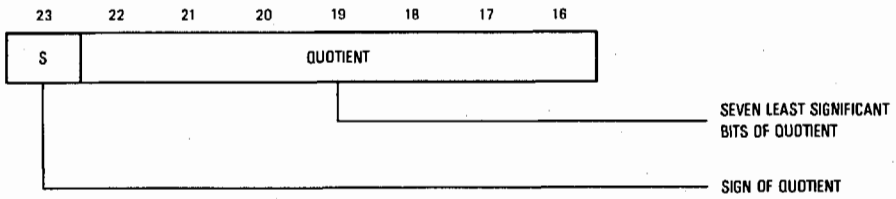


Figure 1-5. Quotient Byte

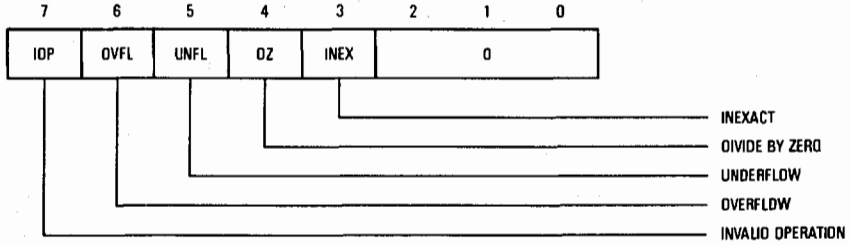


Figure 1-6. Accrued Exception Byte

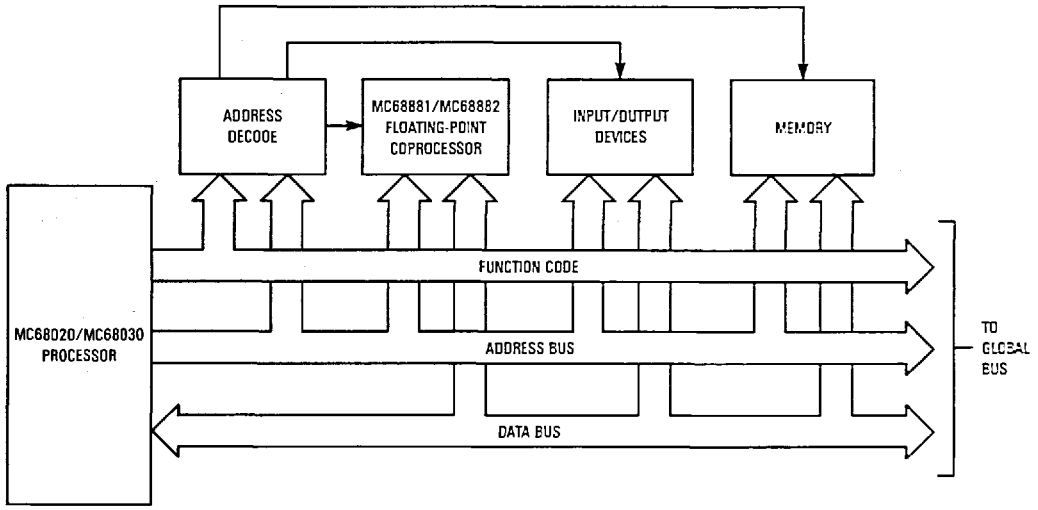


Figure 1-7. Typical Coprocessor Configurations

to these registers, the APU contains a high-speed 67-bit arithmetic unit used for both mantissa and exponent calculations, a barrel shifter that can shift from 1 bit to 67 bits in one machine cycle,and ROM constants (for use by the internal algorithms or user programs).

The control section of the APU contains the clock generator, a two-level microcoded sequencer, the microcode ROM, and self-test circuitry. The built-in self-test capabilities of the FPCP enhance reliability and ease manufacturing requirements; however, these diagnostic functions are not accessible outside of the special test environment supported by VLSI test equipment.

In addition to the BIU and APU, as described for the MC68881 (refer to Figure 1-9), the MC68882 has a conversion unit (CU) that performs data format conversions to the internal extended format. The CU relieves the APU of a significant work load and allows the MC68882 to execute FMOVE instructions concurrently with arithmetic or transcendental operations.

1.2.1 Bus Interface Unit

All communications between the MPU and the FPCP are performed with standard M68000 Family bus transfers. The FPCP is designed to operate on 8-, 16-, or 32-bit data buses.

The FPCP contains a number of coprocessor interface registers (CIRs), which are addressed by the main processor in the same manner as memory. The M68000 Family coprocessor interface is implemented as a protocol of bus cycles in which the main processor reads and writes to these registers. The MPU implements this general-purpose coprocessor interface protocol in hardware and microcode.

When the MPU detects a FPCP general type instruction, the MPU writes the command word of the instruction to the memory-mapped command CIR, and reads the response

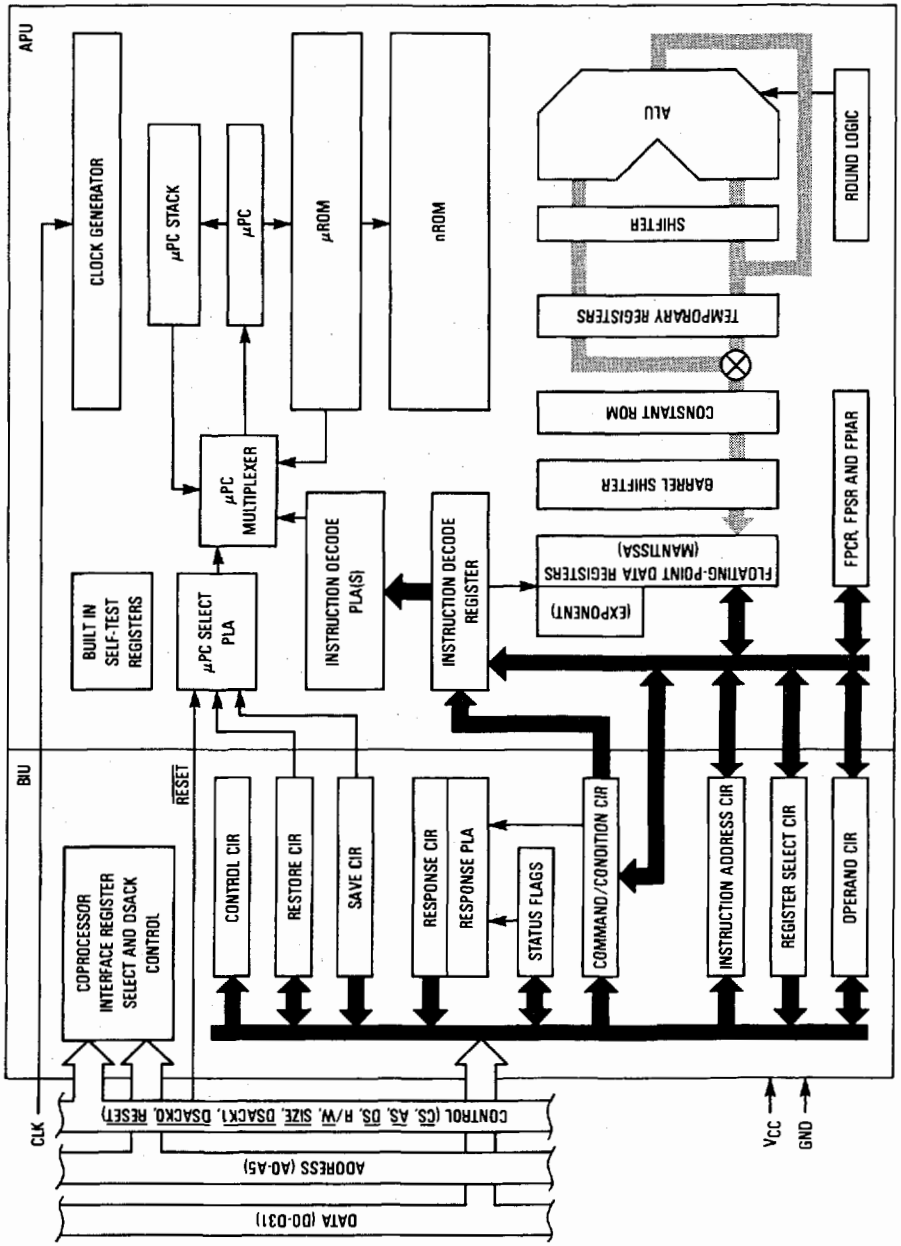


Figure 1-8. MC68881 Simplified Block Diagram

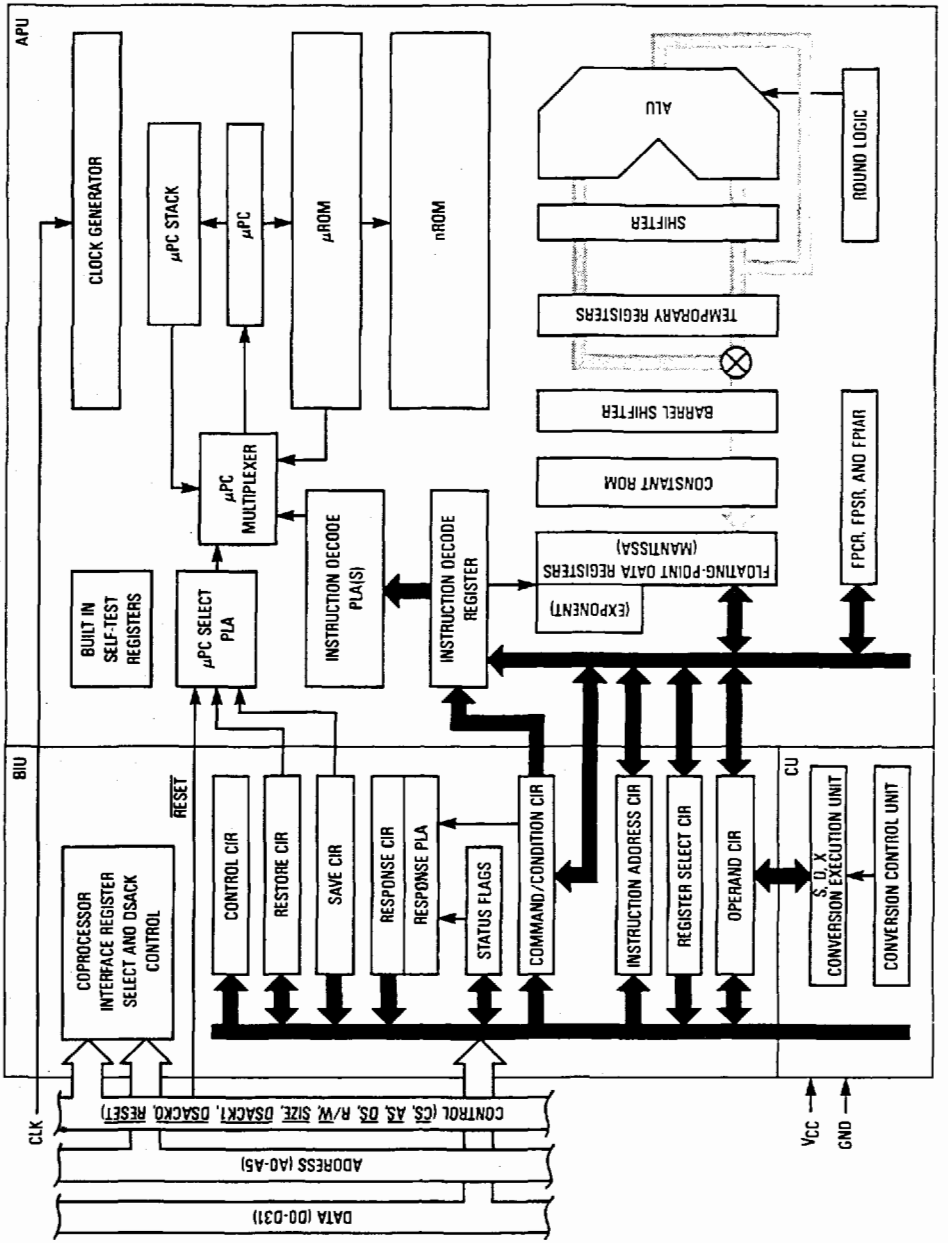


Figure 1-9. MC68882 Simplified Block Diagram

CIR. In this response, the BIU encodes requests for any additional service required of the MPU on behalf of the FPCP. For example, the response may request that the MPU fetch an operand from the evaluated effective address and transfer the operand to the operand CIR. Once the MPU fulfills the coprocessor request(s), the MPU is free to fetch and execute subsequent instructions.

A key concern in a coprocessor interface that allows concurrent instruction execution is synchronization during main processor and coprocessor communication. If a subsequent instruction is written to the command CIR before the APU has completed execution of the previous instruction (in the case of the MC68881) or before the CU has passed its results to the APU (in the case of the MC68882), the response instructs the MPU to wait. Thus, the choice of concurrent or nonconcurrent instruction execution is determined on an instruction-by-instruction basis by the coprocessor.

The only difference between a coprocessor bus transfer and any other bus transfer by the MPU is that the function code issued by the MPU specifies the CPU address space during the cycle (the function codes are generated by the M68000 Family processors to identify one of eight separate address spaces). Thus, the memory-mapped coprocessor interface registers do not infringe upon instruction or data address spaces. The MPU places a coprocessor ID field from the coprocessor instruction words onto three of the upper address lines during coprocessor accesses. This ID, along with the CPU address space function code, is decoded to select one of eight possible coprocessors in the system.

Since the coprocessor interface protocol consists solely of bus transfers, the protocol is easily emulated by software when the FPCP is used as a peripheral with any processor capable of memory-mapped I/O over an M68000-style bus. When used as a peripheral processor with the 8-bit MC68008 or either the 16-bit MC68000 or MC68010, all FPCP instructions are trapped by the main processor to an exception handler at execution time. Trapping the instructions enables the software emulation of the coprocessor interface protocol to be totally transparent to the user. The FPCP can provide a performance option for MC68000-based designs by changing the main processor to an MC68020 or MC68030. The software migrates without change to the next generation equipment using the MPU.

Since the bus is asynchronous, the FPCP need not run at the same clock speed as the main processor. Total system performance can therefore be customized. For a given CPU performance requirement, the floating-point performance can be selected to meet particular price/performance specifications, running the FPCP at slower (or faster) clock speeds than the CPU clock.

1.2.2 Coprocessor Interface

The M68000 Family coprocessor interface is an integral part of the FPCP and MPU designs. The interface partitions MPU and coprocessor operations so that the MPU does not have to completely decode coprocessor instructions, and the FPCP does not have to duplicate main processor functions (such as effective address evaluation).

This partitioning provides an orthogonal extension of the instruction set by permitting FPCP instructions to utilize all MPU addressing modes and to generate execution time exception traps. Thus, from the programmer's view, the MPU and coprocessor appear to be integrated onto a single chip. While the execution of the great majority of FPCP instructions can be overlapped with the execution of MPU instructions, concurrency is completely

transparent to the programmer. The MPU single-step and program flow (trace) modes are fully supported by the FPCP and the M68000 Family coprocessor interface.

While the M68000 Family coprocessor interface permits coprocessors to be bus masters, the FPCP never functions as one. The FPCP requests that the MPU fetch all operands and store all results. In this manner, the MPU 32-bit data bus provides high-speed transfer of floating-point operands and results while reducing the pin count of the FPCP.

Since the coprocessor interface consists solely of bus cycles (to and from the CPU space) and the FPCP never functions as a bus master, the coprocessor can be placed on either the logical or physical side of the system memory management unit (MMU) in an MC68020 system. Since the MMU of the MC68030 is on-chip, the FPCP is always on the physical side of the MMU in an MC68030 system.

The virtual machine architecture of the MPU is supported by the coprocessor interface and the FPCP with the FSAVE and FRESTORE instructions. If the MPU detects a page fault and/or a task timeout, the MPU can force the FPCP to stop whatever operation is in progress at any time and save the FPCP internal state in memory. During the execution of a floating-point instruction, the FPCP can stop at predetermined points as well as at the completion of the instruction.

The size of the saved internal state of the FPCP is dependent upon the state of the APU at the time that the FSAVE is executed. If the MPU is in the reset state when the FSAVE instruction is initiated, only one word of state is transferred to memory. The stored word may be examined by the operating system to determine that the coprocessor programmer's model is empty. If the APU is in the idle state when the FSAVE instruction is decoded, only a few words of internal state are transferred to memory. If the APU is in the middle of executing an instruction, it may be necessary to save the entire internal state of the machine. Instructions that can complete execution in less time than it would take to save the larger state in mid-instruction are automatically allowed to complete execution and then save the idle state. Thus, the size of the saved internal state is kept to a minimum. The ability to utilize several internal state sizes greatly reduces the average context switching time.

The FRESTORE instruction permits reloading an internal state saved earlier and continues any previously suspended operation. Restoring the reset internal state re-establishes default register values, a function identical to the FPCP hardware reset.

1.3 OPERAND DATA FORMATS

The FPCP supports the following data formats:

- Byte Integer (B)
- Word Integer (W)
- Long Word Integer (L)
- Single Precision Real (S)
- Double Precision Real (D)
- Extended Precision Real (X)
- Packed Decimal String Real (P)

The capital letters within the parentheses denote suffixes added to mnemonics of the assembly language instructions to specify the data format to be used.

1.3.1 Integer Data Formats

The three integer data formats (byte, word, and long word) are the standard twos complement data formats defined in the M68000 Family architecture. Whenever an integer is used in a floating-point operation, the integer is automatically converted by the FPCP to an extended precision floating-point number before being used. For example, to add an integer constant of five to the number in floating-point data register 3 (FP3), the following instruction can be used:

```
FADD.W   #5,FP3
```

(The Freescale assembler syntax uses "#" to denote immediate addressing.)

The ability to effectively use integers in floating-point operations saves user memory since an integer representation of a number, if representable, is usually smaller than the equivalent floating-point representation.

1.3.2 Floating-Point Data Formats

The floating-point data formats, single precision (32-bits) and double precision (64-bits), are implemented in the FPCP as defined by the IEEE standard. These data formats are the main floating-point formats and should be used for most calculations involving real numbers. Table 1-1 lists the exponent and mantissa sizes for single, double, and extended precision. The exponent is biased, and the mantissa is in sign and magnitude form. Since single and double precision require normalized numbers, the most significant bit of the mantissa is implied as a one and is not included, thus giving one extra bit of precision.

Table 1-1. Exponent and Mantissa Sizes

Data Format	Exponent Bits	Mantissa Bits
Single	8	23(+1)
Double	11	52(+1)
Extended	15	64

The extended precision data format is also in conformance with the IEEE standard, but the standard does not specify this format to the bit level as it does for single and double precision. The memory format for the FPCP consists of 96 bits (three long words). Only 80 bits are actually used; the other 16 bits are for future expandability and for long-word alignment of floating-point data structures in memory. Extended format has a 15-bit exponent, a 64-bit mantissa, and a 1-bit mantissa sign.

Extended precision numbers are intended for use as temporary variables, intermediate values, or where extra precision is needed. For example, a compiler might select extended precision arithmetic for evaluation of the right side of an equation with mixed sized data and then convert the answer to the data type on the left side of the equation. It is anticipated that extended precision numbers will not be stored in large arrays due to the amount of memory required by each value.

1.3.3 Packed Decimal String Real Data Format

The packed decimal data format allows packed BCD strings to be transferred to and from the FPCP. The strings consist of a 3-digit base 10 exponent and a 17-digit base 10 mantissa.

Both the exponent and mantissa have separate sign bits. All digits are packed BCD, and the entire string fits in 96 bits (three long words). As is the case with all data formats, when packed BCD strings are supplied to the FPCP, the strings are automatically converted to extended precision real values. This conversion allows packed BCD numbers to be used as inputs to any operation. For example:

```
FADD.P   #-6.023E+24,FP5
```

BCD numbers can be supplied by the FPCP in a format readily used for printing by a program generated by a high-level language compiler. For example:

```
FMOVE.P  FP3,BUFFER{#-5}
```

This instruction converts the contents of floating-point data register 3 (FP3) into a packed BCD string with five significant digits to the right of the decimal point (FORTRAN F format).

1.3.4 Data Format Summary

All data formats described in the preceding paragraphs are supported orthogonally by all arithmetic and transcendental operations and by all appropriate MPU addressing modes. For example, all of the following are valid instructions:

```
FADD.B   #0,FP0
FADD.W   D2,FP3
FADD.L   BIGINT,FP7
FADD.S   #3.14159,FP5
FADD.D   (SP)+,FP6
FADD.X   [(TEMP PTR,A7)],FP3
FADD.P   #1.23E25,FP0
```

Most on-chip calculations are performed in the extended precision format, and the eight floating-point data registers always contain extended precision values. All operands are converted to extended precision by the FPCP before a specific operation is performed, and all results are in extended precision. This ensures maximum accuracy without sacrificing performance.

Refer to Figure 1-10 for a summary of the memory formats for the seven data formats supported by the FPCP.

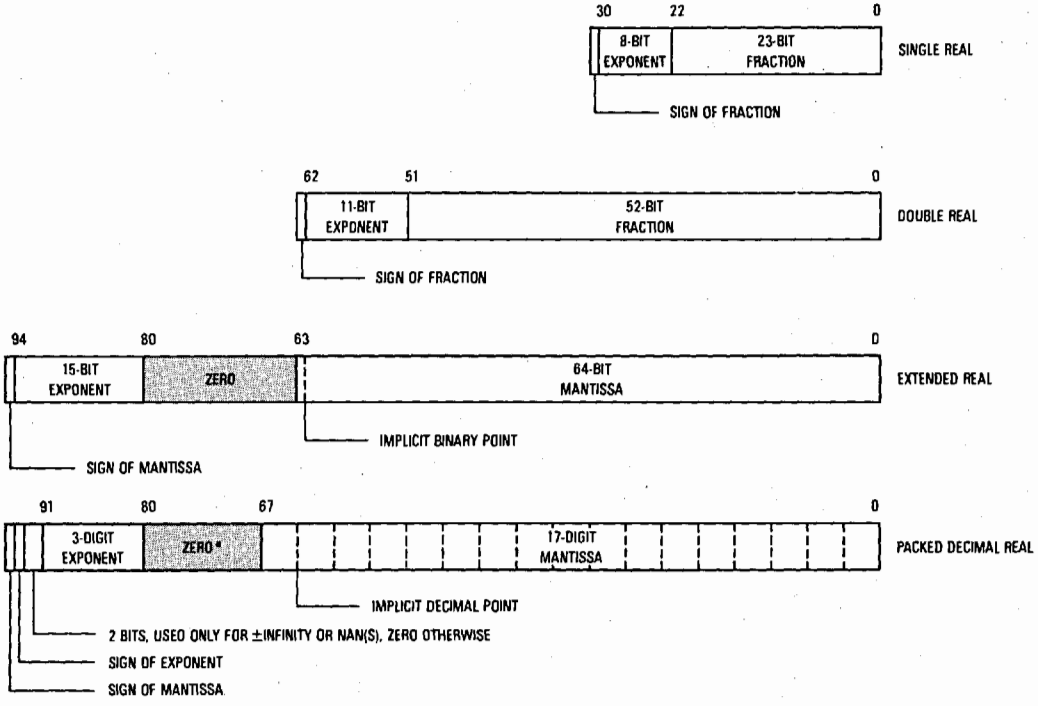
1.4 INSTRUCTION SET

The FPCP instruction set is organized into six major classes:

1. Moves between the FPCP and memory or the MPU (to or from)
2. Move multiple registers (to or from)
3. Monadic operations
4. Dyadic operations
5. Branch, set, or trap conditionally
6. Miscellaneous

1.4.1 Moves

On all moves from memory (or from an MPU data register) to the FPCP, data is converted from the source data format to the internal extended precision format. On all moves from



*Unless a binary-to-decimal conversion overflow occurs

Figure 1-10. MC68881/MC68882 Data Format Summary

the FPCP to memory (or to an MPU data register), data is converted from the internal extended precision format to the destination data format.

Note that data movement instructions perform arithmetic operations, since the result is always rounded to the precision selected in the FPCR mode control byte. The result is rounded using the selected rounding mode and is checked for overflow and underflow.

The syntax for the FMOVE instruction is:

FMOVE.<fmt>	<ea>,FPn	Move to FPCP
FMOVE.<fmt>	FPm,<ea>	Move from FPCP
FMOVE.X	FPm,FPn	Move within FPCP

where:

<ea> is an MPU effective address operand
.<fmt> is the data format size
FPm and FPn are floating-point data registers.

1.4.2 Move Multiple Registers

The floating-point move multiple instruction on the FPCP resembles its integer counterpart on the M68000 Family processors. Any set of the floating-point registers FP0 through FP7

can be moved to or from memory with one instruction. These registers are always moved as 96-bit extended data with no conversion (hence no possibility of conversion errors). Some examples of the move multiple instruction are as follows:

```
FMOVEM    <ea>,FP0-FP3/FP7
FMOVEM    FP2/FP4/FP6,<ea>
```

The move multiple instruction is useful during context switches and interrupts to save or restore the state of a program. It is also useful at the start and end of a procedure to save and restore the calling routine's register set. In order to reduce procedure call overhead, the list of registers to be saved or restored can be stored in a data register thus enabling run-time optimization by allowing a called routine to save as few registers as possible. Note that no rounding or overflow/underflow checking is performed by these operations.

1.4.3 Monadic Operations

A monadic operation has one operand. This operand may be in a floating-point data register, in memory, or in an MPU data register. The result is always stored in a floating-point data register. For example, the syntax for square root is any of the following:

```
FSQRT.<fmt>    <ea>,FPn
FSQRT.X        FPm,FPn
FSQRT.X        FPn
```

The monadic operations available with the FPCP are as follows:

FABS	Absolute Value	FLOG2	Log Base 2
FACOS	Arc Cosine	FLOGN	Log Base e
FASIN	Arc Sine	FLOGNP1	Log Base e of (x+1)
FATAN	Arc Tangent	FNEG	Negate
FATANH	Hyperbolic Arc Tangent	FSIN	Sine
FCOS	Cosine	FSINCOS	Simultaneous Sine and Cosine
FCOSH	Hyperbolic Cosine	FSINH	Hyperbolic Sine
FETOX	e to the x Power	FSQRT	Square Root
FETOXM1	e to the x Power −1	FTAN	Tangent
FGETEXP	Get Exponent	FTANH	Hyperbolic Tangent
FGETMAN	Get Mantissa	FTENTOX	10 to the x Power
FINT	Integer Part	FTST	Test
FINTRZ	Integer Part (Truncated)	FTWOTOX	2 to the x Power
FLOG10	Log Base 10		

1.4.4 Dyadic Operations

Dyadic operations have two operands each. The first operand is in a floating-point data register, memory, or an MPU data register. The second operand is the contents of a floating-point data register. The destination is the same floating-point data register used for the second operand. For example, the syntax for floating-point add is either of the following:

```
FADD.<fmt>    <ea>,FPn
FADD.X        FPm,FPn
```

The dyadic operations available with the FPCP are as follows:

FADD	Add	FREM	IEEE Remainder
FCMP	Compare	FSCALE	Scale Exponent

FDIV	Divide	FSGLDIV	Single Precision Divide
FMOD	Modulo Remainder	FSGLMUL	Single Precision Multiply
FMUL	Multiply	FSUB	Subtract

Assuming that operands are single precision, the FSGLMUL and FSGLDIV instructions round results as such while maintaining the range of extended precision. In special applications where multiply and divide performance are more important than loss of precision, the FSGLMUL and FSGLDIV instructions can be used.

1.4.5 Branch, Set, and Trap-On Condition

The floating-point branch, set, and trap-on condition instructions implemented by the FPCP are similar to the equivalent integer instructions of the M68000 Family processors, except more conditions exist due to the special values in IEEE floating-point arithmetic. When a conditional instruction is executed, the FPCP performs the necessary condition checking and reports the result, true or false, to the MPU; the MPU then takes the appropriate action. Since the FPCP and MPU are closely coupled, the floating-point branch operations are quickly executed.

The FPCP conditional operations are:

FBcc	Branch
FDBcc	Decrement and Branch
FScc	Set According to Condition
FTRAPcc	Trap-on Condition (with an Optional Parameter)

where:

cc is one of the 32 floating-point conditional test specifiers listed in **3.3 PACKED DECIMAL REAL DATA FORMAT**.

1.4.6 Miscellaneous Instructions

Miscellaneous instructions include moves to and from the status, control, and instruction address registers. Also included are the virtual memory/machine FSAVE and FRESTORE instructions that save and restore the internal state of the FPCP.

FMOVE	<ea>,FPcr	Move to Control Register(s)
FMOVE	FPcr,<ea>	Move from Control Register(s)
FSAVE	<ea>	Virtual Machine State Save
FRESTORE	<ea>	Virtual Machine State Restore

1.5 ADDRESSING MODES

The FPCP does not perform address calculations. Thus, when the FPCP instructs the MPU to transfer an operand via the coprocessor interface, the MPU performs the addressing mode calculations requested in the instruction. In this case, the instruction is encoded specifically for the MPU, and the instruction execution by the FPCP is dependent only on the value of the command word written to the FPCP by the main processor.

This interface is quite flexible and allows any addressing mode to be used with floating-point instructions. For the M68000 Family, these addressing modes include immediate, postincrement, predecrement, data or address register direct, and the indexed/indirect

addressing modes of the MPU. Some addressing modes are restricted for some instructions in keeping with the M68000 Family architectural definitions (e.g., program counter relative addressing is not allowed for a destination operand).

The orthogonal instruction set of the FPCP, along with the flexible branches and addressing modes of the MPU, allows a programmer or a compiler writer to think of the FPCP as though it were part of the MPU. There are no special restrictions imposed by the coprocessor interface, and floating-point arithmetic is coded exactly like integer arithmetic.

1.6 MC68882 PROGRAMMING CONSIDERATIONS

To exploit the enhanced performance of the MC68882 requires the programmer to be aware of the manner in which the coprocessor overlaps execution of instructions. Upgrading a system to use the MC68882 requires minor system software changes but no user software changes. To optimize applications code for the MC68882 may require reordering of floating-point instructions. **SECTION 5 COPROCESSOR PROGRAMMING** describes the concurrency capabilities of the MC68882, the required system software changes, and the optimization of existing software for the enhanced floating-point coprocessor.

SECTION 2
PROGRAMMING MODEL

This section describes the registers of the MC68881/MC68882 (FPCP) programming model. The notation used to refer to the registers conforms to the Freescale assembler syntax. The programming model for the MC68882 is identical to that for the MC68881.

Figure 2-1 is a pictorial representation of the registers in the FPCP programming model. The following paragraphs describe each group of registers.

2.1 FLOATING-POINT DATA REGISTERS

The eight 80-bit floating-point data registers (FP7–FP0) are analogous to the integer data registers (D7–D0) of all M68000 Family processors. Floating-point data registers always contain extended precision numbers. The data format used is identical to the extended precision data format described in Table 3-3, except that the reserved (unused) 16 bits are deleted from the table. All external operands, regardless of the data format, are converted to extended precision values before being used in any calculation or stored in a floating-point data register.

A reset function or a restore operation of the null state sets FP7–FP0 to positive nonsignaling not-a-numbers (NANs), described in **3.2.5 Not-A-Numbers.**

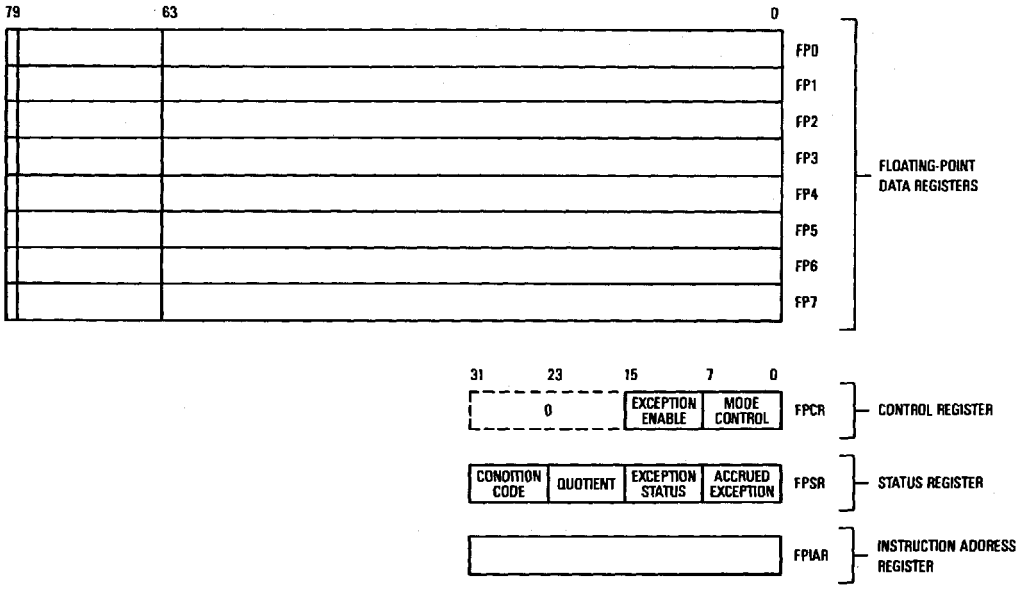


Figure 2-1. MC68881/MC68882 Programming Model

2.2 FLOATING-POINT CONTROL REGISTER

The 32-bit floating-point control register (FPCR) contains an exception enable byte that enables/disables traps for each class of floating-point exceptions and a mode byte that sets the user selectable modes.

The control register can be read or written to by the user. Bits 31–16 are reserved for future definition by Freescale. These bits are always read as zero and are ignored during write operations (but should be zero for future compatibility). This register is cleared by the reset function or a restore operation of the null state. When cleared, this register provides the IEEE standard defaults.

2.2.1 FPCR Exception Enable Byte

One of the bits of the exception enable byte (ENABLE), shown in Figure 2-2, corresponds to each floating-point exception class. The user can separately enable traps for each class of floating-point exceptions.

If a bit in the status register exception byte is set by the FPCP and the corresponding bit in the control register ENABLE byte is also set, an exception is signaled. The address of the exception handler is derived from the vector address corresponding to the exception. When a user writes to the control register ENABLE byte that enables a class of floating-point exceptions, a previously generated floating-point exception does not cause a trap to be taken regardless of the value in the status register exception byte.

The eight floating-point exception classes shown in Figure 2-2 are described in greater detail in **SECTION 6 EXCEPTION PROCESSING**. Note that the bits in the FPSR exception byte and the FPCR enable byte occupy the same positions within each byte.

In a few cases, dual and triple exceptions can be generated by a single instruction execution. When multiple exceptions occur with traps enabled for more than one exception class, the highest priority exception is reported; the lower priority exceptions are never reported or taken. The exception handler routine must check for multiple exceptions. The bits of the

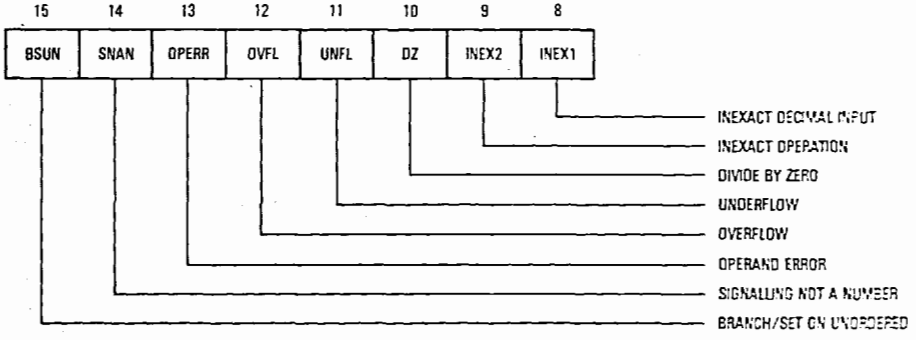


Figure 2-2. MC68881/MC68882 FPCR Exception Enable Byte

ENABLE byte are organized in decreasing priority, left to right, i.e., BSUN is the highest priority, and INEX1 is the lowest priority. The only multiple exception possibilities are:

SNAN and INEX1
OPERR and INEX2
OPERR and INEX1
OVFL and INEX2 and/or INEX1
UNFL and INEX2 and/or INEX1
INEX2 AND INEX1

2.2.2 FPCR Mode Control Byte

The mode control byte (MODE), shown in Figure 2-3, controls the user-selectable rounding modes and rounding precisions. A zero in this byte selects the IEEE defaults.

The rounding mode specifies how inexact results are rounded. "Round to the nearest" specifies that the nearest number to the infinitely precise result should be selected as the rounded value. In case of a tie, the even result is selected. "Round towards zero" truncates the result. "Round towards plus infinity" always rounds numbers towards plus infinity. "Round toward minus infinity" always rounds numbers towards minus infinity. See **6.1.7 Inexact Result** for a detailed description of the rounding algorithm used.

The rounding precision selects where rounding of the mantissa occurs. For extended precision, the result is rounded to a 64-bit boundary. A single precision result is rounded to a 24-bit boundary, and a double precision result is rounded to a 53-bit boundary.

The single and double rounding precisions are provided for emulation of machines that only support those precisions. When the FPCP performs any operation, the calculation is carried out using extended precision inputs and the intermediate result is calculated as if to produce infinite precision. After the calculation is complete, this intermediate result is rounded to the selected precision and stored in the destination.

If the destination is a floating-point data register, the stored value is in the extended precision format rounded to the precision specified by the PREC bits. Thus, all mantissa bits beyond the selected precision are zero after the rounding operation. Also, if the single

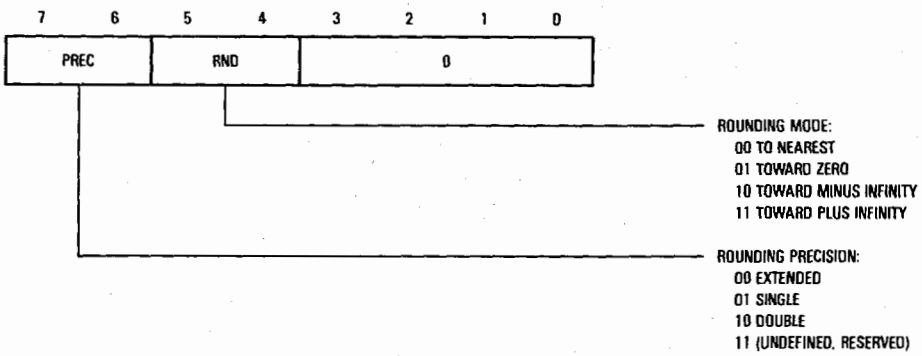


Figure 2-3. MC68881/MC68882 FPCR Mode Control Byte

or double precision mode is selected, the exponent value is in the correct range for the single or double precision format (although it is stored in extended precision format). An important exception to this rule is for the FSGLDIV and FSGLMUL instructions. Regardless of the precision specified by the PREC bits, these instructions round the result mantissa to single precision and generate an extended precision exponent which may be out of range for a single precision number.

If the destination is a memory location, the PREC bits are ignored. In this case, a number in the extended precision format is taken from the source floating-point data register, rounded to the destination format precision, and written to memory.

The execution speed of all instructions is degraded significantly when single and double precision rounding modes are used. Because these modes are intended to be used for emulation, this reduction is not detrimental. When operating in these modes, the FPCP produces the same result as any other machine that conforms to the IEEE standard without supporting extended precision calculations. However, the result obtained by performing a series of operations with single or double precision rounding may not be the same as the result of performing the same operations in extended precision and storing the final result in the single or double precision format.

2.3 FLOATING-POINT STATUS REGISTER

The floating-point status register (FPSR) contains a floating-point condition code byte, a floating-point exception status byte, quotient bits, and a floating-point accrued exception byte. All bits in the FPSR can be read or written by the user. Execution of most floating-point instructions modifies this register.

The reset function or a restore operation of the null state clears the FPSR.

2.3.1 FPSR Floating-Point Condition Code Byte

The floating-point condition code byte (FPCC), shown in Figure 2-4, contains four condition code bits that are set at the end of all arithmetic instructions involving the floating-point data registers. The FMOVE FPm,<ea>, move multiple floating-point data register, and move system control register instructions do not affect the FPCC.

The operation result data type determines how the four condition code bits are set. Table 2-1 lists the condition code bit settings for each result data type. Because of the mutually exclusive nature of the data types described by the condition code bits, the FPCP generates

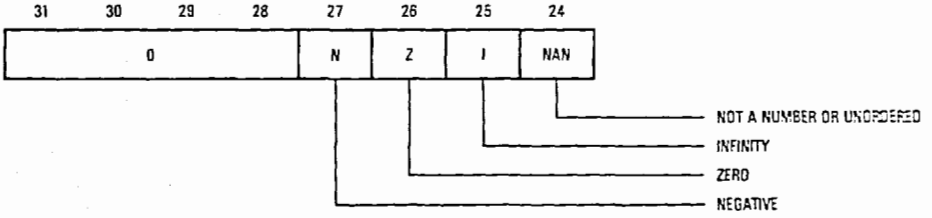


Figure 2-4. MC68881/MC68882 FPSR Condition Code Byte

Table 2-1. Condition Code versus Result Data Type

N	Z	I	NAN	Result Data Type
0	0	0	0	+ Normalized or Denormalized
1	0	0	0	− Normalized or Denormalized
0	1	0	0	+0
1	1	0	0	−0
0	0	1	0	+ Infinity
1	0	1	0	− Infinity
0	0	0	1	+ NAN
1	0	0	1	− NAN

only eight of the 16 possible combinations. Loading the FPCC byte with one of the other condition code bit combinations and executing a conditional instruction may produce an unexpected branch condition.

The IEEE standard defines the following four conditions and only requires the generation of the condition codes as a result of a floating-point compare operation. In addition to this requirement, the FPCP can test these conditions at the end of any operation affecting the condition codes.

EQ —Equal To
GT —Greater Than
LT —Less Than
UN —Unordered

An unordered condition occurs when one or both of the operands in a floating-point compare operation is a NAN. For purposes of the floating-point conditional branch, set byte on condition, decrement and branch on condition, and trap-on condition instructions, the FPCP logically combines the four condition codes to form the IEEE conditions according to the following equations:

$EQ = Z$
$GT = \overline{N \vee NAN \vee Z}$
$LT = N \wedge \overline{NAN \vee Z}$
$UN = NAN$

where:

"$\wedge$" = Logical AND
"$\vee$" = Logical OR

Note that the setting of the FPCP condition codes is independent of the operation executed; the condition codes only indicate the data type of the result generated. Unlike other M68000 condition codes, the IEEE defined conditions can always be derived from the data type of the result. The setting of the M68000 integer condition codes is dependent upon the operation executed as well as the result.

To aid programmers of floating-point subroutine libraries, the FPCP implements the four previously described floating-point condition code bits in hardware instead of the four IEEE defined conditions. The IEEE conditions are derived by an instruction when needed. For example, the programmer of a complex arithmetic multiply subroutine usually prefers to handle "special" data types such as zeros, infinities, or NANs, separately from "normal" data types. The FPCP condition codes allow users to efficiently detect and handle these "special" values.

2.3.2 FPSR Quotient Byte

The quotient byte (see Figure 2-5) is set at the completion of the modulo (FMOD) or IEEE remainder (FREM) instructions. This byte contains the seven least significant bits of the quotient (unsigned) and the sign of the entire quotient.

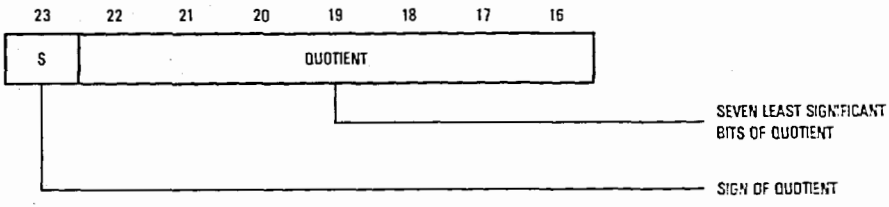


Figure 2-5. MC68881/MC68882 FPSR Quotient Byte

The quotient bits can be used in argument reduction for transcendentals and other functions. For example, seven bits are more than enough to determine the quadrant of a circle in which an operand resides. The quotient bits remain set until they are cleared by the user, or until another FMOD or FREM instruction is executed.

2.3.3 FPSR Exception Status Byte

The exception status byte (EXC), shown in Figure 2-6, contains a bit for each floating-point exception that may have occurred during the most recent arithmetic instruction or move operation. This byte is cleared by the FPCP at the start of most operations; operations that cannot generate any floating-point exceptions (the FMOVEM and FMOVE control register instructions) do not clear this byte. This byte can be used by an exception handler to determine which floating-point exception(s) caused a trap.

If a bit is set by the FPCP in the EXC byte and the corresponding bit in the ENABLE byte is also set, an exception is signaled to the main processor. When a floating-point exception is detected by the FPCP, the corresponding bit in the EXC byte is set, even if the trap for

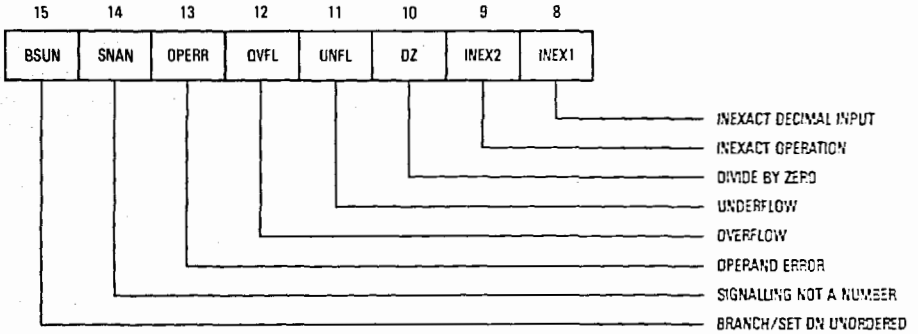


Figure 2-6. MC68881/MC68882 FPSR Exception Status Byte

that exception class is disabled. (A user write operation to the status register, which sets a bit in the EXC byte, does not cause a trap to be taken regardless of the value in the ENABLE byte.)

Note that the bits in the status EXC byte and control ENABLE byte are in the same bit positions within each byte. The eight floating-point exception classes are described in greater detail in **SECTION 6 EXCEPTION PROCESSING**.

2.3.4 FPSR Accrued Exception Byte

The accrued exception byte (AEXC), shown in Figure 2-7, contains the five exception bits required by the IEEE standard for trap disabled operation. These exceptions are logical combinations of the bits in the EXC byte. The AEXC byte contains the history of all floating-point exceptions that have occurred since the user last cleared the AEXC byte. In normal operations, only the user clears this byte (by writing to the status register). The AEXC byte is cleared by the FPCP only by a reset or a restore operation of the null state.

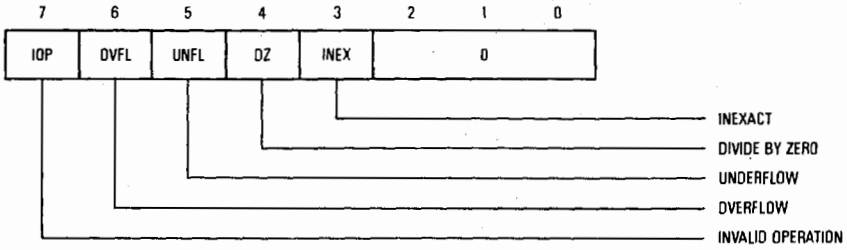


Figure 2-7. MC68881/MC68882 FPSR Accrued Exception Byte

Many users elect to disable traps for all or part of the floating-point exception classes. The AEXC byte is provided to make it unnecessary to poll the EXC byte after each floating-point instruction. At the end of most operations (all but the FMOVEM and FMOVE control register instructions), the bits in the EXC byte are logically combined to form an AEXC value that is logically ORed into the existing AEXC byte. This operation creates "sticky" floating-point exception bits in the AEXC byte that the user need poll only once (at the end of a series of floating-point operations, for example).

The setting or clearing of bits in the AEXC byte does not cause the FPCP to take an exception, nor does it prevent taking an exception. The relationship between the bits in the EXC byte and the bits in the AEXC byte is shown by the following equations. These equations apply to setting the AEXC bits at the end of each operation that affects the AEXC byte:

AEXC(IOP) = AEXC(IOP)vEXC(BSUNvSNANvOPERR)
AEXC(OVFL) = AEXC(OVFL)vEXC(OVFL)
AEXC(UNFL) = AEXC(UNFL)vEXC(UNFLΛINEX2)
AEXC(DZ) = AEXC(DZ)vEXC(DZ)
AEXC(INEX) = AEXC(INEX)vEXC(INEX1vINEX2vOVFL)

where:
"v" = Logical OR
"Λ" = Logical AND

2.4 FLOATING-POINT INSTRUCTION ADDRESS REGISTER

A majority of the FPCP instructions operate concurrently with the MC68020/MC68030 (MPU). That is, the MPU can be executing instructions while the FPCP is simultaneously executing a floating-point instruction. Additionally, the MC68882 can execute two floating-point instructions concurrently. As a result of this nonsequential instruction execution, the program counter value stacked by the MPU, in response to an enabled floating-point exception trap may not point to the offending instruction.

For the subset of the FPCP instructions that generate floating-point exception traps, the 32-bit floating-point instruction address (FPIAR) register is loaded with the logical address of an instruction before the instruction is executed (unless all arithmetic exceptions are disabled). This address can then be used by a floating-point exception handler to locate a floating-point instruction that has caused an exception. Since the FPCP FMOVE to/from the FPCR, FPSR, or FPIAR and FMOVEM instructions cannot generate floating-point exceptions, these instructions do not modify the FPIAR. These instructions can be used to read the FPIAR in the trap handler without changing the previous value.

This register is cleared by the reset operation or a restore operation of the null state.

SECTION 3
OPERAND DATA FORMATS

The following paragraphs describe the MC68881/MC68882 (FPCP) operand data formats. Seven data formats are supported: three signed binary integer formats, three binary floating-point formats, and one packed binary-coded decimal (BCD) floating-point format. All data formats are supported uniformly by all arithmetic and transcendental instructions. These formats are as follows:

Byte Integer (B)
Word Integer (W)
Long Word Integer (L)
Single Precision Real (S)
Double Precision Real (D)
Extended Precision Real (X)
Packed Decimal Real (P)

The capital letter in parentheses is the suffix added to an instruction in the assembly language syntax to specify the data format of operands external to the FPCP. All data formats are organized in memory consistently with the M68000 Family data organization, i.e., the most significant byte is located at the lowest address (nearest $00000000), with each successively less significant byte located at the next address (N+1, N+2, etc.). The least significant byte is located at the highest address (nearest $FFFFFFFF).

Within the floating-point data formats, there are five types of numbers that can be represented: normalized numbers, denormalized numbers, zeros, infinities, and not-a-numbers (NANs). These data types are represented with special encodings corresponding to each data format.

3.1 INTEGER DATA FORMATS

The three signed (twos complement) integer data formats supported by the FPCP (byte, word, and long word) are identical to those supported by the M68000 Family architecture (see Figure 3-1).

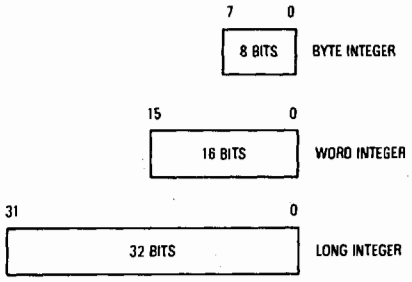


Figure 3-1. Signed Integer Data Formats

Since all FPCP internal operations are performed in full extended precision format, signed integer operands are converted to extended precision values before the specified operation is performed. Thus, mixed mode arithmetic is implicitly supported.

3.2 BINARY REAL DATA FORMATS

Floating-point numbers can be encoded in any of three binary real data formats: single precision (32 bits), double precision (64 bits), and double-extended precision (96 bits, 80 of which are used). All three of these formats fully comply with the *IEEE Standard for Binary Floating-Point Arithmetic*.

NOTE

The single-extended precision data format defined in the IEEE standard is redundant in a device that supports the double-extended precision format. Thus, all references in this manual to extended precision imply double-extended precision as defined by the IEEE standard.

Since all FPCP internal operations are performed in extended precision, single and double precision operands are converted to extended precision values before the specified operation is performed. Thus, mixed mode arithmetic is implicitly supported. The memory formats for the real data formats are shown in Figure 3-2.

The exponent in all three binary formats is an unsigned binary integer with an implied bias added to it. The bias values for single, double, and extended precision are 127, 1023, and 16383, respectively. When the bias is subtracted from the value of the exponent, the result represents a signed twos-complement power of two that yields the magnitude of a normalized floating-point number when multiplied by the mantissa. Since biased exponents are used, a program can execute an integer compare instruction (CMP) to compare floating-point numbers in memory (regardless of the absolute magnitude of the exponents).

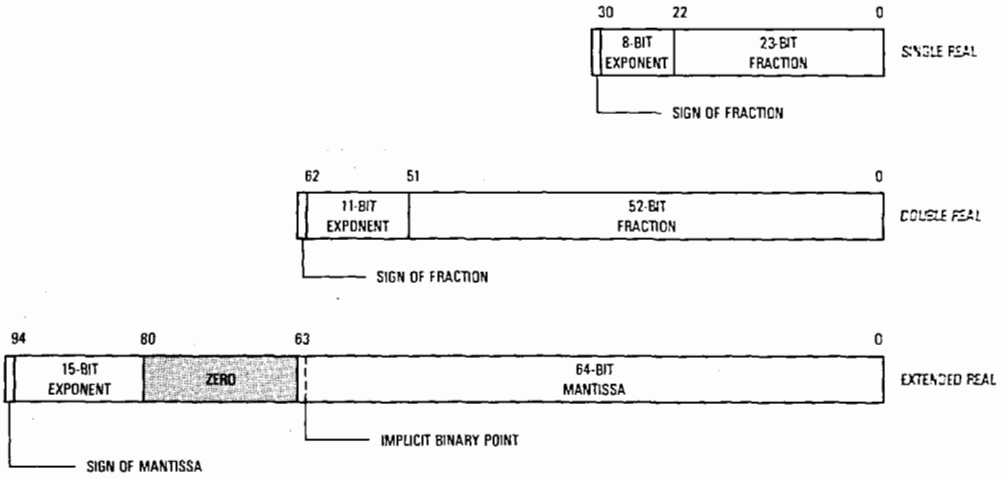


Figure 3-2. Binary Real Data Formats

Data formats for single and double precision numbers differ slightly from the data formats for extended precision numbers in the representation of the mantissa. A normalized mantissa, for all three precisions, is always in the range [1.0 . . . 2.0). The extended precision data format explicitly represents the entire mantissa, including the explicit integer part bit. However, for single and double precision data formats, only the fractional portion of the mantissa is explicitly represented and the integer part, always one, is implied.

The IEEE standard has created the term "significand" to bridge this difference and to avoid the historical implications of the term mantissa. The IEEE standard defines a significand as the component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of the implied binary point. This manual uses the terms mantissa and significand, defined as follows, interchangeably.

Single Precision Mantissa = Single Precision Significand
= 1.<23-Bit Fraction Field>
Double Precision Mantissa = Double Precision Significand
= 1.<52-Bit Fraction Field>
Extended Precision Mantissa = Extended Precision Significand
= 1.Fraction
= <64-Bit Mantissa Field>

NOTE

Throughout this manual, ranges are specified using traditional set notation with the format "bound . . . bound" specifying the boundaries of the range. The type of brackets enclosing the range defines whether the endpoint is inclusive or exclusive. A square bracket indicates inclusive, and a parenthesis indicates exclusive. For example, the range specification "[1.0...2.0]" defines the range of numbers greater than or equal to 1.0 and less than or equal to 2.0. The range specification "(0.0... + inf]" defines the range of numbers greater than 0.0 and less than or equal to positive infinity.

Each of the three floating-point data formats can represent five unique floating-point data types:

Normalized Numbers
Denormalized Numbers
Zeros
Infinities
Not-A-Numbers (NANs)

The normalized data type never uses the maximum or minimum exponent value for a given format (except for the extended precision format, see following note). These exponent values in each precision are reserved for representing the special data types: zeros, infinities, denormalized numbers, and NANs. Details of each type of number for each format are shown in **3.6 DATA FORMAT DETAILS.**

NOTE

There is a subtle difference between the definition of an extended precision number with an exponent equal to zero and a single or double precision number with an exponent equal to zero. If the exponent of a single or double precision number is zero, the number is defined to be denormalized, and the implied integer bit is also a zero. However, an extended precision number with an exponent of zero may have an explicit integer bit equal to one, which results in a normalized number (even though the exponent is equal to the minimum value).

For simplicity, the following discussion treats all three real formats in the same manner, where an exponent value of zero identifies a denormalized number. However, it should be noted that the extended precision format may deviate from this rule.

3.2.1 Normalized Numbers

Normalized numbers encompass all representable real values between the overflow and underflow thresholds, i.e., those numbers whose exponents lie between the maximum and minimum values. Normalized numbers may be positive or negative. For normalized numbers, the implied integer part bit in single and double precision is a one (1). In extended precision, the integer bit is explicitly a one (1). See Figure 3-3.

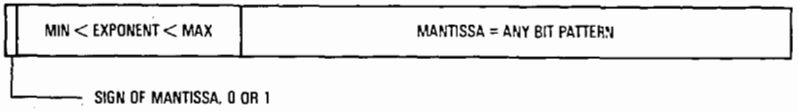


Figure 3-3. Format of Normalized Numbers

3.2.2 Denormalized Numbers

Denormalized numbers represent real values near the underflow threshold (underflow is detected for a given data format and operation when the result exponent is less than or equal to the minimum exponent value). Denormalized numbers may be positive or negative. For denormalized numbers, the implied integer part bit in single and double precision is a zero (0). In extended precision, the integer bit is explicitly a zero (0). See Figure 3-4.

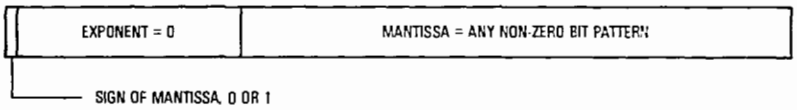


Figure 3-4. Format of Denormalized Numbers

Traditionally, floating-point number systems perform a "flush-to-zero" when underflow is detected. This leaves a large gap in the number line between the smallest magnitude normalized number and zero. The IEEE standard implements gradual underflows: the result mantissa is shifted right (denormalized) while the result exponent is incremented until the result exponent reaches the minimum value. If all mantissa bits of the result are shifted off to the right during this denormalization, the result becomes zero. In many instances, gradual underflow limits the potential underflow damage to no more than a round-off error. (This underflow and denormalization description ignores the effects of rounding and the user selectable rounding modes.) Thus, the large gap in the number line created by "flush-to-zero" floating-point number systems is filled with representable (denormalized) numbers in the IEEE "gradual underflow" floating-point number system.

NOTE

Since the extended precision data format has an explicit integer part bit, a number can be formatted with a nonzero exponent (less than the maximum value) and a zero integer bit, which is not defined by the IEEE standard. Such a number is called an unnormalized number. The MC68881 never generates an unnormalized number as the result of any operation. Unnormalized inputs are always converted to normalized or denormalized numbers or zero before being used. Thus, as required by the IEEE standard, the FPCP does not distinguish between redundant encodings of extended precision values.

3.2.3 Zeros

Zeros are signed (positive or negative) and represent the real values +0.0 and −0.0. See Figure 3-5.

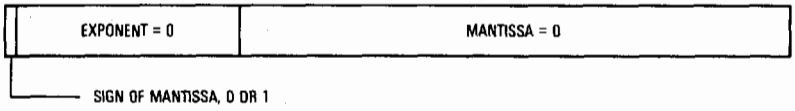
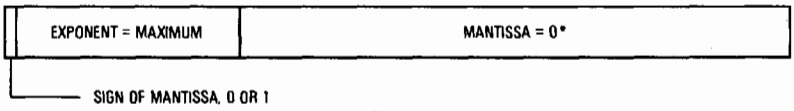


Figure 3-5. Format of Zero

3.2.4 Infinities

Infinities are signed (positive or negative) and represent real values that exceed the overflow threshold. Overflow is detected for a given data format and operation when the result exponent is greater than or equal to the maximum exponent value. (This overflow description ignores the effects of rounding and the user selectable rounding modes.) See Figure 3-6. For extended precision infinities, the most significant bit of the mantissa (the integer bit) can be either one or zero.



*For the extended precision format, the most significant bit of the mantissa (the integer bit) is a don't care.

Figure 3-6. Format of Infinity

3.2.5 Not-A-Numbers

When created by the FPCP, not-a-numbers (NANs) represent the results of operations that have no mathematical interpretation, such as infinity divided by infinity. All operations involving a NAN operand as an input return a NAN result. When created by the user, NANs can protect against uninitialized variables and arrays, or represent user-defined special

number types. See Figure 3-7. For extended precision NANs, the most significant bit of the mantissa (the integer bit) can be either one or zero.

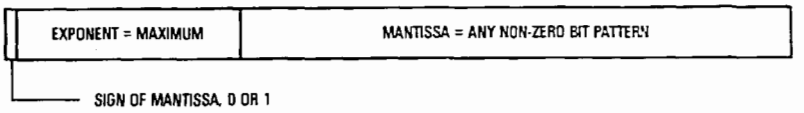


Figure 3-7. Format of Not-A-Numbers

Two different types of NANs are implemented by the FPCP. The value of the most significant bit (MSB) of the fraction identifies the type. The identifying bit is the MSB of the mantissa for single and double precision and the MSB of the mantissa minus one for extended precision. NANs with a leading fraction bit equal to one are nonsignaling NANs; NANs with a leading fraction bit equal to zero are signaling NANs (SNANs). A SNAN can be used as an escape mechanism for a user-defined non-IEEE data type. The FPCP never creates a SNAN as a result of an operation.

The IEEE specification defines the manner in which a NAN is processed when used as an input to an operation. Particularly, if a SNAN is used as an input and the SNAN trap is not enabled, a nonsignaling NAN must be returned as the result. The FPCP accomplishes this by using the source SNAN, setting the most significant bit of the fraction, and storing the resultant nonsignaling NAN in the destination. Due to the IEEE formats for NANs, the result of setting the most significant fraction bit of a SNAN is always a nonsignaling NAN.

When NANs are created by the FPCP, the NANs always contain the same bit pattern in the mantissa; for any precision, all bits of the mantissa are ones. When a NAN is created by the user, any nonzero bit pattern can be stored in the mantissa.

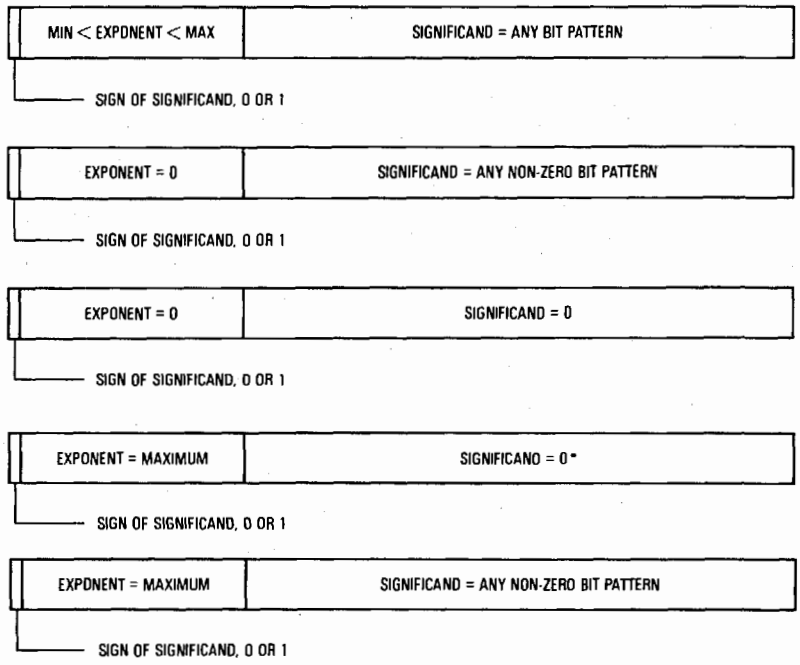
3.2.6 Binary Real Data Summary

Figure 3-8 presents a summary, for quick reference, of the five floating-point data types for the single, double, and extended precision formats.

3.3 PACKED DECIMAL REAL DATA FORMAT

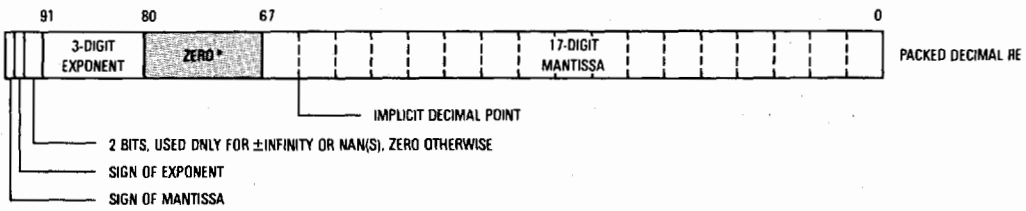
The packed decimal floating-point data format consists of a 24 digit packed decimal string as shown in Figure 3-9. A decimal floating-point source operand is converted to an extended precision value before the specified operation is performed. Thus, mixed mode arithmetic is implicitly supported.

The packed decimal representation for the special data types of zero, infinity, and NAN is described in **3.6 DATA FORMAT DETAILS**, which also defines all possible data patterns in the packed decimal data format.



*For the extended precision format, the most significant bit of the significand (the integer bit) is a don't care.

Figure 3-8. Binary Real Data Type Summary



*Unless a binary-to-decimal conversion overflow occurs

Figure 3-9. Packed Decimal Real Data Format

3.4 INTERNAL DATA FORMAT

All FPCP internal operations are performed in extended precision. All external operands, regardless of data format, are converted to extended precision values before the specified operation is performed.

The format used in the eight floating-point data registers is identical to the extended precision data format described previously and in **3.6 DATA FORMAT DETAILS** (with the

deletion of the 16 unused bits). The extended precision data format has a 15-bit biased integer exponent and a 64-bit mantissa.

The format of an intermediate result is shown in Figure 3-10. The intermediate result exponent for some dyadic operations (multiply and divide) can easily overflow or underflow the 15-bit exponent. In order to simplify overflow and underflow detection, intermediate results in the FPCP maintain a 17-bit twos-complement integer exponent. When an overflow or underflow intermediate result is detected, the intermediate 17-bit exponent is always converted into a 15-bit biased exponent before it is stored in a floating-point data register. Additionally, the mantissa is maintained internally as 67 bits for rounding purposes, but is always rounded to 64 bits (or less, depending on the selected rounding precision) before it is stored in a floating-point data register.

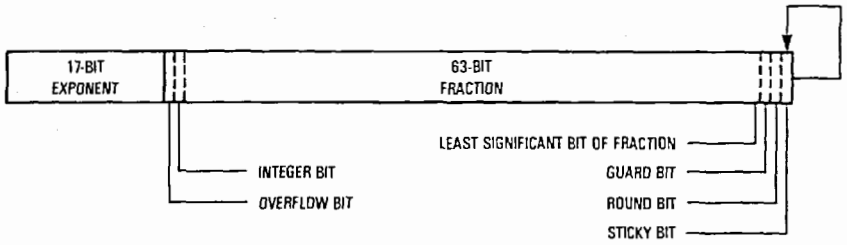


Figure 3-10. Intermediate Result Format

3.5 FORMAT CONVERSIONS

Two cases of conversion between two data formats are:

- Converting an operand in any memory data format to the extended precision data format and storing it in a floating-point data register, or using it as the source operand for an arithmetic operation.
- Converting the extended precision value in a floating-point data register to any data format and storing it in a memory destination.

3.5.1 Conversion to Extended Precision Data Format

Since the internal data format used by the FPCP is always extended precision, all external operands, regardless of data format, are converted to extended precision values before the specified operation is performed. If the external operand, regardless of data format, is a denormalized number, the number is normalized before the specified operation is performed. Conversion and normalization apply not only to loading a floating-point data register but also to external operands involved in arithmetic operations.

Since floating-point data registers always contain extended precision data format values, an external extended precision denormalized number moved into a floating-point data register is stored as an extended precision denormalized number. In this case, the number

is first normalized and then denormalized before it is stored in the designated floating-point data register. This method simplifies the handling of all other data formats and types.

If an external operand is an extended precision unnormalized number, the number is normalized before it is used in an arithmetic operation. If the external operand is an extended precision unnormalized zero (i.e., with a mantissa of all zeros), the number is converted to an extended precision normalized zero before the specified operation is performed. This normalization and conversion applies not only to external unnormalized operands involved in arithmetic operations, but also applies to loading a floating-point data register. Note that the regular use of unnormalized inputs defeats the purpose of the IEEE standard and may produce gross inaccuracy in the results.

3.5.2 Conversions to Other Data Formats

Conversion from the extended precision data format to any of the other six data formats occurs when the contents of an FPCP floating-point data register are stored to memory or an MPU data register. Since no operation performed by the FPCP can create an unnormalized result, the result of moving a floating-point data register to an extended precision external destination can never be an unnormalized number.

3.6 DATA FORMAT DETAILS

The following paragraphs provide the format specification details for the single (S), double (D), extended (X) precision binary real, and packed decimal (P) real string data formats. Refer to Tables 3-1 through 3-4 and Figure 3-11.

Table 3-1. Single Precision Binary Real Format

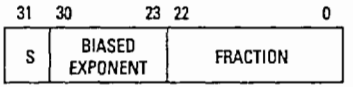
Memory Format:	
Field Size (in Bits):	
s = Sign	1
e = Biased Exponent	8
f = Fraction	23
Total	32
Interpretation of Sign:	
Positive Mantissa, s =	0
Negative Mantissa, s =	1
Normalized Numbers:	
Bias of e	+127 ($7F)
Range of e	0 < e < 255 ($FF)
Range of f	Zero or Nonzero
Mantissa = Significand =	1.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{e-127} \times 1.f$
Denormalized Numbers:	
e = Format Minimum =	0 ($00)
Bias of e	+126 ($7E)
Range of f	Nonzero
Mantissa = Significand =	0.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{-126} \times 0.f$
Signed Zeros:	
e = Format Minimum =	0 ($00)
f = Mantissa = Significand =	0.f = 0.0
Signed Infinities:	
e = Format Maximum =	255 ($FF)
f = Mantissa = Significand =	0.f = 0.0
NANs (Not-A-Number):	
s =	Don't Care
e = Format Maximum =	255 ($FF)
f =	Non-Zero
Representation of f	0.1xxxx . . . xxxx, Nonsignaling
	0.0xxxx . . . xxxx, Signaling
xxxx . . . xxxx	Nonzero Bit Pattern
f When Created by the FPCP	.11111 . . . 1111
Ranges (Approximate):	
Maximum Positive Normalized	3.4×10^{38}
Minimum Positive Normalized	$1.2 \times 10^{s-38}$
Minimum Positive Denormalized	1.4×10^{-45}

Table 3-2. Double Precision Binary Real Format

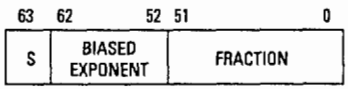
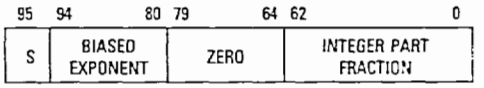
Memory Format:	
Field Size (in Bits):	
s = Sign	1
e = Biased Exponent	11
f = Fraction	52
Total	64
Interpretation of Sign:	
Positive Mantissa, s =	0
Negative Mantissa, s =	1
Normalized Numbers:	
Bias of e	+1023
Range of e	0 < e < 2047 ($7FF)
Range of f	Zero or Nonzero
Mantissa = Significand =	1.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{e-1023} \times 1.f$
Denormalized Numbers:	
e = Format Minimum =	0 ($000)
Bias of e	+1022 ($3FE)
Range of f	Nonzero
Mantissa = Significand =	0.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{-1022} \times 0.f$
Signed Zeros:	
e = Format Minimum =	0 ($00)
f = Mantissa = Significand =	0.f = 0.0
Signed Infinities:	
e = Format Maximum =	2047 ($7FF)
f = Mantissa = Significand =	0.f = 0.0
NANs (Not-A-Number):	
s =	Don't Care
e = Format Maximum =	2047 ($7FF)
f =	Nonzero
Representation of f	0.1xxxx . . . xxxx, Nonsignaling
	0.0xxxx . . . xxxx, Signaling
xxxx . . . xxxx	Nonzero Bit Pattern
f When Created by the FPCP	.11111 . . . 1111
Ranges (Approximate):	
Maximum Positive Normalized	18×10^{307}
Minimum Positive Normalized	2.2×10^{-308}
Minimum Positive Denormalized	4.9×10^{-324}

Table 3-3. Extended Precision Binary Real Format

Memory Format:



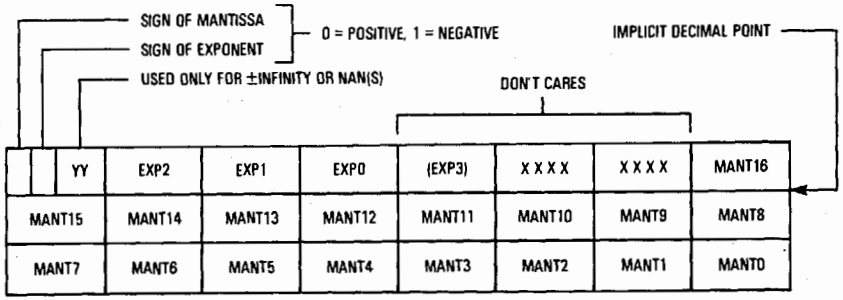
Field Size (in Bits):	
s = Sign	1
e = Biased Exponent	15
u = Zero, Reserved	16
j = Integer Part	1
f = Fraction	63
Total	96
Interpretation of Unused Bits:	
Input	Don't Care
Output	All Zeros
Interpretation of Sign:	
Positive Mantissa, s =	0
Negative Mantissa, s =	1
Normalized Numbers:	
Bias of e	+16383 ($3FFF)
Range of e	0 <= e < 32767 ($7FFF)
j =	1
Range of f	Zero or Nonzero
j.f = Mantissa = Significand =	1.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{e-16383} \times j.f$
Denormalized Numbers:	
e = Format Minimum	0 ($0000)
Bias of e	+16383 ($3FFF)
j =	0
Range of f	Nonzero
j.f = Mantissa = Significand =	0.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{-16383} \times 0.f$
Signed Zeros:	
e = Format Minimum =	0 ($0000)
j.f = Mantissa = Significand =	0.0
Signed Infinities:	
e = Format Maximum =	32767 ($7FFF)
j =	Don't Care
j.f = Mantissa = Significand	j.000 . . . 0000
NANs (Not-A-Numbers):	
s =	Don't Care
j =	Don't Care
e = Format Maximum =	32767 ($7FFF)
f =	Nonzero
Representation of f	j.1xxx . . . xxxx, Non-Signaling
	j.0xxx . . . xxxx, Signaling
xxx . . . xxxx	Non-Zero Bit Pattern
f When Created by the FPCP	1.11111 . . . 1111
Ranges (Approximate):	
Maximum Positive Normalized	6×10^{4931}
Minimum Positive Normalized	8×10^{-4933}
Minimum Positive Denormalized	9×10^{4952}

Table 3-4. Decimal String Type Definitions

Operand Type	Word 5					Word 4	Words 3–0
	15	14	13	12	11 . . . 0	15 . . . 0	
	SM	SE	y	y	3-Digit Exponent	1-Digit Integer	16-Digit Fraction
±INFINITY	0/1	1	1	1	$FFF	$xxx	$00 . . . 00
±NAN	0/1	1	1	1	$FFF	$xxxx	Nonzero (see Note 1)
±SNAN	0/1	1	1	1	$FFF	$xxxx	Nonzero (see Note 1)
±ZERO	0	0/1	x	x	$000–$999	$xxx0	$00 . . . 00
−ZERO	1	0/1	x	x	$000–$999	$xxx0	$00 . . . 00
+In-Range	0	0/1	x	x	$000–$999	$xxx0–$xxx9	$00 . . . 01–$99 . . . $99
−In-Range	1	0/1	x	x	$000–$999	$xxx0–$xxx9	$00 . . . 01–$99 . . . $99

NOTES:

1. A decimal string with the SE and y bits set, an exponent of $FFF, and a nonzero 16-digit decimal fraction is a NAN. When this string is used by the FPCP, the fraction part of the NAN is moved bit-for-bit into the extended precision mantissa of a floating-point register. The exponent of the register is set to signify a NAN, but no decimal-to-binary conversion or any other conversion is performed. Therefore, the most significant bit of the most significant digit in the decimal fraction (most significant bit of MANT15) is a don't care (as in extended NANs), and the most significant bit minus one of MANT15 is the signaling NAN (SNAN) bit. If the NAN bit is a zero, then it is a SNAN.
2. If a nondecimal digit [$A . . . $F] appears in the exponent of a zero, the number is converted to a true zero. The FPCP does not detect nondecimal digits [$A . . . $F] in the exponent, integer, or fraction digits of an in-range decimal string. These nondecimal digits are converted to binary in the same manner as decimal digits; however, the result is probably useless although it is repeatable.
3. Since in-range numbers cannot overflow or underflow when converted to extended precision, normalized extended precision numbers are always produced by conversion from the decimal data format.



MANTn Is the nth digit of the mantissa.
EXPn Is the nth digit of the exponent. EXP3 is only generated during a move out operation if the source operand exponent exceeds the magnitude of a three digit exponent; otherwise, it is a don't care. Only EXP0-EXP2 are used for input.
XXXX Are don't care bits, which are zero when written and ignored when read.

Figure 3-11. Packed Decimal Real Data Format Detail

SECTION 4
INSTRUCTION SET

This section describes the MC68881/MC68882 (FPCP) instruction set in detail, using the Freescale assembly language syntax and notation. As an introduction, a summary of the instruction set is presented, followed by a detailed description of each instruction. Also, included at the end of this section is a listing of the binary pattens of all of the instructions and an opcode map summary for use by assembler and compiler writers.

4.1 INSTRUCTION DESCRIPTION CONVENTIONS

The instruction set is discussed in this section using functional grouping and the following notation:

B, W, L	The same size codes as all M68000 Family processors; specifies a signed integer data type (twos complement) of byte (8 bits), word (16 bits), or long word (32 bits)
S	Single precision real data format (32 bits)
D	Double precision real data format (64 bits)
X	Extended precision real data format (96 bits, 16 bits unused)
P	Packed BCD real data format (96 bits, 12 bytes)
FPm, FPn	One of eight floating-point data registers
FPcr	One of the three floating-point system control registers (FPCR, FPSR, or FPIAR)
<ea>	Any valid MC68020/MC68030 (MPU) address mode
k	A twos-complement signed integer (−64 to +17) that specifies the format of a number to be stored in the packed decimal format
ccc	An index into the FPCP constant ROM
<list>	A list of floating-point data registers or control registers
<label>	A relative label used by an assembler to calculate a displacement

4.2 INSTRUCTION GROUPS

The following paragraphs briefly describe each instruction group along with tables showing the Freescale syntax for each instruction. The FPCP instructions can be separated into the following groups:

Data Movement
Dyadic Operations
Monadic Operations
Program Control
System Control

4.2.1 Data Movement Operations

This group of instructions includes those that load or store the user-visible configuration of the FPCP and that move operands into, between, or out of the floating-point data registers. Data format conversion functions are also implicitly supported since all external data formats are converted to extended precision for internal storage, and vice versa. Operations to move the system control registers into and out of the FPCP are also provided. The move constant ROM (FMOVECR) instruction allows floating-point data registers to be loaded quickly with commonly used constants such as π, e, 0.0, 1.0, etc. Table 4-1 summarizes the data movement instructions that are available and the operand data formats supported.

Table 4-1. Data Movement Operations

Instruction	Operand Syntax	Operand Format	Operation
FMOVE	FPm,FPn ⟨ea⟩,FPn FPm,⟨ea⟩ FPm,⟨ea⟩{#k} FPm,⟨ea⟩{Dn} ⟨ea⟩,FPcr FPcr,⟨ea⟩	X B,W,L,S,D,X,P B,W,L,S,D,X P P L L	source ⧫ destination
FMOVECR	#ccc,FPn	X	ROM constant ⧫ FPn
FMOVEM	⟨ea⟩,⟨list⟩[1] ⟨ea⟩,Dn ⟨list⟩[1],⟨ea⟩ Dn,⟨ea⟩	L,X X L,X X	Listed register ⧫ destination source ⧫ listed registers

NOTE:
1. The register list may include any combination of the eight floating-point data registers, or it may contain any combination of the three control register FPCR, FPSR, and FPIAR. If the register list mask resides in a main processor data register, only floating-point data registers may be specified.

4.2.2 Dyadic Operations

The dyadic floating-point instructions provide several arithmetic functions that require two input operands such as add, subtract, multiply, and divide. For these operations, the first operand may be located in memory, in an integer data register, or in a floating-point data register, and the second operand is always contained in a floating-point data register. The results of the operation are stored in the register specified as the second operand. With two exceptions, all operations support any data format and are performed to extended precision. The exceptions are the single precision multiply and divide instructions (FSGLMUL and FSGLDIV). These instructions support any precision inputs, but return results accurate only to single precision. These instructions provide very high-speed operations by sacrificing accuracy. The general format of the dyadic instructions is given in Table 4-2; the available operations are listed in Table 4-3.

Table 4-2. Dyadic Operation Format

Instruction	Operand Syntax	Operand Format	Operation
F⟨dop⟩	⟨ea⟩,FPn FPm,FPn	B,W,L,S,D,X,P X	FPn ⟨function⟩ source ⧫ FPn

where:
<dop> is any one of the dyadic operation specifiers.

Table 4-3. Dyadic Operations

Instruction	Function
FADD	Add
FCMP	Compare
FDIV	Divide
FMOD	Modulo Remainder
FMUL	Multiply
FREM	IEEE Remainder
FSCALE	Scale Exponent
FSGLDIV	Single Precision Divide
FSGLMUL	Single Precision Multiply
FSUB	Subtract

4.2.3 Monadic Operations

The monadic floating-point instructions provide several arithmetic functions that require only one input operand. Unlike the integer counterparts to these function (e.g., NEG <ea>), a source and a destination may be specified. The operation is performed on the source operand, and the result is stored in the destination, which is always a floating-point data register. When the source is not a floating-point data register, all data formats are supported; the data format is always extended precision for register-to-register operations. The general format of these instructions is shown in Table 4-4, and the available operations are listed in Table 4-5. The form of the simultaneous sine and cosine instruction is given in Table 4-6.

Table 4-4. Monadic Operation Format

Instruction	Operand Syntax	Operand Format	Operation
F(mop)	(ea),FPn	B,W,L,S,D,X,P	source ➧ function ➧ FPn
	FPm,FPn	X	
	FPn	X	FPn ➧ function ➧ FPn

where:
<mop> is any one of the monadic operation specifiers.

Table 4-5. Monadic Operations

Instruction	Function
FABS	Absolute Value
FACOS	Arc Cosine
FASIN	Arc Sine
FATAN	Arc Tangent
FATANH	Hyperbolic Arc Tangent
FCOS	Cosine
FCOSH	Hyperbolic Cosine
FETOX	e^x
FETOXM1	$e^x - 1$
FGETEXP	Extract Exponent
FGETMAN	Extract Mantissa
FINT	Extract Integer Part
FINTRZ	Extract Integer Part, Rounded-to-Zero

Instruction	Function
FLOGN	ln(x)
FLOGNP1	ln(x+1)
FLOG10	$\log_{10}(x)$
FLOG2	$\log_2(x)$
FNEG	Negate
FSIN	Sine
FSINH	Hyperbolic Sine
FSQRT	Square Root
FTAN	Tangent
FTANH	Hyperbolic Tangent
FTENTOX	10^x
FTWOTOX	2^x

Table 4-6. Dual Monadic Operation Format

Instruction	Operand Syntax	Operand Format	Operation
FSINCOS	⟨ea⟩,FPc:FPs FPm,PFc:FPs	B,W,L,S,D,X,P X	SIN(source) ⇨ FPs; COS(source) ⇨ FPc

4.2.4 Program Control Operations

The program control instructions provide a means of affecting program flow based on conditions present in the floating-point status register after any operation that sets the condition codes. In addition to allowing direct control of program flow with the branch conditionally (FPcc) and the decrement and branch conditionally (FDBcc) instructions, the set conditionally (FScc) instruction allows the user to set a Boolean variable based on the floating-point condition codes as an intermediate result in the evaluation of a complex Boolean equation. Also included is a test operand instruction (FTST) that sets the floating-point condition codes for use by the other program and system control instructions, and a no operation instruction (FNOP) that may be used to force synchronization of the FPCP with the main processor. Table 4-7 summarizes the program control instructions that are available.

Table 4-7. Program Control Operations

Instruction	Operand Syntax	Operand Format	Operation
FBcc	⟨label⟩	W,L	*If Condition True,* Then PC+d ⇨ PC
FDBcc	Dn,⟨label⟩	W	If Condition True, Then No Operation; Else Dn−1 ⇨ Dn; If Dn≠ −1 The PC+d ⇨ PC
FNOP	None	None	No Operation
FScc	⟨ea⟩	B	If Condition True, The 1's ⇨ Destination Else 0's ⇨ Destination
FTST	⟨ea⟩ FPn	B,W,L,S,D,X,P X	Set FPSR Condition Codes

The FPCP supports 32 conditional tests that are separated into two groups — 16 that cause an exception if an unordered condition is present when the conditional test is attempted, and 16 that do not cause an exception if an unordered condition is present. (An unordered condition occurs when an input to an arithmetic operation is a NAN.) Table 4-8 lists the 32 condition code mnemonics along with the conditional test function. Refer to **4.4 CONDITIONAL TEST DEFINITIONS** for a detailed description of the conditional equation used by each test.

Table 4-8. Conditional Test Mnemonics

Exception on Unordered		No Exception on Unordered	
GE	Greater Than or Equal	OGE	Ordered Greater Than or Equal
GL	Greater Than or Less Than	OGL	Ordered Greater Than or Less Than
GLE	Greater Than or Less	OR	Ordered
GT	Greater Than	OGT	Ordered Greater Than

Table 4-8. Conditional Test Mnemonics (Continued)

Exception on Unordered		No Exception on Unordered	
LE	Less Than or Equal	OLE	Ordered Less Than or Equal
LT	Less Than	OLT	Ordered Less Than
NGE	Not (Greater Than or Equal)	UGE	Unordered or Greater Than Equal
NGL	Not (Greater Than or Less Than)	UEQ	Unordered or Equal
NGLE	Not (Greater Than or Less Than or Equal)	UN	Unordered
NGT	Not Greater Than	UGT	Unordered or Greater Than
NLE	Not (Less Than or Equal)	ULE	Unordered or Less Than or Equal
NLT	Not Less Than	ULT	Unordered or Less Than
SEQ	Signaling Equal	EQ	Equal
SNE	Signaling Not Equal	NE	Not Equal
SF	Signaling Always False	F	Always False
ST	Signaling Always True	T	Always True

4.2.5 System Control Operations

The system control instructions communicate with the operating system via a conditional trap instruction (FTRAPcc) and save or restore (FSAVE or FRESTORE) the nonuser visible portion of the FPCP during context switches in a virtual memory or other type of multitasking system. The conditional trap instruction uses the same conditional tests as the program control instructions and allows an optional 16- or 32-bit immediate operand to be included as part of the instruction for passing parameters to the operating system. Table 4-9 summarizes the system control instructions.

Table 4-9. System Control Operations

Instruction	Operand Syntax	Operand Size	Operation
FRESTORE	⟨ea⟩	None	State Frame ⇞ Internal Registers
FSAVE	⟨ea⟩	None	Internal Registers ⇟ State Frame
FTRAPcc	None #xxx	None W,L	If Condition True, Then Take Exception

4.3 COMPUTATIONAL ACCURACY

Whenever an attempt is made to represent a real number in a binary format of finite precision, there is a possibility that the number cannot be represented exactly; this is commonly referred to as round-off error. Furthermore, when two inexact numbers are used in a calculation, the error present in each number is reflected and possibly aggravated in the result.

One of the major reasons that the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985) was developed is to define the error bounds for calculation of binary

floating-point values so that all machines conforming to the standard produce the same results for an operation. The operation must meet the following conditions:

1. Same input values,
2. Same rounding mode, and
3. Same precision.

The IEEE standard specifies not only the format of data items, but also defines:

1. The maximum allowable error that may be introduced during a calculation, and
2. The manner in which rounding of the result is performed.

However, the IEEE specification defines only the operation of some of the instructions supported by the FPCP; those not specified by the IEEE standard are described in detail in the following paragraphs. The following paragraphs discuss the accuracy of the calculations performed by the FPCP, grouping them as follows:

1. The IEEE specified operations and nontranscendental functions,
2. The transcendental functions, and
3. The IEEE specified conversions between binary and decimal real formats.

4.3.1 Arithmetic Instructions

The *IEEE Specification for Binary Floating-Point Arithmetic* specifies that the following operations must be supported for each data format: add, subtract, multiply, divide, remainder, square root, integer part, and compare. Conversions between the various data formats are also required. In addition to these arithmetic functions, the FPCP also supports the nontranscendental operations of: absolute value, get exponent, get mantissa, negate, modulo remainder, scale exponent, and test. Since the IEEE specification defines the error bounds to which all calculations are performed, the result obtained by any conforming machine can be predicted exactly for a particular precision and rounding mode. The error bound defined by the IEEE specification is one-half unit in the last place of the destination data format in the round-to-nearest mode and one unit in the last place in the other rounding modes.

The FPCP performs all calculations using a 67-bit mantissa for the intermediate results. The three bits beyond the precision of the extended format allow the FPCP to perform all calculations as if to infinite precision and then round the result to the desired precision before storing it in the destination. By performing calculations in this manner, the final result is always correct for the specified destination data format before rounding is performed (unless an overflow or underflow error occurs). The specified rounding operation then produces a number that is as close as possible to the infinitely precise intermediate value and is still representable in the selected precision. An example of how the 67-bit mantissa allows the FPCP to meet the error bound of the IEEE specification is as follows:

	Mantissa	1	g	r	s	
Intermediate Result:	x.x.....x00		1	0	0	(Tie Case)
Round-to-Nearest Result:	x.x.....x00					

In this case, the least significant bit (1) of the rounded result is not incremented, even though the guard bit (g) is set in the intermediate result. The IEEE standard specifies that tie cases should be handled in this manner. Assuming that the destination data format is

extended, if the difference between the infinitely precise intermediate result and the round-to-nearest result is calculated, the relative difference is 2^{-64} (the value of the guard bit). This error is equal to one-half of the value of the least significant bit and is the worst-case error that can be introduced when using the round-to-nearest mode. Thus, the term one-half unit in the last place correctly identifies the error bound for this operation. This error specification is the relative error present in the result; the absolute error bound is equal to $2^{exponent} \times 2^{-64}$. An example of the error bound for the other rounding modes is as follows:

	Mantissa	1	g	r	s
Intermediate Result:	x.x.....x00		1	1	1
Round-to-Zero Result:	x.x.....x00				

In this case, the difference between the infinitely precise result and the rounded result is the error bound for this operation is not more than one unit in the last place. For all of the the error bound for this operation is not more than one unit in the last place. For all of the arithmetic operations, these error bounds are met by the FPCP, thus providing accurate and repeatable results.

4.3.2 Transcendental Instructions

The IEEE specification does not define the error bound to which transcendental (except square root) functions are to be performed. In this context, the transcendental functions are all of those operations not mentioned in the previous paragraphs (i.e., the trigonometric, hyperbolic, logarithmic, and exponential instructions). Due to the highly recursive nature of the algorithms used to calculate these functions, the round-off error in the input operands to a function, combined with the limited precision of the FPCP ALU, do not allow the calculation of a result with the same error limit as the arithmetic functions. However, these operations are quite accurate given the constraint of using an ALU with a finite precision of 67 bits. In general, the worst-case accuracy of any transcendental function is one unit in the last place of double precision (which is equal to 4096 units in the last place of extended precision). The typical error bound for these instructions is approximately 64 units in the last place of extended precision. The following example illustrates the significance of this error bound:

	Mantissa
Correct Result:	x.x.....x00000000
FPCP Calculated Result:	x.x.....x01000000

In this case, the relative difference between the correct result and the result calculated by the FPCP is 2^{-57} (assuming an extended precision result), which is 26 times the value of the least significant bit. This difference corresponds to an error of 64 units in the last place.

Note that the transcendental functions perform limited checking for special case input values such as boundary conditions. For example, the exponential functions check for a zero input value, but do not check for exact integer values. Thus, raising a number to an exact integer value may not produce an exact result (e.g., the instruction FTENTOX #1,FP0 does not produce an extended precision value of exactly 10.0), and the INEX2 bit in the FPSR may be set even if an exact result is produced.

4.3.3 Decimal Conversions

The IEEE standard does not specify the format of the decimal real representation used by any conforming machine, but it does define the error bounds for conversions between decimal and the single- and double-precision binary formats. Thus, such conversions always produce consistently rounded results, and those results are predictable and repeatable on any conforming system. However, it is not always possible to perform an exact conversion between these data formats, due to the limited precision of the numbers and the different radices of the values. The error bound for these conversions is 0.97 unit in the last digit of the destination precision for the round-to-nearest mode; and 1.47 units in the last digit of the destination precision for the other rounding modes. When an input conversion cannot produce an exact result, the FPCP sets the INEX1 bit in the FPSR exception byte. This indication allows for special handling of these conversion errors that is separate from the handling of other types of inaccurate results. When an output conversion cannot produce an exact result, the INEX2 bit is set.

The packed decimal data format supported by the FPCP allows the representation of double-precision binary number in a decimal form, in accordance with the IEEE specification. When a packed decimal number is converted to extended precision, the result is always in range although the conversion may be inexact. The result is within range because the magnitudes of the exponent and mantissa of a packed decimal number are less than the largest values representable in the extended precision format. Refer to **6.1.8 Inexact Result on Decimal Input** for a description of the handling of inaccurate decimal-to-binary conversions.

When an extended precision number is converted to packed decimal, the result may be a number that cannot be represented exactly, or a number that is too large to be represented with a three-digit exponent. When this type of conversion is performed, the k factor specified is used to locate the decimal rounding boundary. If the magnitude of the rounded decimal result exponent exceeds 999, the FPCP signals an operand error and calculates a fourth exponent digit, which is included in the destination operand (see Figure 3-11 for the position of the fourth digit). Refer to **6.1.7 Inexact Result** for a description of the handling of inaccurate binary-to-decimal conversions.

Note that the error bounds specified by the IEEE standard apply only to conversions of values in the range of the double-precision format. The error bound for conversions by the FPCP of extended precision values which cannot be represented in double precision is significantly larger. Software must be provided to convert such extended precision values to decimal. The conversion must generate decimal results with an error bound analogous to those specified in the IEEE standard for double-precision values. The software envelope must utilize a super extended precision to achieve such error bounds.

Note that the binary to/from decimal conversions performed by the FPCP utilize the on-chip ROM values of powers of 10 for speed and accuracy, thus allowing exact conversions in many cases (particularly for values that are exact powers of 10).

4.4 CONDITIONAL TEST DEFINITIONS

The FPCP provides a very simple mechanism for performing conditional tests of the result of any arithmetic floating-point operation. First, the condition code bits in the FPSR are set or cleared at the end of any arithmetic operation or move operation to a single floating-point data register. The condition code bits are always set consistently based on the result

of the operation. Second, the FPCP provides 32 conditional tests that are supported in hardware by the M68000 Family coprocessor interface. This mechanism allows conditional instructions that test floating-point conditions to be coded in exactly the same way as the integer conditional instructions. The evaluation of the conditional test by the FPCP is performd automatically. The combination of the consistent setting of the condition code bits and the simple programming of conditional instructions gives the MC68020/MC68030 and FPCP combination a very flexible, high-performance method of altering program flow based on floating-point results.

One important programming consideration is that the inclusion of the NAN data type in the IEEE floating-point number system requires each conditional test to include the NAN condition code bit in its Boolean equation. Because a comparison of a NAN with anything is unodered (i.e., it is impossible to determine if a NAN is bigger or smaller than an in-range number), the compare instruction sets the NAN condition code bit when an unordered compare is attempted. All arithmetic instructions also set the NAN bit if the result of an operation is a NAN. The conditional instructions interpret the NAN condition code bit equal to one as the unordered condition.

The inclusion of the unordered condition in floating-point branches destroys the familiar trichotomy relationship (greater than, equal, less than) that exists for integers. For example, the opposite of floating-point branch greater than (FBGT) is not floating-point branch less than or equal (FBLE). Rather, the opposite condition is floating-point branch not greater than (FBNGT). If the result of the previous instruction was unordered, FBNGT is true; whereas, both FBGT would be false since unordered fails both of these tests (and sets BSUN). Compiler programmers should be particularly careful of the lack of trichotomy in the floating-point branches since it is common for compilers to invert the sense of conditions.

In the following paragraphs, the conditional tests are described in three main categories:

1. IEEE nonaware tests,
2. IEEE aware test, and
3. Miscellaneous.

The set of IEEE nonaware tests is best used:

1. When porting a program from a system that does not support the IEEE standard to a conforming system, or
2. When generating high-level language code that does not support IEEE floating-point concepts (i.e., the unordered condition).

When using the set of IEEE nonaware tests, the user receives a BSUN exception whenever a branch is attempted and the NAN condition code bit is set, unless the branch is an FBEQ or an FBNE. If the BSUN trap is enabled in the FPCR register, the exception causes a trap. Therefore, the IEEE nonaware program is interrupted if something unexpected occurs.

The IEEE aware branch set should be used in programs that contain ordered and unordered conditions by compilers and programmers who are knowledgeable of the IEEE standard. Since the ordered or unordered attribute is explicitly included in the conditional test, the BSUN bit is not set in the status register EXC byte when the unordered condition occurs.

4.4.1 IEEE Nonaware Tests

All of the conditional tests in the following table, except EQ and NE, set the BSUN bit in the status register exception byte if the NAN condition code bit is set when a conditional instruction is executed.

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not Equal	$\overline{Z}$	001110
GT	Greater Than	$\overline{NAN \vee Z \vee N}$	010010
NGT	Not Greater Than	NANvZvN	011101
GE	Greater Than or Equal	Zv($\overline{NAN}$vN)	010011
NGE	Not (Greater Than or Equal)	NANv(N$\wedge\overline{Z}$)	011100
LT	Less Than	N$\wedge$($\overline{NAN \vee Z}$)	010100
NLT	Not Less Than	NANv(Zv$\overline{N}$)	011011
LE	Less Than or Equal	Zv(N$\wedge\overline{NAN}$)	010101
NLE	Not (Less Than or Equal)	NANv($\overline{N \vee Z}$)	011010
GL	Greater or Less Than	$\overline{NAN \vee Z}$	010110
NGL	Not (Greater or Less Than)	NANvZ	011001
GLE	Greater, Less or Equal	$\overline{NAN}$	010111
NGLE	Not (Greater, Less or Equal)	NAN	011000

where:
"v" = Logical OR
"$\wedge$" = Logical AND

4.4.2 IEEE Aware Tests

The following conditional tests do not set the BSUN bit in the status register exception byte under any circumstances.

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not Equal	$\overline{Z}$	001110
OGT	Ordered Greater Than	$\overline{NAN} \vee Z \vee \overline{N}$	000010
ULE	Unordered or Less or Equal	NAN∨Z∨N	001101
OGE	Ordered Greater Than or Equal	Z∨($\overline{NAN} \vee \overline{N}$)	000011
ULT	Unordered or Less Than	NAN∨(N∧$\overline{Z}$)	001101
OLT	Ordered Less Than	N∧($\overline{NAN} \vee \overline{Z}$)	000100
UGE	Unordered or Greater or Equal	NAN∨Z∨N	001011
OLE	Ordered Less Than or Equal	Z∨(N∧$\overline{NAN}$)	000101
UGT	Unordered or Greter Than	NAN∨($\overline{N} \vee \overline{Z}$)	001010
OGL	Ordered Greater or Less Than	$\overline{NAN} \vee \overline{Z}$	000110
UEQ	Unordered or Equal	NAN∨Z	001001
OR	Ordered	$\overline{NAN}$	000111
UN	Unordered	NAN	001000

where:
"∨" = Logical OR
"∧" = Logical AND

4.4.3 Miscellaneous Tests

The following tests are not generally used but are implemented for completeness of the set. If the NAN condition code bit is set, T and F do not set the BSUN bit, but SF, ST, SEQ, and SNE do set the BSUN bit.

Mnemonic	Definition	Equation	Predicate
F	False	False	000000
T	True	True	001111
SF	Signaling False	False	010000
ST	Signaling True	True	011111
SEQ	Signaling Equal	Z	010001
SNE	Signaling Not Equal	$\overline{Z}$	011110

4.5 DETAILED INSTRUCTION DESCRIPTIONS

Subsequent paragraphs contain detailed information about each instruction in the FPCP instruction set. Instructions are arranged in alphabetical order by assembler mnemonic. The following paragraphs provide background information to aid in reading the detailed instruction information presented.

4.5.1 Addressing Modes

Due to the nature of the MC68020/MC68030 and FPCP coprocessor interface, the FPCP supports all MC68020/MC68030 addressing modes. The MC68020/MC68030 effective address modes are categorized by the manner in which the modes are used. The following classifications are used in the instruction details.

Data — If an effective address is used to refer to data operands, it is considered a data addressing mode.

Memory — If an effective address is used to refer to memory operands, it is considered a memory addressing mode.

Alterable — If an effective address is used to refer to alterable (writable) operands, it is considered an alterable addressing mode.

Control — If an effective address is used to refer to memory operands that do not have an associated size, it is considered a control addressing mode.

Table 4-10 shows the various addressing categories of each effective address mode. These categories may be combined so that additional, more restrictive, classifications may be defined. For example, the instruction descriptions use such classifications as memory alterable or data alterable. The former refers to those addressing modes which are both memory and alterable addresses (i.e., the intersection of the two sets of modes), and the latter refers to addressing modes which are both data and alterable.

Table 4-10. Effective Addressing Mode Categories

Address Modes	Mode	Register	Data	Memory	Control	Alterable	Assembler Syntax
Data Register Direct	000	reg. no.	X	—	—	X	Dn
Address Register Direct	001	reg. no.	—	—	—	X	An
Address Register Indirect	010	reg. no.	X	X	X	X	(An)
Address Register Indirect with Postincrement	011	reg. no.	X	X	—	X	(An)+
Address Register Indirect with Predecrement	100	reg. no.	X	X	—	X	–(An)
Address Register Indirect with Displacement	101	reg. no.	X	X	X	X	(d_{16},An)
Address Register Indirect with Index (8-Bit Displacement)	110	reg. no.	X	X	X	X	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	110	reg. no.	X	X	X	X	(bd,An,Xn)
Memory Indirect Postindexed	110	reg. no.	X	X	X	X	([bd,An],Xn,od)
Memory Indirect Preindexed	110	reg. no.	X	X	X	X	([bd,An,Xn],od)
Absolute Short	111	000	X	X	X	X	(xxx).W
Absolute Long	111	001	X	X	X	X	(xxx).L
Program Counter Indirect with Displacement	111	010	X	X	X	—	(d_{16},PC)
Program Counter Indirect with Index (8-Bit) Displacement)	111	011	X	X	X	—	(d_8,PC,Xn)
Program Counter Indirect with Index (Base Displacement)	111	011	X	X	X	—	(bd,PC,Xn)
PC Memory Indirect Postindexed	111	011	X	X	X	—	([bd,PC],Xn,od)
PC Memory Indirect Preindexed	111	011	X	X	X	—	([bd,PC,Xn],od)
Immediate	111	100	X	X	—	—	#⟨data⟩

4.5.2 Instruction Description Format

The details of each instruction are provided in **4.6 INDIVIDUAL INSTRUCTION DESCRIPTIONS**. Figure 4-1 illustrates what information is given in these instruction descriptions.

4.5.3 Operation Tables

An operation table is included for most instructions. This table lists the result data types for the instruction based on types of input operand(s). For example, Figure 4-2 illustrates the table for the FADD instruction.

In this table, the type of source operand is shown along the top, and the type of the destination operand is shown along the side. In-range numbers are normalized, denormalized, or unnormalized real numbers, integers, or packed decimal numbers that are converted to normalized or denormalized extended precision numbers upon entering the FPCP.

From this table, it can be seen that if both the source and destination operand are positive zero, the result is also a positive zero. For another example, if the source operand is a positive zero and the destination operand is an in-range number, then the ADD algorithm

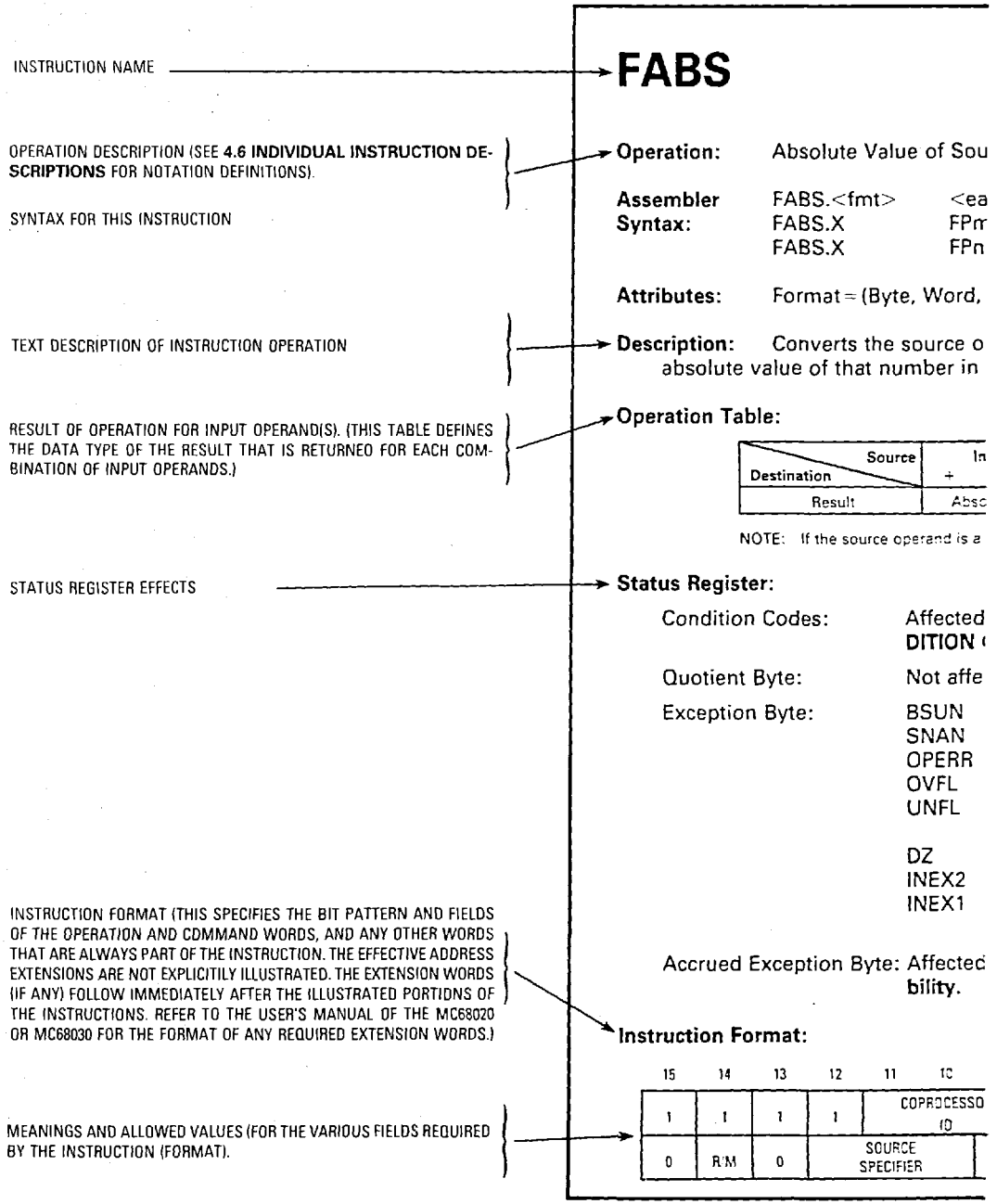


Figure 4-1. Instruction Description Format

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range +	Add		Add		+inf	−inf
In Range −						
Zero +	Add		+0.0 0.0[1]	0.0[1] −0.0	+inf	−inf
Zero −						
Zero +	+inf −inf		+inf −inf		+inf NAN[2]	NAN[2] −inf
Zero −						

NOTES:
1. Returns +0.0 in rounding modes RN, RZ, and RP; returns −0.0 in RM.
2. Sets the OPERR bit in the FPSR exception byte.
3. If *either operand is a NAN, refer to* **4.5.4 NANs** for more information.

Figure 4-2. Operation Table Example (FADD Instruction)

is executed to obtain the result. If a label such as ADD appears in the table, it indicates that the FPCP performs the indicated operation and returns the correct result.

A third example of using the tables is when a source operand is plus infinity and the destination operand is minus infinity. Since the result of such an operation is undefined, a not-a-number (NAN) is returned as the result, and the OPERR bit is set in the FPSR exception byte.

4.5.4 NANs

In addition to the data types covered in the operation tables for each instruction, NANs can also be used as inputs to an arithmetic operation. The operation tables do not contain a row and column for NANs because NANs are handled the same way in all operations.

4.5.4.1 NONSIGNALING NANs. If either, but not both, operand of an operation is a NAN, and it is a nonsignaling NAN, then that NAN is returned as the result. If both operands are nonsignaling NANs, then the destination operand nonsignaling NAN is returned as the result.

4.5.4.2 SIGNALING NANs. If either operand to an operation is a signaling NAN (SNAN), then the SNAN bit is set in the FPSR EXC byte. If the SNAN trap enable bit is set in the FPCR ENABLE byte, then the trap is taken and the destination is not modified. If the SNAN trap enable bit is not set, then the SNAN is converted to a nonsignaling NAN (by setting the SNAN bit in the operand to a one), and the operation continues as described in the preceding paragraph for nonsignaling NANs.

4.5.5 Operation Post Processing

Most floating-point operations end with an identical post processing step. While reading the summary for each instruction, it should be assumed that an instruction performs post processing unless the summary specifically states that the instruction does not do so. The following paragraphs describe post processing in detail.

4.5.5.1 SETTING FLOATING-POINT CONDITION CODES. Unlike the integer arithmetic condition codes found in the MC68020/MC68030, which are set uniquely for each instruction, the floating-point condition codes are either not changed by an instruction or are always set in the same way by an instruction. Therefore, it is not necessary to include details of condition code settings for each FPCP instruction in the detailed instruction descriptions. The following paragraphs describe how condition codes are set for all instructions that modify any condition codes.

Refer to **2.3.1 FPSR Floating-Point Condition Code Byte** for a description of the FPSR condition code byte. The four condition code bits are:

N	Sign of Mantissa	I	Infinity
Z	Zero	NAN	Not-A-Number

These condition code bits differ slightly from integer condition codes. The floating-point condition codes are not dependent on the type of operation being performed, but rather, can be set at the end of the operation by examining the result. (The M68000 integer condition code bits N and Z have this characteristic, but the V and C bits are set differently for different instructions.) At the end of any floating-point operation, the result is inspected, and the condition code bits are set or cleared accordingly. For example, if the result of an operation is a positive normalized number, then all of the condition code bits are set to zero. If the result is a minus infinity, then the N and I bits are set, and the Z and NAN bits are cleared.

Refer to **2.3.1 FPSR Floating-Point Condition Code Byte** for a description of the use of these bits to generate the four conditions required by the IEEE floating-point standard. Refer to **4.4 CONDITIONAL TEST DEFINITIONS** for a description of the use of the four condition code bits to generate the 32 floating-point conditional tests.

4.5.5.2 UNDERFLOW, ROUND, OVERFLOW. During calculation of an arithmetic result, the ALU of the FPCP has more precision and range than the 80-bit extended precision format. However, the final result of these operations is an extended precision floating-point value. In some cases, an internal result becomes either smaller or larger than can be represented in extended precision. Also, the operation may have generated a larger exponent or more bits of precision than can be represented in the chosen rounding precision. For these reasons, every arithmetic instruction ends by rounding the result and checking for overflow and underflow.

At the completion of an arithmetic operation, the internal result is checked to see if it is too small to be represented as a normalized number in the selected precision. If so, the underflow (UNFL) bit is set in the FPSR EXC byte. It is also denormalized unless denormalization provides a zero value. Denormalizing a number causes a loss of accuracy, but a zero is not returned unless absolutely necessary. If a number is grossly underflowed, the FPCP returns a correctly signed zero or the correctly signed smallest denormalized number, depending on the rounding mode in effect. For more details on underflow, refer to **6.1.5 Underflow**.

If no underflow occurs, the internal result is rounded according to the user-selected rounding precision and rounding mode. Refer to Figure 6-3 for a detailed description of rounding. After rounding, the inexact bit (INEX2) is set appropriately. Lastly, the magnitude of the result is checked to see if it is too large to be represented in the current rounding precision. If so, the overflow (OVFL) bit is set and a correctly signed infinity or correctly signed largest

normalized number is returned, depending on the rounding mode in effect. For details on overflow refer to **6.1.4 Overflow.**

Two important exceptions to the above description are: the execution of the FSGLDIV instruction and of the FSGLMUL instruction. For these two instructions, the rounding precision programmed in the mode control byte is ignored (although the selected rounding mode is used). The input operands to these instructions are assumed to be single-precision values, but no checking is performed to verify the inputs (each mantissa is truncated to 23 bits, and the exponent is accepted as an extended-precision value).

These two instructions first check the intermediate result for undeflow as previously described, but use the underflow threshold of extended precision regardless of the selected rounding precision. If no underflow occurs, the mantissa is rounded to the single-precision boundary and is denormalized if necessary. Finally, the exponent is checked for overflow, again using the overflow threshold of extended precision. Thus, the final result generated has the range of an extended-precision number with a mantissa accurate to only 23 bits. If an underflow or overflow occurs, the correctly signed number returned (largest normalized number, infinity, zero, or smallest denormalized number) is an extended precision-number with an extended-precision mantissa value.

4.6 INDIVIDUAL INSTRUCTION DESCRIPTIONS

The following notation is used in the detailed instruction definitions that follow:

(operand)	Contents of the referenced location or register.
<fmt>	Operand data format: Byte, word, long, single, double, extended, or packed (denoted in the assembler syntax as an extension to the instruction mnemonic of .B, .W, .L, .S, .D, .X, or .P, respectively).
<ea>	Any valid MC68020/MC68030 addressing mode.
<label>	A relative label used by an assembler to calculate a displacement.
<list>	A list of the floating-point data registers or control registers.
➧	The left operand is moved to the location specified by the right operand.
FPcr	One of the three floating-point system control registers (FPCR, FPSR, or FPIAR).
FPn	One of eight floating-point data registers (always specifies the destination register).
FPm	One of eight floating-point data registers (always specifies the source register).
FPc:FPs	Two of eight floating-point data registers. This notation is used only with the FSINCOS instruction and specifies the register pair where the cosine and sine values are stored.
+inf	Postive infinity
−inf	Negative infinity
NAN	Not-A-Number
d	Displacement
k	An integer (−64 to +17) that specifies the format of a number to be stored in the packed BCD format.
ccc	An index into the FPCP constant ROM.

Operation: Absolute Value of Source ➧ FPn

Assembler Syntax:

FABS.<fmt>	<ea>,FPn
FABS.X	FPm,FPn
FABS.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and stores the absolute value of that number in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Absolute Value		Absolute Value		Absolute Value	

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES.**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**
OPERR	Cleared
OVFL	Cleared
UNFL	If the source is an extended precision denormalized number, refer to **6.1.5 Underflow**; cleared otherwise.
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility.**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is then written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Arc Cosine of Source ➧ FPn

Assembler Syntax:

FACOS.<fmt>	<ea>,FPn
FACOS.X	FPm,FPn
FACOS.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the arc cosine of that number. Stores the result in the destination floating-point data register. This function is not defined for source operands outside of the range [−1 . . . +1]; if the source is not in the correct range, a NAN is returned as the result and the OPERR bit is set in the FPSR. If the source is in the correct range, the result is in the range of [0 . . . π].

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Arc Cosine		+π/2		NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES.**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is infinity, > −1 or < −1; cleared otherwise.
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility.**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M = 0, this field is unused, and should be all zeros.
If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M = 0, specifies the source floating-point data register, FPm.
If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn. If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is then written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Source + FPn ◗ FPn

Assembler Syntax:

FADD.<fmt>	<ea>,FPn
FADD.X	FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Converts the source operand to extended precision (if necessary) and adds that number to the number contained in the destination floating-point data register. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Add		Add		+inf	−inf
	−						
Zero	+	Add		+0.0 0.0[1]	0.0[1] −0.0	−inf	−inf
	−						
Infinity	+	+inf −inf		+inf −inf		−inf NAN[2]	NAN[2] −inf
	−						

NOTES:
1. Returns +0.0 in rounding modes RN, RZ, and RP; returns −0.0 in RM.
2. Sets the OPERR bit in the FPSR exception byte.
3. If either operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES.**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source and the destination are opposite-signed infinities; cleared otherwise.
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility.**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: Arc Sine of the Source ➧ FPn

Assembler Syntax:

FASIN.<fmt>	<ea>,FPn
FASIN.X	FPm,FPn
FASIN.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the arc sine of the number. Stores the result in the destination floating-point data register. This function is not defined for source operands outside of the range $[-1 \ldots +1]$; if the source is not in the correct range, a NAN is returned as the result and the OPERR bit is set in the FPSR. If the source is in the correct range, the result is in the range of $[-\pi/2 \ldots +\pi/2]$.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Arc Sine		+0.0	−0.0	NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**.

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is infinity, > +1 or < −1; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	1	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is then written into the same register. If the single register syntax is used, **Freescale** assemblers set the source and destination fields to the same value.

Operation: Arc Tangent of Source ➧ FPn

Assembler Syntax:

FATAN.<fmt>	<ea>,FPn
FATAN.X	FPm,FPn
FATAN.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Converts the source operand to extended precision (if necessary) and calculates the arc tangent of that number. Stores the result in the destination floating-point data register. The result is in the range of $[-\pi/2 \ldots +\pi/2]$.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Arc Tangent		+0.0	−0.0	$+\pi/2$	$-\pi/2$

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES.**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	0	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is then written into the same register. If the single register syntax is used, **Freescale** assemblers set the source and destination fields to the same value.

Operation: Hyperbolic Arc Tangent of Source ➧ FPn

Assembler Syntax:

FATANH.<fmt>	<ea>,FPn
FATANH.X	FPm,FPn
FATANH.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the hyperbolic arc tangent of that value. Stores the result in the destination floating-point data register. This function is not defined for source operands outside of the range (−1 . . . +1); and the result is equal to −infinity or +infinity if the source is equal to +1 or −1, respectively. If the source is outside of the range [−1 . . . ÷1], a NAN is returned as the result and the OPERR bit is set in the FPSR.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Hyperbolic Arc Tangent		+0.0	−0.0	NAN[1]	

NOTE:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is > +1 or < −1; cleared otherwise
OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow.**
DZ	Set if the source is equal to ÷1 or −1; cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	1	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M = 0, this field is unused, and should be all zeros.
If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M = 0, specifies the source floating-point data register, FPm.
If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.
If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is then written into the same register. If the single register syntax is used, **Freescale** assemblers set the source and destination fields to the same value.

Operation: If condition true, then PC + d ➧ PC

Assembler Syntax: FBcc.<size> <label>

Attributes: Size = (Word, Long)

4

Description: If the specified floating-point condition is met, program execution continues at the location (PC) + displacement. The displacement is a twos-complement integer that counts the relative distance in bytes. The value of the PC used to calculate the destination address is the address of the branch instruction plus two. If the displacement size is word, then a 16-bit displacement is stored in the word immediately following the instruction operation word. If the displacement size is long word, then a 32-bit displacement is stored in the two words immediately following the instruction operation word.

The conditional specifier cc selects any one of the 32 floating-point conditional tests as described in **4.4 CONDITIONAL TEST DEFINITIONS.**

Status Register:

Condition Codes:	Not affected	
Quotient Byte:	Not affected	
Exception Byte:	BSUN	Set if the NAN condition code is set and the condition selected is an IEEE nonaware test
	SNAN	Not Affected
	OPERR	Not Affected
	OVFL	Not Affected
	UNFL	Not Affected
	DZ	Not Affected
	INEX2	Not Affected
	INEX1	Not Affected

Accrued Exception Byte: The IOP bit is set if the BSUN bit is set in the exception byte. No other bit is affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	1	SIZE	CONDITIONAL PREDICATE					
16-BIT DISPLACEMENT, OR MOST SIGNIFICANT WORD OF 32-BIT DISPLACEMENT															
LEAST SIGNIFICANT WORD OF 32-BIT DISPLACEMENT (IF NEEDED)															

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Size Field — Specifies the size of the signed displacement:

If Format = 0, then the displacement is 16-bits and is sign extended before use.

If Format = 1, then the displacement is 32-bits.

Conditional Predicate Field — Specifies one of 32 conditional tests as defined in **4.4 CONDITIONAL TEST DEFINITIONS**.

NOTE

When a BSUN exception occurs, the main processor takes a pre-instruction exception. If the exception handler returns without modifying the image of the PC on the stack frame (to point to the instruction following the FBcc), then it must clear the cause of the exception (by clearing the NAN bit or disabling the BSUN trap) or the exception occurs again immediately upon return to the routine that caused the exception.

Operation: FPn — Source

Assembler Syntax: FCMP.<fmt> <ea>,FPn
FCMP.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and subtracts the operand from the destination floating-point data register. The result of the subtraction is not retained, but it is used to set the floating-point condition codes as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES.**

Operation Table The entries in this operation table differ from those of the tables describing most of the FPCP instructions. For each combination of input operand types, the condition code bits that may be set are indicated. If the name of a condition code bit is given and is not enclosed in brackets, then it is always set. If the name of a condition code bit is enclosed in brackets, then that bit is either set or cleared, as appropriate. If the name of a condition code bit is not given, then that bit is always cleared by the operation. The infinity bit is always cleared by the FCMP instruction, since it is not used by any of the conditional predicate equations. Note that the NAN bit is not shown, since NANs are always handled in the same manner (as described in **4.5.4 NANs**).

Destination \ Source		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	{NZ}	none	none	none	N	none
	−	N	{NZ}	N	N	N	none
Zero	+	N	none	Z	Z	N	none
	−	N	none	NZ	NZ	N	none
Infinity	+	none	none	none	none	Z	none
	−	N	N	N	N	N	NZ

NOTE: If either operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in the operation table above

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Cleared
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	1	1	0	0	0

4

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: Cosine of Source ➧ FPn

Assembler Syntax:

FCOS.<fmt>	<ea>,FPn
FCOS.X	FPm,FPn
FCOS.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the cosine of that number. Stores the result in the destination floating-point data register. This function is not defined for source operands of (±) infinity. If the source operand is not in the range of $[-2\pi \ldots +2\pi]$, then the argument is reduced to within that range before the cosine is calculated. However, large arguments may lose accuracy during reduction, and very large arguments (greater than approximately 10^{20}) lose all accuracy. The result is in the range of $[-1 \ldots +1]$.

Operation Table:

Source / Destination	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Cosine		+1.0		NAN[1]	

NOTE:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source operand is (+ or −)infinity; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility.**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	1	1	1	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M=0, this field is unused, and should contain zeros.
If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M=0, specifies the source floating-point data register, FPm.
If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.
If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, **Freescale** assemblers set the source and destination fields to the same value.

Operation: Hyperbolic Cosine of Source ➧ FPn

Assembler Syntax:

FCOSH.<fmt>	<ea>,FPn
FCOSH.X	FPm,FPn
FCOSH.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the hyperbolic cosine of that number. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Hyperbolic Cosine		+1.0		+inf	

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	0	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn. If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, **Freescale** assemblers set the source and destination fields to the same value.

Operation: If condition true then no operation
else Dn − 1 ➧ Dn
if Dn ≠ −1
then PC + d ➧ PC
else execute next instruction

Assembler Syntax: FDBcc Dn,<label>

Attributes: Unsized

Description: This instruction is a looping primitive of three parameters: a floating-point condition, a counter (an MPU data register) and a 16-bit displacement. The FPCP first tests the condition to determine if the termination condition for the loop has been met, and if so, the main processor proceeds to execute the next instruction in the instruction stream. If the termination condition is not true, the low order 16-bits of the counter register are decremented by one. If the result is −1, the count is exhausted, and execution continues with the next instruction. If the result is not equal to −1, execution continues at the location specified by the current value of the PC plus the sign-extended 16-bit displacement. The value of the PC used in the branch address calculation is the address of the displacement word.

The conditional specifier cc selects any one of the 32 floating-point conditional tests as described in **4.4 CONDITIONAL TEST DEFINITIONS**.

Status Register:

Condition Codes:	Not affected	
Quotient Byte:	Not affected	
Exception Byte:	BSUN	Set if the NAN condition code is set and the condition selected is an IEEE nonaware test
	SNAN	Not Affected
	OPERR	Not Affected
	OVFL	Not Affected
	UNFL	Not Affected
	DZ	Not Affected
	INEX2	Not Affected
	INEX1	Not Affected

Accrued Exception Byte: The IOP bit is set if the BSUN bit is set in the exception byte. No other bit is affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	0	0	1	COUNT REGISTER		
0	0	0	0	0	0	0	0	0	0	CONDITIONAL PREDICATE					
16-BIT DISPLACEMENT															

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID = 1 for the FPCP.

Count Register Field — Specifies main processor data register that is used as the counter.

Conditional Predicate Field — Specifies one of the 32 floating-point conditional tests as described in **4.4 CONDITIONAL TEST DEFINITIONS**.

Displacement Field — Specifies the branch distance (from the address of the instruction plus two) to the destination in bytes.

NOTES:

1. The terminating condition is like that defined by the UNTIL loop constructs of high-level languages. For example: FDBOLT can be stated as "decrement and branch until ordered less than".
2. There are two basic ways of entering a loop: at the beginning, or by branching to the trailing FDBcc instruction. If a loop structure terminated with FDBcc is entered at the beginning, the control counter must be one less than the number of loop executions desired. This count is useful for indexed addressing modes and dynamically specified bit operations. However, when entering a loop by branching directly to the trailing FDBcc instruction, the count should equal the loop execution count. In this case, if the counter is zero when the loop is entered, the FDBcc instruction does not branch, causing a complete bypass of the main loop.
3. When a BSUN exception occurs, a pre-instruction exception is taken by the main processor. If the exception handler returns without modifying the image of the PC on the stack frame (to point to the instruction following the FDBcc), then it must clear the cause of the exception (by clearing the NAN bit or disabling the BSUN trap) or the exception occurs again immediately upon return to the routine that caused the exception.

Operation: FPn (÷) Source ➧ FPn

Assembler Syntax:
FDIV.<fmt> <ea>,FPn
FDIV.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and divides that number into the number in the destination floating-point data register. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source		In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range	+	Divide		+inf[1]	−inf[1]	+0.0	−0.0
	−			−inf[1]	+inf[1]	−0.0	+0.0
Zero	+	+0.0	−0.0	NAN[2]		+0.0	−0.0
	−	−0.0	+0.0			−0.0	+0.0
Infinity	+	+inf	−inf	−inf	−inf	NAN[2]	
	−	−inf	+inf	−inf	−inf		

NOTES:
1. Sets the DZ bit in the FPSR exception byte.
2. Sets the OPERR bit in the FPSR exception byte.
3. If either operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: *Not affected*

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set for 0(÷)0 or infinity(÷)infinity; cleared otherwise
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Set if the source is zero and the destination is in range; cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M = 0, this field is unused, and should be all zeros.
If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
# data	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M = 0, specifies the source floating-point data register, FPm.
If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: e(Source) ➧ FPn

Assembler Syntax:

FETOX.<fmt>	<ea>,FPn
FETOX.X	FPm,FPn
FETOX.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Converts the source operand to extended precision (if necessary) and calculates e to the power of that number. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	e^X		+1.0		+inf	−0.0

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, **Freescale** assemblers set the source and destination fields to the same value.

Operation: $e^{(Source)} - 1 \blacklozenge FPn$

Assembler Syntax:
FETOXM1.<fmt> <ea>,FPn
FETOXM1.X FPm,FPn
FETOXM1.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates e to the power of that number. Then, subtracts one from that value, and stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	$e^X - 1$		+0.0	−0.0	−inf	−1.0

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Cleared
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow**.
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, *refer to* **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Exponent of Source ➧ FPn

Assembler Syntax:
FGETEXP.<fmt> <ea>,FPn
FGETEXP.X FPm,FPn
FGETEXP.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and extracts the binary exponent. Removes the exponent bias, converts the exponent to an extended precision floating-point number, and stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Exponent		+0.0	−0.0	NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is (+ or −)infinity; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	1	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Mantissa of Source ➧ FPn

Assembler Syntax:
FGETMAN.<fmt> <ea>,FPn
FGETMAN.X FPm,FPn
FGETMAN.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and extracts the mantissa. Converts the mantissa to an extended precision value and stores the result in the destination floating-point data register. The result is in the range [1.0 . . . 2.0) with the sign of the source mantissa, zero, or is a NAN.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Mantissa		+0.0	−0.0	NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is (+ or −)infinity; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	1	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M=0, this field is unused, and should be all zeros.
If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M=0, specifies the source floating-point data register, FPm.
If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.
If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Integer Part of Source → FPn

Assembler Syntax:

FINT.<fmt>	<ea>,FPn
FINT.X	FPm,FPn
FINT.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Converts the source operand to extended precision (if necessary) and extracts the integer part and converts it to an extended precision floating-point number. Stores the result in the destination floating-point data register. The integer part is extracted by rounding the extended precision number to an integer using the current rounding mode selected in the FPCR mode control byte. Thus, the integer part returned is the number that is to the left of the radix point when the exponent is zero, after rounding. For example, the integer part of 137.57 is 137.0 for the round-to-zero and round-to-minus infinity modes, and 138.0 for the round-to-nearest and round-to-plus infinity modes. Note that the result of this operation is a floating-point number.

Operation Table:

Source / Destination	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Integer		+0.0	−0.0	+inf	−inf

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input;** cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	0	1

4

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Integer Part of Source ➧ FPn

Assembler Syntax:

FINTRZ.<fmt>	<ea>,FPn
FINTRZ.X	FPm,FPn
FINTRZ.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and extracts the integer part and converts it to an extended precision floating-point number. Stores the result in the destination floating-point data register. The integer part is extracted by rounding the extended precision number to an integer using the round-to-zero mode, regardless of the rounding mode selected in the FPCR mode control byte (making it useful for FORTRAN assignments). Thus, the integer part returned is the number that is to the left of the radix point when the exponent is zero. For example, the integer part of 137.57 is 137.0; the integer part of 0.1245 × 102 is 12.0. Note that the result of this operation is a floating-point number.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Integer, Forced Round-To-Zero		+0.0	−0.0	+inf	−inf

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input;** cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	0	1	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If RM = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Log_{10} of Source ➧ FPn

Assembler Syntax:
FLOG10.<fmt> <ea>,FPn
FLOG10.X FPm,FPn
FLOG10.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Convert the source operand to extended precision (if necessary) and calculates the logarithm of that number using base 10 arithmetic. Stores the result in the destination floating-point data register. This function is not defined for input values less than zero.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Log_{10}	NAN[1]	−inf[2]		+inf	NAN[1]

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. Sets the DZ bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source operand is <0; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Set if the source is (+ or −); cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	1	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Log_2 of Source ⇒ FPn

Assembler Syntax:

FLOG2.<fmt>	<ea>,FPn
FLOG2.X	FPm,FPn
FLOG2.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the logarithm of that number using base two arithmetic. Stores the result in the destination floating-point data register. This function is not defined for input values less than zero.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Log_2	NAN[1]	−inf[2]		+inf	NAN[1]

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. Sets the DZ bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is < 0; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Set if the source is (+ or −)0; cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	1	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M = 0, this field is unused, and should be all zeros.
If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M = 0, specifies the source floating-point data register, FPm.
If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.
If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: $\log_e$ of Source ➧ FPn

Assembler Syntax:
FLOGN.<fmt> <ea>,FPn
FLOGN.X FPm,FPn
FLOGN.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Converts the source operand to extended precision (if necessary) and calculates the natural logarithm of that number. Stores the result in the destination floating-point data register. This function is not defined for input values less than zero.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	ln(x)	NAN[1]	−inf[2]		+inf	NAN[1]

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. Sets the DZ bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source operand is < 0; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Set if the source is (+ or −)0; cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	1	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: $\log_e$ of (Source + 1) ➧ FPn

Assembler Syntax:
FLOGNP1.<fmt> <ea>,FPn
FLOGNP1.X FPm,FPn
FLOGNP1.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary), adds one to that value, and calculates the natural logarithm of that intermediate result. Stores the result in the destination floating-point data register. This function is not defined for input values less than −1.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	ln(x+1)	ln(x+1)[1]	+0.0	−0.0	−inf	NAN[2]

NOTES:
1. If the source is −1, sets the DZ bit in the FPSR exception byte and returns a NAN. If the source is < −1, sets the OPERR bit in the FPSR exception byte and returns a NAN.
2. Sets the OPERR bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source operand is < −1; cleared otherwise
OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow.**
DZ	Set if the source operand is −1; cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	1	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M=0, this field is unused, and should be all zeros.
If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M=0, specifies the source floating-point data register, FPm.
If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.
If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Modulo Remainder of (FPn (÷) Source) ➧ FPn

Assembler Syntax:
FMOD.<fmt> <ea>,FPn
FMOD.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the modulo remainder of the number in the destination floating-point data register, using the source operand as the modulus. Stores the result in the destination floating-point data register, and stores the sign and seven least significant bits of the quotient in the FPSR quotient byte (the quotient is the result of FPn (÷) Source). The modulo remainder function is defined as:

$$FPn - (Source \times N)$$

where:

N = INT(FPn (/) Source) in the round-to-zero mode

The FMOD function is not defined for a source operand equal to zero or for a destination operand equal to infinity. Note that this function is not the same as the FREM instruction, which uses the round-to-nearest mode and thus returns the remainder that is required by the *IEEE Specification for Binary Floating-Point Arithmetic.*

Operation Table:

Destination \ Source		In Range	Zero	Infinity
		+ −	+ −	+ −
In Range	+ −	Modulo Remainder	NAN[1]	FPn[2]
Zero	+ −	+0.0 −0.0	NAN[1]	+0.0 −0.0
Infinity	+ −	NAN[1]	NAN[1]	NAN[1]

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. Returns the value of FPn before the operation. However, the result is processed by the normal instruction termination procedure to round it as required. Thus, an overflow and or inexact result may occur if the rounding precision has been changed to a smaller size since the FPn value was loaded.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Loaded with the sign and least significant seven bits of the quotient (FPn (÷) Source). The sign of the quotient is the exclusive OR of the sign bits of the source and destination operands.

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is zero, or the destination is infinity; cleared otherwise
OVFL	Cleared

UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: Source ➧ Destination

Assembler Syntax:

FMOVE.<fmt>	<ea>,FPn
FMOVE.<fmt>	FPm,<ea>
FMOVE.P	FPm,<ea>{Dn}
FMOVE.P	FPm,<ea>{#k}

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Moves the contents of the source operand to the destination operand. Although the primary function of this instruction is data movement, it is also considered an arithmetic instruction since conversions from the source operand format to the destination operand format are performed implicitly during the move operation. Also, the source operand is rounded according to the selected rounding precision and mode.

Unlike the M68000 Family integer data movement instruction, the floating-point move instruction does not support a memory-to-memory format (for such transfers, it is much faster to utilize the M68000 Family integer MOVE instruction to transfer the floating-point data than to use the FMOVE instruction). The FMOVE instruction only supports memory-to-register, register-to-register, and register-to-memory operations (in this context, memory may refer to an MPU data register if the data format is byte, word, long or single). The memory-to-register and register-to-register operations use a command word encoding distinctly different from that used by the register-to-memory operation, and these two operation classes are described separately below.

Memory-to-Register or Register-to-Register Operation:

Converts the source operand to an extended precision floating-point number (if necessary) and stores it in the destination floating-point data register. Depending on the source data format and the rounding precision, some operations may produce an inexact result. In the following table, combinations that can produce an inexact result are marked with a dot (•), but all other combinations produce an exact result.

	Source Format:	B	W	L	S	D	X	P
Rounding Precision:	Single			•		•	•	•
	Double						•	•
	Extended							•

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Cleared
OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow** if the source is an extended precision denormalized number; cleared otherwise.
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result** if <fmt> is L, D or X; cleared otherwise.
INEX1	Refer to **6.1.8 Inexact Result on Decimal Input** if <fmt> is P; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.
Source Specifier Field — Specifies the source register or data format.
If R/M = 0, specifies the source floating-point data register, FPm.
If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Register-to-Memory Operation:

Rounds the source operand to the size of the specified destination format and stores it at the destination effective address. If the format of the destination is packed decimal, a third operand is required to specify the format of the resultant string. This operand, called the k-factor, is a 7-bit signed integer (twos complement) and may be specified as an immediate value or in a main processor data register. If a data register contains the k-factor, only the least significant seven bits are used, and the rest of the register is ignored.

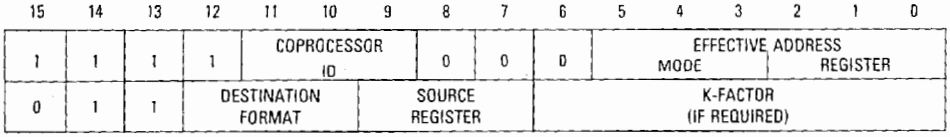
Status Register:

Condition Codes:	Not affected	
Quotient Byte:	Not affected	
Exception Byte:	BSUN	Cleared
<fmt> is B, W, or L	SNAN	Refer to **4.5.4 NANs.**
	OPERR	Set if the source operand is infinity, or if the destination size is exceeded after conversion and rounding; cleared otherwise
	OVFL	Cleared
	UNFL	Cleared
	DZ	Cleared
	INEX2	Refer to **6.1.7 Inexact Result.**
	INEX1	Cleared

<fmt> is S, D, or X	BSUN	Cleared
	SNAN	Refer to **4.5.4 NANs**.
	OPERR	Cleared
	OVFL	Refer to **6.1.4 Overflow**.
	UNFL	Refer to **6.1.5 Underflow**.
	DZ	Cleared
	INEX2	Refer to **6.1.7 Inexact Result**.
	INEX1	Cleared
<fmt> is P	BSUN	Cleared
	SNAN	Refer to **4.5.4 NANs**.
	OPERR	Set if the k-factor > +17, or the magnitude of the decimal exponent exceeds 3 digits; cleared otherwise
	OVFL	Cleared
	UNFL	Cleared
	DZ	Cleared
	INEX2	Refer to **6.1.7 Inexact Result**.
	INEX1	Cleared

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:



Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Encoded with the M68000 addressing mode for the destination operand as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Only if <fmt> is Byte, Word, Long, or Single.

Destination Format Field — Specifies the data format of the destination operand:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P{#k}	Packed Decimal Real with static k-factor
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer
111	P{Dn}	Packed Decimal Real with dynamic k-factor

Source Register Field — Specifies the source floating-point data register, FPm.

k-factor Field — Only used if the destination format is Packed Decimal, to specify the format of the decimal string. For any other destination format, this field should be set to all zeros. For a static k-factor, this field is encoded with a twos-complement integer where the value defines the format as follows:

−64 to 0 — Indicates the number of significant digit to the right of the decimal point (Fortran "F" format).

+1 to +17 — Indicates the number of significant digits in the mantissa (Fortran "E" format).

+18 to +63 — Sets the OPERR bit in the FPSR exception byte, treated as +17.

The format of this field for a dynamic k-factor is:

r r r 0 0 0 0

where:

"rrr" is the number of the main processor data register that contains the k-factor value.

The following table gives several examples of how the k-factor value affects the format of the decimal string that is produced by the FPCP. The format of the string that is generated is independent of the source of the k-factor (static or dynamic).

k-Factor	Source Operand Value	Destination String
−5	+12345.678765	+1.234567877 E+4
−3	+12345.678765	+1.2345679 E+4
−1	+12345.678765	+1.23457 E+4
0	+12345.678765	+1.2346 E+4
+1	+12345.678765	+1. E+4
+3	+12345.678765	+1.23 E+4
+5	+12345.678765	+1.2346 E+4

Operation: Source ➧ Destination

Assembler Syntax:
FMOVE.L <ea>,FPcr
FMOVE.L FPcr,<ea>

Attributes: Size = (Long)

Description: Moves the contents of a floating-point system control register into or out of the FPCP (the control registers are the FPCR, FPSR and FPIAR). The external operand may be in memory or a main processor register. A 32-bit transfer is always performed, even though the system control register may not have 32 implemented bits. Unimplemented bits of a control register are read as zeros and are ignored during writes (but must be zero for compatability with future devices).

This instruction does not cause pending exceptions (other than protocol violations) to be reported to the main processor. Furthermore, a write to the FPCR exception enable byte or the FPSR exception status byte cannot generate a new exception, regardless of the value written.

Status Register: Changed only if the destination is the FPSR; in which case all bits are modified to reflect the value of the source operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
1	0	dr	REGISTER SELECT			0	0	0	0	0	0	0	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for the operation:

Memory-to-Register

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if the source register is the FPIAR.

Register-to-Memory

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Only if the destination register is the FPIAR.

dr Field — Specifies the direction of the data transfer.
0 — Move an external operand to the specified system control register.
1 — Move the specified system control register to an external location.

Register Select Field — Specifies the system control register to be moved:

100	FPCR	Floating-point Control Register
010	FPSR	Floating-point Status Register
001	FPIAR	Floating-point Instruction Address Register

Operation: ROM Constant ➧ FPn

Assembler Syntax: FMOVECR.X #ccc,FPn

Attributes: Format = (Extended)

Description: Fetches an extended precision constant from the FPCP on-chip ROM, rounds it to the precision specified in the FPCR mode control byte, and stores it in the destination floating-point data register. The constant is specified by a predefined offset into the constant ROM. The values of the constants contained in the ROM are shown in the offset table at the end of this description.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Cleared
OPERR	Cleared
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	Cleared

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	0	0	0	0	0	0
0	1	0	1	1	1	DESTINATION REGISTER			ROM OFFSET						

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Destination Register Field — Specifies the destination floating-point data register, FPn.

ROM Offset Field — Specifies the offset into the FPCP on-chip constant ROM where the desired constant is located. The offsets for the available constants are:

Offset	Constant
$00	π
$0B	$\log_{10}(2)$
$0C	e
$0D	$\log_2(e)$
$0E	$\log_{10}(e)$
$0F	0.0
$30	ln(2)
$31	ln(10)
$32	10^0
$33	10^1
$34	10^2
$35	10^4
$36	10^8
$37	10^{16}
$38	10^{32}
$39	10^{64}
$3A	10^{128}
$3B	10^{256}
$3C	10^{512}
$3D	10^{1024}
$3E	10^{2048}
$3F	10^{4096}

The on-chip ROM contains other constants useful only to the on-chip microcode routines. The values contained at offsets other than those defined above are reserved for the use of Freescale, and may be different on various mask sets of the FPCP.

Operation: Register List ⧫ Destination
Source ⧫ Register List

Assembler Syntax:

FMOVEM.X	<list>,<ea>
FMOVEM.X	Dn,<ea>
FMOVEM.X	<ea>,<list>
FMOVEM.X	<ea>,Dn

<list> A list of any combination of the eight floating-point data registers, with individual register names separated by a slash (/); and/or contiguous blocks of registers specified by the first and last register names separated by a dash (–).

Attributes: Format = (Extended)

Description: Moves one or more extended precision numbers to or from a list of floating-point data registers. No conversion or rounding is performed during this operation, and the FPSR is not affected by the instruction. This instruction does not cause pending exceptions (other than protocol violations) to be reported to the main processor.

Any combination of the eight floating-point data registers can be transferred, with the selected registers specified by a user-supplied mask. This mask is an 8-bit number, where each bit corresponds to one register; if a bit is set in the mask, that register is moved. The register select mask may be specified as a static value contained in the instruction, or a dynamic value in the least significant 8-bits of a main processor data register (the remaining bits of the register are ignored).

FMOVEM allows three types of addressing modes: the control modes, the predecrement mode, or the postincrement mode. If the effective address is one of the control addressing modes, the registers are transferred between the FPCP and memory starting at the specified address and up through higher addresses. The order of the transfer is from FP7–FP0.

If the effective address is the predecrement mode, only a register-to-memory operation is allowed. The registers are stored starting at the address contained in the address register and down through lower addresses. Before each register is stored, the address register is decremented by 12 (the size of an extended precision number in memory) and the floating-point data register is then stored at the resultant address. When the operation is complete, the address register points to the image of the last floating-point data register stored. Each register is stored in the format described in **SECTION 3 OPERAND DATA FORMATS**, with the most significant byte of the register image stored at the lowest address, and the least significant byte at the highest address. The order of the transfer is from FP7–FP0.

If the effective address is the postincrement mode, only a memory-to-register operation is allowed. The registers are loaded starting at the specified address and up

through higher addresses. After each register is stored, the address register is incremented by 12 (the size of an extended precision number in memory). When the operation is complete, the address register points to the byte immediately following the image of the last floating-point data register loaded. The order of the transfer is the same as for the control addressing modes: FP7–FP0.

Status Register: Not Affected. Note that the FMOVEM instruction provides the only mechanism for moving a floating-point data item between the FPCP and memory without performing any data conversions or affecting the condition code and exception status bits.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
1	1	dr	MODE		0	0	0	REGISTER LIST							

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for the operation:

Memory-to-Register

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register-to-Memory

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An)+	—	—			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

dr Field — Specifies the direction of the transfer.
0 — Move the listed registers from memory to the FPCP.
1 — Move the listed registers from the FPCP to memory.

Mode Field — Specifies the type of the register list and addressing mode.
00 — Static register list, predecrement addressing mode.
01 — Dynamic register list, predecrement addressing mode.
10 — Static register list, postincrement or control addressing mode.
11 — Dynamic register list, postincrement or control addressing mode.

Register List Field:
Static list — contains the register select mask; if a register is to be moved, the corresponding bit in the mask is set as shown below, otherwise it is clear.
Dynamic list — contains the main processor data register number, rrr, as shown below:

Register List Format

Static, −(An)	—	FP7	FP6	FP5	FP4	FP3	FP2	FP1	FP0
Static, (An)+ or Control	—	FP0	FP1	FP2	FP3	FP4	FP5	FP6	FP7
Dynamic	—	0	r	r	r	0	0	0	0

The format of the dynamic list mask is the same as for the static list and is contained in the least significant eight bits of the specified main processor data register.

Programming Note: This instruction provides a very useful feature, dynamic register list specification, that can significantly enhance system performance. If the calling conventions used for procedure calls utilize the dynamic register list feature, the number of floating-point data registers saved and restored can be reduced. A minimum of six bus cycles is required to load or save a floating-point data register (more if the memory address is not long word aligned). Thus, a minimum of 36 clock cycles (2×6 bus cycles×3 clocks per bus cycle) is eliminated from the procedure call and return overhead for each register not saved and restored unnecessarily.

In order to utilize the dynamic register specification feature of the FMOVEM instruction, both the calling and the called procedures must be written to communicate information about register usage. When one procedure calls another, a register mask must be passed to the called procedure to indicate which registers must not be altered upon return to the calling procedure. The called procedure then saves only those registers

that are modified and are already in use. There are several techniques that can be used to utilize this mechanism, and an example follows.

In this example, a convention is defined by which each called procedure is passed a word mask in D7 that identifies all floating-point registers in use by the calling procedure. Bits 15–8 identify the registers in the order FP0–FP7, and bits 0–7 identify the registers in the order FP0–FP7 (the two masks are required due to the different transfer order used by the predecrement and postincrement addressing modes). The code used by the calling procedure consists of simply moving the mask (which is generated at compile time) for the floating-point data registers currently in use into D7:

```
Calling procedure . . .
MOVE.W   #ACTIVE,D7     Load the list of FP registers that are in use
BSR      PROC_2
```

The entry code for all other procedures computes two masks. The first mask identifies the registers in use by the calling procedure that are used by the called procedure (and therefore saved and restored by the called procedure). The second mask identifies the registers in use by the calling procedure that are used by the called procedure (and therefore not saved on entry). The appropriate registers are then stored along with the two masks:

```
Called procedure . . .
MOVE.W   D7,D6          Copy the list of active registers
AND.W    #WILL_USE,D7   Generate the list of doubly-used registers
FMOVEM   D7,-(A7)       Save those registers
MOVE.W   D7,-(A7)       Save the register list
EOR.W    D7,D6          Generate the list of not saved active registers
MOVE.W   D6,P(A7)       Save it for later use
```

If the second procedure calls a third procedure, a register mask is passed to the third procedure that indicates which registers must not be altered by the third procedure. This mask identifies any registers in the list from the first procedure that were not saved by the second procedure, plus any registers used by the second procedure that must not be altered by the third procedure. An example of the calculation of this mask is:

```
Nested calling sequence . . .
MOVE.W   UNSAVED        Load the list of active registers not saved at entry
           (A7),D7
OR.W     #ACTIVE,D7     Combine with those active at this time
BSR      PROC_3
```

Upon return from a procedure, the restoration of the necessary registers follows the same convention, and the register mask generated during the save operation on entry is used to restore the required floating-point data registers:

```
Return to caller . . .
ADDQ.L   #2,A7          Discard the list of registers not saved
MOVE.B   (A7)+,D7       Get the saved register list (pop word, use byte)
FMOVEM   (A7)+,D7       Restore the registers
    •
    •
    •
RTS      Return to the
         calling routine
```

Operation: Register List ➧ Destination
Source ➧ Register List

Assembler Syntax:

FMOVEM.L	<list>,<ea>
FMOVEM.L	<ea>,<list>

<list> A list of any combination of the three floating-point system control registers (FPCR, FPSR and FPIAR) with individual register names separated by a slash (/).

Attributes: Size = (Long)

Description: Moves one or more 32-bit values into or out of the specified system control registers. Any combination of the three system control registers may be specified. The registers are always moved in the same order, regardless of the addressing mode used; with the FPCR moved first, followed by the FPSR, and the FPIAR moved last (if a register is not selected for the transfer, the relative order of the transfer of the other registers is the same). The first register is transferred between the FPCP and the specified address, with successive registers located up through higher addresses.

When more than one register is moved, the memory or memory-alterable addressing modes are allowed as shown in the addressing mode tables. If the addressing mode is predecrement, the address register is first decremented by the total size of the register images to be moved (i.e., four times the number of registers) and then the registers are transferred starting at the resultant address. For the postincrement addressing mode, the selected registers are transferred to or from the specified address, and then the address register is incremented by the total size of the register images transferred. If a single system control register is selected, the data register direct addressing mode may be used; or, if the only register selected is the FPIAR, then the address register direct addressing mode is allowed. Note that if a single register is selected, the opcode generated is the same as for the FMOVE single system control register instruction.

Status Register: Is changed only if the destination list includes the FPSR; in which case all bits are modified to reflect the value of the source register image.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
1	0	dr	REGISTER LIST				0	0	0	0	0	0	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for the operation:

Memory-to-Register

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An**	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if a single FPcr is selected.
**Only if the FPIAR is the single register selected.

Register-to-Memory

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An**	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Only if a single FPcr is selected.
**Only if the FPIAR is the single register selected.

dr Field — Specifies the direction of the transfer.
0 — Move the listed registers from memory to the FPCP.
1 — Move the listed registers from the FPCP to memory.
Register List Field: — Contains the register select mask; if a register is to be moved, the corresponding bit in the list is set, otherwise it is clear.

Bit Number	—	12	11	10
Register	—	FPCR	FPSR	FPIAR

Operation: Source × FPn ➧ FPn

Assembler Syntax:
FMUL.<fmt> <ea>,FPn
FMUL.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and multiplies that number by the number in the destination floating-point data register. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source		In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range	+	Multiply		+0.0 −0.0	−0.0 +0.0	+inf −inf	−inf +inf
	−						
Zero	+	+0.0 −0.0	−0.0 +0.0	+0.0 −0.0	−0.0 +0.0	NAN[1]	
	−						
Infinity	+	+inf −inf	−inf +inf	NAN[1]		+inf −inf	−inf +inf
	−						

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set for 0 × infinity; cleared otherwise
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	1	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: −(Source) ➧ FPn

Assembler Syntax:

FNEG.<fmt>	<ea>,FPn
FNEG.X	FPm,FPn
FNEG.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and inverts the sign of the mantissa. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Negate		−0.0	−0.0	−inf	−inf

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Cleared
UNFL	If source is an extended precision denormalized number, refer to **6.1.5 Underflow**; cleared otherwise.
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	1	1	0	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: None

Assembler Syntax: FNOP

Attributes: Unsized

Description: This instruction does not perform any explicit operation. However, it is useful to force synchronization of the FPCP with a main processor, or to force processing of pending exceptions. The synchronization function is inherent in the way that the FPCP uses the M68000 Family coprocessor interface. For most FPCP instructions, the main processor is allowed to continue with the execution of the next instruction once the FPCP has any operands needed for an operation, thus supporting concurrent execution of floating-point and integer instructions. However, if the main processor attempts to initiate the execution of a new floating-point instruction in the MC68881 before the previous one is completed, the main processor is forced to wait until that instruction execution is finished before proceeding with the new instruction. FNOP is treated in the same way as other instructions and thus cannot be executed until the previous floating-point instruction is completed, and the main processor is synchronized with the MC68881.

The MC68882 may not wait to begin execution of another floating-point instruction until it has completed execution of the current instruction. However, the FNOP instruction synchronizes the coprocessor and MPU by causing the MPU to wait until the current instruction (or both instructions) have completed.

The FNOP instruction also forces the processing of exceptions pending from the execution of previous instructions. This is also inherent in the way that the FPCP utilizes the M68000 Family coprocessor interface. Once the FPCP has received the input operand for an arithmetic instruction, it always releases the main processor to execute the next instruction (regardless of whether or not concurrent execution is prevented for the instruction due to tracing) without reporting the exception during the execution of that instruction. Then, when the main processor attempts to initiate the execution of the next FPCP instruction, a pre-instruction exception may be reported to initiate exception processing for an exception that occurred during a previous instruction. By using the FNOP instruction, the user can force any pending exceptions to be processed without performing any other operations.

Status Register: Not Affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. **Freescale** assemblers default to ID=1 for the FPCP.

NOTE

FNOP uses the same opcode as the FBcc.W <label> instruction, with cc=F (non-trapping false) and <label>=*+2 (which results in a displacement of 0).

Operation: IEEE Remainder of (FPn (÷) Source) ➧ FPn

Assembler Syntax:
FREM.<fmt> <ea>,FPn
FREM.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the modulo remainder of the number in the destination floating-point data register, using the source operand as the modulus. Stores the result in the destination floating-point data register, and stores the sign and seven least significant bits of the quotient in the FPSR quotient byte (the quotient is the result of FPn (÷) Source). The IEEE remainder function is defined as:

$$FPn - (Source \times N)$$

where:

N = INT(FPn (÷) Source) in the round-to-nearest mode

The FREM function is not defined for a source operand equal to zero or for a destination operand equal to infinity. Note that this function is not the same as the FMOD instruction, which uses the round-to-zero mode and thus returns a remainder that is different from the remainder required by the *IEEE Specification for Binary Floating-Point Arithmetic*.

Operation Table:

Destination \ Source		In Range	Zero	Infinity
		+ −	+ −	+ −
In Range	+ −	IEEE Remainder	NAN[1]	FPn[2]
Zero	+ −	+0.0 −0.0	NAN[1]	+0.0 −0.0
Infinity	+ −	NAN[1]	NAN[1]	NAN[1]

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. Returns the value of FPn before the operation. However, the result is processed by the normal instruction termination procedure to round it as required. Thus, an underflow and or inexact result may occur if the rounding precision has been changed to a smaller size since the FPn value was loaded.
3. If either operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Loaded with the sign and least significant seven bits of the quotient (FPn (÷) Source). The sign of the quotient is the exclusive OR of the sign bits of the source and destination operands.

Exception Byte:
- BSUN — Cleared
- SNAN — Refer to **4.5.4 NANs**.
- OPERR — Set if the source is zero, or the destination is infinity; cleared otherwise

OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: If in supervisor state
then FPCP State Frame ⧫ Internal State
else TRAP

Assembler Syntax: FRESTORE <ea>

Attributes: Unsized, privileged.

Description: Aborts the execution of any operation in progress, and loads a new internal state from the state frame located at the effective address. The first word at the specified address is the format word of the state frame, which specifies the size of the frame and the revision number of the FPCP that created it. The MPU writes the first word to the FPCP Restore CIR to initiate the restore operation, and then reads the response CIR to verify that the FPCP recognizes the format word as valid. If the format word is invalid for the FPCP (either because the size of the frame is not recognized, or the revision number does not match the revision of the processor), the MPU is instructed to take a format exception. The MPU then writes an abort to the control CIR, and the FPCP enters the IDLE state. If the format word is valid, the appropriate state frame is loaded, starting at the specified location and proceeding through higher addresses.

The FRESTORE instruction does not normally affect the programmer's model registers of the FPCP (except for the NULL state size, as described below), but is used only to restore the user invisible portion of the machine. The FRESTORE instruction is used with the FMOVEM instruction to perform a full context restoration of the FPCP, including the floating-point data registers and system control registers. In order to accomplish a complete restoration, the FMOVEM instructions are first executed to load the programmer's model, followed by the FRESTORE instruction to load the internal state and continue any previously suspended operation. Refer to **6.4 CONTEXT SWITCHING** for more information.

The current implementation of the FPCP supports three state frames. Refer to **6.4.2 State Frames** for the exact format of these state frames.

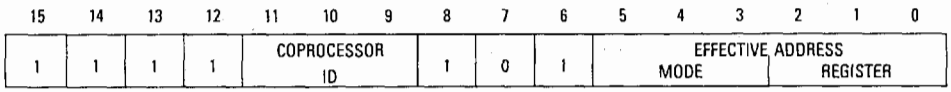
NULL: This state frame is four bytes long, with a format word of $0000. A FRESTORE operation with this size state frame is equivalent to a hardware reset of the FPCP. The programmer's model is set to the reset state, with nonsignaling NANs in the floating-point data registers and zeros in the FPCR, FPSR and FPIAR. (Thus, it is unnecessary to load the programmer's model before this operation.)

IDLE: This state frame is 28 ($1C) bytes long in the MC68881, and 60 ($3C) bytes long in the MC68882. An FRESTORE operation with this size state frame causes the FPCP to be restored to the idle state, waiting for the initiation of the next instruction. Exceptions that were pending before the execution of the previous FSAVE instruction are pending following the execution of the FRESTORE instruction. The programmer's model is not affected by loading this type of state frame.

BUSY: This state frame is 184 ($B8) bytes long in the MC68881 and 216 ($D8) bytes long in the MC68882. An FRESTORE operation with this size state frame causes the FPCP to be restored to the busy state, executing the instruction that was suspended by a previous FSAVE operation. The programmer's model is not affected by loading this type of state frame (although the completion of the suspended instruction after the restore is executed may modify the programmer's model).

Status Register: Cleared if the state size is NULL, otherwise not affected

Instruction Format:



Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for the state frame. Only postincrement or control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Save Internal State (Privileged Instruction)

Operation: If in supervisor state
then FPCP Internal State ⧫ State Frame
else TRAP

Assembler Syntax: FSAVE <ea>

Attributes: Unsized, privileged.

Description: Suspends the execution of any operation in progress, and saves the internal state in a state frame located at the effective address. After the save operation, the FPCP is in the idle state, waiting for the execution of the next instruction. The first word written to the state frame is the format word, which specifies the size of the frame and the revision number of the FPCP. The MPU initiates the FSAVE instruction by reading the FPCP save CIR, which is encoded with a format word that indicates the appropriate action to be taken by the main processor. The current implementation of the FPCP always returns one of five responses in the save CIR:

Value	Definition
$0018	Save NULL state frame
$0118	Not ready, come again
$0218	Illegal, take format exception
$XX18	Save IDLE state frame
$XXB4	Save BUSY state frame

where:
XX is the FPCP version number.

The Not Ready format word indicates that the FPCP is not prepared to perform a state save and that the MPU should process interrupts, if necessary, and then re-read the save CIR. The FPCP uses this format word to cause the main processor to wait while an internal operation is completed, if possible, in order to allow an IDLE frame to be saved rather than a BUSY frame. The Illegal format word is used to abort an FSAVE instruction that is attempted while the FPCP is executing a previous FSAVE instruction. All other format words cause the MPU to save the indicated state frame at the specified address. For state frame details see **6.4.2 State Frames**. These state frames are defined as follows:

NULL: This state frame is four bytes long. An FSAVE instruction that generates this size state frame indicates that the FPCP state has not been modified since the last hardware reset or FRESTORE instruction with a NULL state frame. This indicates that the programmer's model is in the reset state, with nonsignaling NANs in the floating-point data registers and zeros in the FPCR, FPSR, and FPIAR. (Thus, it is not necessary to save the programmer's model.)

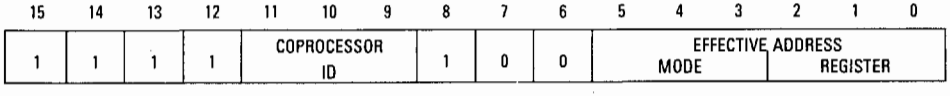
IDLE: This state frame is 28 ($1C) bytes long in the MC68881, and 60 ($3C) bytes long in the MC68882. An FSAVE instruction that generates this size state frame indicates that the FPCP was in an idle condition, waiting for the initiation of the next instruction. Any exceptions that were pending are saved in the frame and are then cleared internally. Thus, the pending exceptions are not reported until after a subsequent FRESTORE instruction loads the state frame. In addition to being used for context switching, this frame may be used by exception handler routines, since it contains the value of the operand that caused the last floating-point exception.

BUSY: This state frame is 184 ($B8) bytes long in the MC68881, and 216 ($D8) bytes long in the MC68882. An FSAVE instruction that generates this size state frame indicates that the FPCP was at a point within an instruction where it was necessary to save the entire internal state of the processor. This frame size is only used when absolutely necessary because of the large size of the frame and the amount of time required to transfer it. The action of the FPCP when this state frame is saved is the same as for the IDLE state frame.

The FSAVE does not save the programmer's model registers of the FPCP, but is used to save only the user invisible portion of the machine. The FSAVE instruction may be used with the FMOVEM instruction to perform a full context save of the FPCP including the floating-point data registers and system control registers. In order to accomplish a complete context save, an FSAVE instruction is first executed to suspend the current operation and save the internal state, followed by the appropriate FMOVEM instructions to store the programmer's model. Refer to **6.4 CONTEXT SWITCHING** for more information.

Status Register: Not affected

Instruction Format:



Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for the state frame. Only predecrement or control-alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Operation: FPn × INT(2^{Source}) ➧ FPn

Assembler Syntax:
FSCALE.<fmt> <ea>,FPn
FSCALE.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to an integer (if necessary) and adds that integer to the destination exponent. Stores the result in the destination floating-point data register. This function has the effect of multiplying the destination by 2^{Source}, but is much faster than a multiply operation when the source is an integer value.

The FPCP assumes that the scale factor is an integer value before the operation is executed. If not, the value is chopped (i.e., rounded using the round-to-zero mode) to an integer before it is added to the exponent. When the absolute value of the source operand is (≥) 2^{14}, an overflow or underflow always results.

Operation Table:

Destination \ Source		In Range	Zero	Infinity
		+ –	+ –	+ –
In Range	+ –	Scale Exponent	FPn[1]	NAN[2]
Zero	+ –	+0.0 −0.0	+0.0 −0.0	NAN[2]
Infinity	+ –	+inf −inf	+inf −inf	NAN[2]

NOTES:
1. Returns the value FPn before the operation. However, the result is processed by the normal instruction termination procedure to round it as required. Thus, an underflow and/or inexact result may occur if the rounding precision has been changed to a smaller size since the FPn value was loaded.
2. Sets the OPERR bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Set if the source operand is (+ or −)infinity; cleared otherwise
OVFL	Refer to **6.1.4 Overflow**.
UNFL	Refer to **6.1.5 Underflow**.

DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.
Source Specifier Field — Specifies the source register or data format.
If R/M = 0, specifies the source floating-point data register, FPm.
If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: If (condition true)
then 1s ⇨ Destination
else 0s ⇨ Destination

Assembler Syntax: FScc.<size> <ea>

Attributes: Size = (Byte)

Description: If the specified floating-point condition is true, sets the byte integer operand at the destination to TRUE (all ones), otherwise sets the byte to FALSE (all zeros). The conditional specifier cc may select any one of the 32 floating-point conditional tests as described in **4.4 CONDITIONAL TEST DEFINITIONS.**

Status Register:

Condition Codes:	Not affected	
Quotient Byte:	Not affected	
Exception Byte:	BSUN	Set if the NAN condition code is set and the condition selected is an IEEE nonaware test
	SNAN	Not Affected
	OPERR	Not Affected
	OVFL	Not Affected
	UNFL	Not Affected
	DZ	Not Affected
	INEX2	Not Affected
	INEX1	Not Affected

Accrued Exception Byte: The IOP bit is set if the BSUN bit is set in the exception byte. No other bit is affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	0	0	0	CONDITIONAL PREDICATE					

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Specifies the addressing mode for the byte integer operand:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Conditional Predicate Field — Specifies one of 32 conditional tests as defined in **4.4 CONDITIONAL TEST DEFINITIONS**.

NOTE

When a BSUN exception occurs, a pre-instruction exception is taken by the main processor. If the exception handler returns without modifying the image of the PC on the stack frame (to point to the instruction following the FScc), then it must clear the cause of the exception (by clearing the NAN bit or disabling the BSUN trap) or the exception occurs again immediately upon return to the routine that caused the exception.

Operation: FPn (÷) Source ➧ FPn

Assembler Syntax: FSGLDIV.<fmt> <ea>,FPn
FSGLDIV.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and divides that number into the number in the destination floating-point data register. Stores the result in the destination floating-point data register, rounded to single precision (regardless of the current rounding precision). This function is undefined for 0(÷)0 and infinity(÷)infinity.

Both the source and destination operands are assumed to be representable in the single precision format. If either operand requires more than 24 bits of mantissa to be accurately represented, the accuracy of the result is not guaranteed. Furthermore, the result exponent may exceed the range of single precision, regardless of the rounding precision selected in the FPCR mode control byte. Refer to **4.5.5.2 UNDERFLOW, ROUND, OVERFLOW** for more information.

Operation Table:

Destination \ Source		In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range	+	Divide (Single Precision)		+inf[1]	−inf[1]	+0.0	−0.0
	−			−inf[1]	+inf[1]	−0.0	+0.0
Zero	+	+0.0	−0.0	NAN[2]		+0.0	−0.0
	−	−0.0	+0.0			−0.0	+0.0
Infinity	+	+inf	−inf	+inf	−inf	NAN[2]	
	−	−inf	+inf	−inf	+inf		

NOTES:
1. Sets the DZ bit in the FPSR exception byte.
2. Sets the OPERR bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Set for 0(÷)0 or infinity(÷)infinity
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Set if the source is zero and the destination is in range; cleared otherwise
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: Source × FPn ➧ FPn

Assembler Syntax:
FSGLMUL.<fmt>
FSGLMUL.X
<ea>,FPn
FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and multiplies that number by the number in the destination floating-point data register. Stores the result in the destination floating-point data register, rounded to single precision (regardless of the current rounding precision).

Both the source and destination operands are assumed to be representable in the single precision format. If either operand requires more than 24 bits of mantissa to be accurately represented, the accuracy of the result is not guaranteed. Furthermore, the result exponent may exceed the range of single precision, regardless of the rounding precision selected in the FPCR mode control byte. Refer to **4.5.5.2 UNDERFLOW, ROUND, OVERFLOW** for more information.

Operation Table:

Destination \ Source		In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range	+	Multiply (Single Precision		+0.0	−0.0	+inf	−inf
	−			−0.0	+0.0	−inf	+inf
Zero	+	+0.0	−0.0	+0.0	−0.0	NAN[1]	
	−	−0.0	+0.0	−0.0	+0.0		
Infinity	+	+inf	−inf	NAN[1]		+inf	−inf
	−	−inf	+inf			−inf	+inf

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if one operand is zero and the other is infinity; cleared otherwise
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**

INEX1 If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	1	0	0	1	1	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: Sine of Source ➧ FPn

Assembler Syntax:

FSIN.<fmt>	<ea>,FPn
FSIN.X	FPm,FPn
FSIN.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the sine of that number. Stores the result in the destination floating-point data register. This function is not defined for source operands of (±)infinity. If the source operand is not in the range of $[-2\pi \ldots +2\pi]$, the argument is reduced to within that range before the sine is calculated. However, large arguments may lose accuracy during reduction, and very large arguments (greater than approximately 10^{20}) lose all accuracy. The result is in the range of $[-1 \ldots +1]$.

Operation Table:

Source / Destination	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Sine		+0.0	−0.0	NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. *If the source operand is a NAN, refer to* **4.5.4 NANs** *for more information.*

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is (+ or −)infinity; cleared otherwise.
OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	1	1	0

4

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, then the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Sine of Source ➧ FPs
Cosine of Source ➧ FPc

Assembler Syntax:
FSINCOS.<fmt> <ea>,FPc:FPs
FSINCOS.X FPm,FPc:FPs

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

4

Description: Converts the source operand to extended precision (if necessary) and calculates both the sine and the cosine of that number. Calculates both functions simultaneously; thus, this instruction is significantly faster than performing separate FSIN and FCOS instructions. Loads the sine result into the destination floating-point data register, FPs, and the cosine result into the destination floating-point data register FPc. Sets the condition code bits according to the sine result. If FPs and FPc are specified to be the same register, the cosine result is first loaded into the register and then is overwritten with the sine result. This function is not defined for source operands of (±)infinity.

If the source operand is not in the range of $(-2\pi \ldots +2\pi]$, the argument is reduced to within that range before the sine and cosine are calculated. However, large arguments may lose accuracy during reduction, and very large arguments (greater than approximately 10^{20}) lose all accuracy. The results are in the range of $[-1 \ldots +1]$.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
FPs	Sine		+0.0	−0.0	NAN[1]	
FPc	Cosine		+1.0		NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES** (for the sine result)

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source is (±)infinity; cleared otherwise
OVFL	Cleared
UNFL	Set if a sine underflow occurs, in which case the cosine result is 1. Cosine cannot underflow. Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**

INEX1 If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER, FPs			0	1	1	0	DESTINATION REGISTER, FPc		

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register, FPc Field — Specifies the destination floating-point data register, FPc. The cosine result is stored in this register.

Destination Register, FPs Field — Specifies the destination floating-point data register, FPs. The sine result is stored in this register. If FPc and FPs specify the same floating-point data register, the sine result is stored in the register, and the cosine result is discarded.

If R/M = 0 and the source register field is equal to either of the destination register fields, the input operand is taken from the specified floating-point data register, and the appropriate result is written into the same register.

Operation: Hyperbolic Sine of Source ➧ FPn

Assembler Syntax:

FSINH.<fmt>	<ea>,FPn
FSINH.X	FPm,FPn
FSINH.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the hyperbolic sine of that number. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Hyperbolic Sine		+0.0	−0.0	+inf	−inf

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Cleared
OVFL	Refer to **6.1.4 Overflow**.
UNFL	Refer to **6.1.5 Underflow**.
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result**.
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Square Root of Source ➧ FPn

Assembler Syntax:

FSQRT.<fmt>	<ea>,FPn
FSQRT.X	FPm,FPn
FSQRT.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the square root of that number. Stores the result in the destination floating-point data register. This function is not defined for negative operands.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	√x	NAN[1]	+0.0	−0.0	+inf	NAN[1]

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if the source operand is not zero and is negative; cleared otherwise
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	1	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: FPn − Source ➧ FPn

Assembler Syntax:
FSUB.<fmt> <ea>,FPn
FSUB.X FPm,FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and subtracts that number from the number in the destination floating-point data register. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source		In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range	+	Subtract		Subtract		−inf	+inf
	−						
Zero	+	Subtract		+0.0[1]	−0.0	−inf	+inf
	−			−0.0	+0.0[1]		
Infinity	+	+inf		+inf		NAN[2]	−inf
	−	−inf		−inf		−inf	NAN[2]

NOTES:
1. Returns +0.0 in rounding modes RN, RZ, and RP; returns −0.0 in RM.
2. Sets the OPERR bit in the FPSR exception byte.
3. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Set if both the source and destination are like-signed infinities; cleared otherwise
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	1	0	0	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

Operation: Tangent of Source ⧫ FPn

Assembler Syntax:

FTAN.<fmt>	<ea>,FPn
FTAN.X	FPm,FPn
FTAN.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the tangent of that number. Stores the result in the destination floating-point data register. This function is not defined for source operands of (±)infinity. If the source operand is not in the range of $[-\pi/2 \ldots +\pi/2]$, the argument is reduced to within that range before the tangent is calculated. However, large arguments may lose accuracy during reduction, and very large arguments (greater than approximately 10^{20}) lose all accuracy.

Operation Table:

Source / Destination	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Tangent		+0.0	−0.0	NAN[1]	

NOTES:
1. Sets the OPERR bit in the FPSR exception byte.
2. If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs**.
OPERR	Set if the source is (±)infinity; cleared otherwise
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	1	1	1

4

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: Hyperbolic Tangent of Source ➧ FPn

Assembler Syntax:

FTANH.<fmt>	<ea>,FPn
FTANH.X	FPm,FPn
FTANH.X	FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates the hyperbolic tangent of that number. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Hyperbolic Tangent		+0.0	−0.0	+1.0	−1.0

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Cleared
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	0	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: $10^{Source} \rightarrow FPn$

Assembler Syntax:
FTENTOX.<fmt> <ea>,FPn
FTENTOX.X FPm,FPn
FTENTOX.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates 10 to the power of that number. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	10^X		+1.0		+inf	+0.0

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	0	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M = 0, this field is unused, and should be all zeros.

If R/M = 1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M = 0, specifies the source floating-point data register, FPm.

If R/M = 1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M = 0 and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

Operation: If condition true, then TRAP

Assembler Syntax:
FTRAPcc
FTRAPcc.W #<data>
FTRAPcc.L #<data>

Attributes: Size = (Word, Long)

Description: If the selected condition is true, the main processor initiates exception processing. A vector number is generated to reference the TRAPcc exception vector. The stacked program counter points to the next instruction. If the selected condition is not true, no operation is performed, and execution continues with the next instruction in sequence. The immediate data operand is placed in the word(s) following the conditional predicate word and is available for user definition for use within the trap handler.

The conditional specifier cc selects one of the 32 conditional tests defined in **4.4 CONDITIONAL TEST DEFINITIONS.**

Status Register:

Condition Codes:	Not affected	
Quotient Byte:	Not affected	
Exception Byte:	BSUN	Set if the NAN condition code is set and the condition selected is an IEEE nonaware test
	SNAN	Not Affected
	OPERR	Not Affected
	OVFL	Not Affected
	UNFL	Not Affected
	DZ	Not Affected
	INEX2	Not Affected
	INEX1	Not Affected

Accrued Exception Byte: The IOP bit is set if the BSUN bit is set in the exception byte. No other bit is affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	1	1	1	MODE		
0	0	0	0	0	0	0	0	0	0	CONDITIONAL PREDICATE					
16-BIT OPERAND OR MOST SIGNIFICANT WORD OF 32-BIT OPERAND (IF NEEDED)															
LEAST SIGNIFICANT WORD OR 32-BIT OPERAND (IF NEEDED)															

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID = 1 for the FPCP.

Mode Field — Specifies the form of the instruction.

010 — The instruction is followed by a word operand.

011 — The instruction is followed by a long word operand.

100 — The instruction has no operand.

Conditional Predicate Field — Specifies one of 32 conditional tests as described in **4.4 CONDITIONAL TEST DEFINITIONS**.

Operand Field — Contains an optional word or long word operand that is user defined.

NOTE

When a BSUN exception occurs, a pre-instruction exception is taken by the main processor. If the exception handler returns without modifying the image of the PC on the stack frame (to point to the instruction following the FTRAPcc), it must clear the cause of the exception (by clearing the NAN bit or disabling the BSUN trap) or the exception occurs again immediately upon return to the routine that caused the exception.

Operation: Condition Codes for Operand ➧ FPCC

Assembler Syntax:

FTST.<fmt>	<ea>
FTST.X	FPm

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and sets the condition code bits according to the data type of the result.

Operation Table: The contents of this table differ from the other operation tables. A letter in an entry of this table indicates that the designated condition code bit is always set by the FTST operation. All unspecified condition code bits are cleared during the operation.

Source / Destination	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	none	N	Z	NZ	I	NI

NOTES:
1. If the source operand is a NAN, set the NAN condition code bit.
2. If the source operand is a SNAN, set the SNAN bit in the FPSR exception byte.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Cleared
UNFL	Cleared
DZ	Cleared
INEX2	Cleared
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	1	1	0	1	0

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.
If R/M=0, this field is unused, and should be all zeros.
If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.
0 — The operation is register to register.
1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.
If R/M=0, specifies the source floating-point data register, FPm.
If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

Destination Register Field — Since the FPCP uses a common command word format for all of the arithmetic instructions (including FTST), this field is treated in the same manner for FTST as for the other arithmetic instructions, even though the destination register is not modified. This field should be set to zero in order to maintain compatibility with future devices, although the FPCP does not signal an illegal instruction trap if it is not zero.

Operation: $2^{Source} \rightarrow FPn$

Assembler Syntax:
FTWOTOX.<fmt><ea>,FPn
FTWOTOX.X FPm,FPn
FTWOTOX.X FPn

Attributes: Format = (Byte, Word, Long, Single, Double, Extended, Packed)

Description: Converts the source operand to extended precision (if necessary) and calculates two to the power of that number. Stores the result in the destination floating-point data register.

Operation Table:

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	2^x		+1.0		+inf	+0.0

NOTE: If the source operand is a NAN, refer to **4.5.4 NANs** for more information.

Status Register:

Condition Codes: Affected as described in **4.5.5.1 SETTING FLOATING-POINT CONDITION CODES**

Quotient Byte: Not affected

Exception Byte:

BSUN	Cleared
SNAN	Refer to **4.5.4 NANs.**
OPERR	Cleared
OVFL	Refer to **6.1.4 Overflow.**
UNFL	Refer to **6.1.5 Underflow.**
DZ	Cleared
INEX2	Refer to **6.1.7 Inexact Result.**
INEX1	If <fmt> is Packed, refer to **6.1.8 Inexact Result on Decimal Input**; cleared otherwise.

Accrued Exception Byte: Affected as described in **6.1.10 IEEE Exception and Trap Compatibility**

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	0	0	1

Instruction Fields:

Coprocessor ID Field — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

Effective Address Field — Determines the addressing mode for external operands.

If R/M=0, this field is unused, and should be all zeros.

If R/M=1, this field is encoded with an M68000 addressing mode as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Only if <fmt> is Byte, Word, Long, or Single.

R/M Field — Specifies the source operand address mode.

0 — The operation is register to register.

1 — The operation is <ea> to register.

Source Specifier Field — Specifies the source register or data format.

If R/M=0, specifies the source floating-point data register, FPm.

If R/M=1, specifies the source data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	Word Integer
101	D	Double Precision Real
110	B	Byte Integer

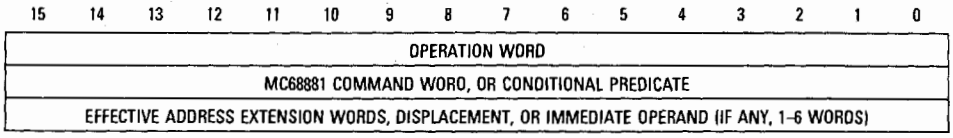
Destination Register Field — Specifies the destination floating-point data register, FPn.

If R/M=0 and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Freescale assemblers set the source and destination fields to the same value.

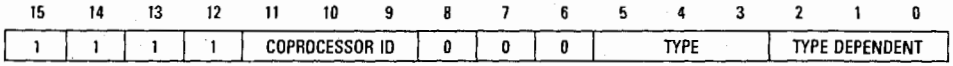
4.7 INSTRUCTION ENCODING DETAILS

The following paragraphs provide the details of the object code formats for the general, branch, set on condition, save, and restore type coprocessor instructions.

All FPCP instructions are from two to eight words in length as shown below (the longest case is for an immediate operand of six words — the X or P format).



All FPCP instructions begin with an operation word, formatted as follows:



Coprocessor ID — Specifies which coprocessor in the system is to execute this instruction. Freescale assemblers default to ID=1 for the FPCP.

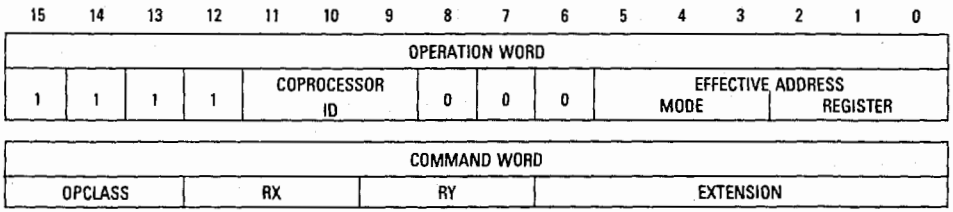
Type — Specifies the type of coprocessor instruction:

- 000 — General Instructions (Arithmetics, FMOVE, FMOVEM)
- 001 — FDBcc, FScc, FTRAPcc
- 010 — FBcc.W
- 011 — FBcc.L
- 100 — FSAVE
- 101 — FRESTORE
- 110 — (Undefined, Reserved)
- 111 — (Undefined, Reserved)

Type Dependent — Normally specifies the effective address or conditional predicate, but usage depends on the type field.

4.7.1 General Type Coprocessor Instruction Format

The general type coprocessor instruction format (shown below) is used for all FPCP arithmetic, move, move multiple, move constant, and transcendental instructions.



The interpretation of the command word fields, OPCLASS, RX, RY, and EXTENSION field varies with the instruction type and is summarized in Table 4-11.

Table 4-11. General Type Instruction Command Word Fields

Opclass	RX	RY	Instruction Class
000	Source, FPm	Destination, FPn	FPm to FPn. The extension field specifies the operation (move, add, etc.)
001	—	—	Undefined, reserved.
010	000-110 Source Data Format	Destination, FPn	Memory to Fpn. The extension field specifies the operation (move, add, etc.).
010	111	Destination, FPn	Move constant to FPn. The extension field contains the offset of the ROM constant.
011	Destination Data Format	Source, FPm	Move FPm to an external destination. If the destination format is packed decimal, the extension field specifies the k-factor (#k or Dn); otherwise it should be zero.
100	FPcr Select	000	Move single or multiple to the system control registers. The extension field should be zero.
101	FPcr Select	000	Move single or multiple system control registers to memory. The extension field should be zero.
110	Register list and addressing mode select.	00m	Move mutliple to the floating-point data registers. The least significant bit of the RY field and the extension field contains the register list, or the number of the main processor data register that contains the list.
111	Register list and addressing mode select.	00m	Move multiple from the floating-point data registers. The least significant bit of the RY field and the extension field contains the register list, or the number of the main processor data register that contains the list.

The FPCP general type instructions are classified into groups based upon instruction function and argument location (external or internal to the FPCP) as follows:

1. Floating-Point Register to Register
2. External Operand to Floating-Point Data Register
3. Move Constant to Floating-Point Data Register
4. Move Floating-Point Data Register to External Destination
5. Move System Control Register
6. Move Multiple Floating-Point Data Registers

Subdivision of the instruction set on this basis simplifies the specification of the MPU services required by each FPCP instruction. The FPCP requests services from the MPU via the coprocessor interface primitives described in **7.5 INSTRUCTION DIALOGS**.

If the command word indicates that an operand external to the FPCP is to be fetched or stored, the effective address field of the operation word is an MPU effective address descriptor. When the FPCP requests an external data access, the MPU evaluates the source/destination effective address based upon this effective address descriptor and transfers operand(s) to/from the FPCP.

If all operands are contained in FPCP floating-point data registers, the effective address field should be all zeros. If the effective address field is not all zeros, instruction execution proceeds normally; no F-line emulator exception trap is taken. However, to ensure compatibility with future devices, assembler and compiler programmers should fill this field with zeros when it is not used.

4.7.1.1 REGISTER-TO-REGISTER INSTRUCTIONS. This class of instructions includes floating-point data register to floating-point data register moves and the monadic, dyadic arithmetic, and transcendental instructions. For dyadic arithmetic instructions, the destination operand is replaced by the result.

FPm <op> FPn ➧ FPn

For monadic arithmetic instructions, the operand is the source FPm and the result is placed into the destination FPn. The source FPm and destination FPn can be the same floating-point data register.

FPm <op> ➧ FPn

The encoding format for this class of instructions is:

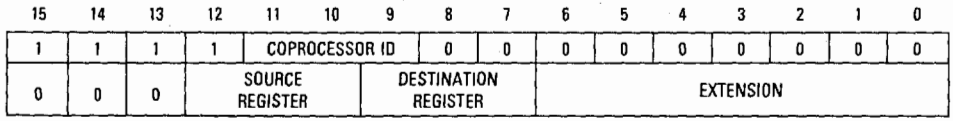


Table 4-12 shows the encoding of the source and destination register field.

Table 4-12. Register Field Encoding

000 — FP0	100 — FP4
001 — FP1	101 — FP5
010 — FP2	110 — FP6
011 — FP3	111 — FP7

The extension field indicates the operation to be performed. Table 4-13 lists the extension field encodings and functions. Also shown are the services requested of the MPU by the FPCP.

4.7.1.2 EXTERNAL OPERAND-TO-REGISTER INSTRUCTIONS. This class of instructions includes external operand to floating-point data register move and arithmetic instructions. External operands are located either in memory or an MPU data register (for B, W, L, or S data types). Data format conversion from one of the seven memory data formats to the extended data format is implicit in these instructions. For dyadic arithmetic instructions, the value in FPn is replaced by the result.

External Operand <op> FPn ➧ FPn

For monadic arithmetic instructions, the external operand is the source, and the result is placed in the destination FPn.

External Operand <op> ➧ FPn

The encoding format for this class of instructions is:

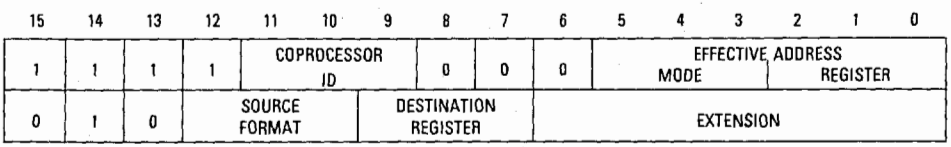


Table 4-13. Extension Field Encoding for Arithmetic Operations

Extension Field Services	Instruction Type	MPU
$00	FMOVE to FPn	Note 1
$01	FINT	Note 1
$02	FSINH	Note 1
$03	FINTRZ	Note 1
$04	FSQRT	Note 1
$06	FLOGNP1	Note 1
$08	FETOXM1	Note 1
$09	FTANH	Note 1
$0A	FATAN	Note 1
$0C	FASIN	Note 1
$0D	FATANH	Note 1
$0E	FSIN	Note 1
$0F	FTAN	Note 1
$10	FETOX	Note 1
$11	FTWOTOX	Note 1
$12	FTENTOX	Note 1
$14	FLOGN	Note 1
$15	FLOG10	Note 1
$16	FLOG2	Note 1
$18	FABS	Note 1
$19	FCOSH	Note 1
$1A	FNEG	Note 1
$1C	FACOS	Note 1
$1D	FCOS	Note 1
$1E	FGETEXP	Note 1
$1F	FGETMAN	Note 1
$20	FDIV	Note 1
$21	FMOD	Note 1
$22	FADD	Note 1
$23	FMUL	Note 1
$24	FSGLDIV	Note 1
$25	FREM	Note 1
$26	FSCALE	Note 1
$27	FSGLMUL	Note 1
$28	FSUB	Note 1
$30–$37	FSINCOS	Note 1
$38	FCMP	Note 1
$3A	FTST	Note 1
$40–$7F	(Undefined, Reserved)	Note 2

NOTES:
1. Two primitives can be issued for these operations. If the operation is register-to-register, the first primitive issued is null. If any exceptions, other than BSUN, are enabled, PC is set to one to request that the MPU pass the current program counter. If the operation is external operand-to-register, the first primitive is evaluate effective address and transfer data (with CA = 1, and PC = 1 if any exceptions other then BSUN are enabled). The second primitive is null (CA = 0) to terminate the instruction dialog.
2. The FPCP issues the take pre-instruction exception primitive with a vector number of 11 to instruct the MPU to take an F-line emulator trap.
3. Some extension field encodings are unspecified, are redundant with valid instructions implemented by the FPCP, and do not cause an F-line exception if executed. However, these encodings are reserved for future definition by Motorola, and thus should not be generated by assemblers or compilers. The redundant encodings are: $05, $07, $0B, $13, $17, $1B, $29–$2F, $39, and $3B–$3F.

The destination register is encoded as shown in Table 4-12.

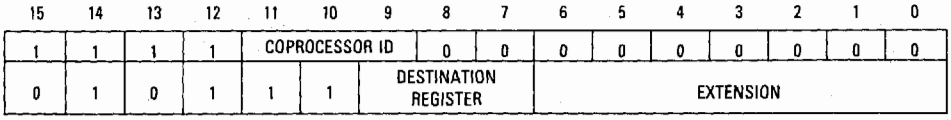
The source format field specifies the data format of the external operand. From the external operand are derived the length (in bytes) of the operand and the allowed effective addressing modes. The FPCP decodes the source format field as listed in Table 4-14. The extension field indicates the operation to be performed. Table 4-13 lists the extension field encodings and functions. Also listed are services requested of the MPU by the FPCP.

4.7.1.3 MOVE CONSTANT TO FLOATING-POINT DATA REGISTER INSTRUCTIONS. The FPCP constant ROM contains frequently used constants such as 0.0 and π. These instructions load a correctly rounded constant into a floating-point data register without an external data access.

Table 4-14. Source Format Field Encoding

Source Format Encoding	External Operand Data Format	Length in Bytes	Allowed <ea>
000	Long Word Integer	4	Data
001	Single Precision Real	4	Data
010	Extended Precision Real	12	Memory
011	Packed Decimal Real	12	Memory
100	Word Integer	2	Data
101	Double Precision Real	8	Memory
110	Byte Integer	1	Data

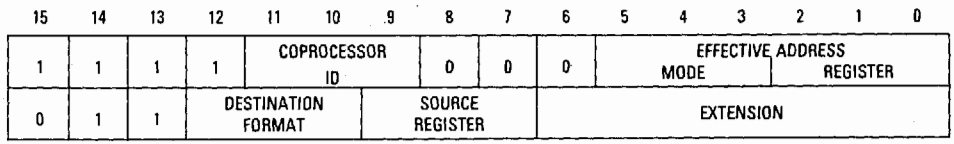
The encoding format for this class of instructions is:



The destination register field is encoded as shown in Table 4-12.

The extension field is used as an offset into the FPCP constant ROM. The FMOVECR instruction definition in **4.6 INDIVIDUAL INSTRUCTION DESCRIPTIONS** provides the valid extension field values for the FMOVECR instruction. The only service required by the FPCP from the MPU is the passing of the MPU PC to FPIAR if exceptions (other than BSUN) are enabled. This service is requested with the null (CA = 1, PC = 1) primitive.

4.7.1.4 MOVE TO EXTERNAL DESTINATION INSTRUCTIONS. External destinations are either memory or an MPU data register. Data format conversion from the extended data format to one of the seven memory data formats is implicit for these instructions. The encoding format for this class of instructions is:



The source register field is encoded as shown in Table 4-12.

The destination format field indicates the data format of the external destination. The MPU performs all transfers to an external destination at the request of the FPCP. When the FPCP

makes a request for a transfer to an external destination, the length (in bytes) of the operand, and the allowed effective addressing modes are specified in the primitive.

The FPCP decodes the destination format field to determine the length of the operand to be stored and the allowed effective addressing modes as listed in Table 4-15.

The extension field affects instruction execution only when the destination data format is packed decimal. A destination format encoding of 011 specifies a packed decimal string destination with the formatting parameter, k, in the extension field (encoded as a twos-complement value).

Table 4-15. Destination Format Field Encoding

Destination Format Encoding	External Operand Data Format	Length in Bytes	Allowed <ea>
000	Long Word Integer	4	Data Alterable
001	Single Precision Real	4	Data Alterable
010	Extended Precision Real	12	Memory Alterable
011	Packed Decimal Real with Static k-factor	12	Memory Alterable
100	Word Integer	2	Data Alterable
101	Double Precision Real	8	Memory Alterable
110	Byte Integer	1	Data Alterable
111	Packed Decimal Real	12	Memory Alterable with Dynamic k-factor

A destination format encoding of 111 indicates a packed decimal string destination with the formatting parameter, k, in an MPU data register. The extension field contains the number of the MPU data register that contains the k-factor. The MPU data register number is encoded in bits 6–4 of the extension field; bits 3–0 should be zero. The seven least significant bits of the MPU data register contain a twos-complement k-factor. The 25 most significant bits of the MPU data register are ignored. Table 4-16 lists the destination format field encodings, related extension field encodings, instruction operation, and the services requested of the MPU by the FPCP.

4.7.1.5 MOVE SYSTEM CONTROL REGISTER INSTRUCTIONS. This class of instructions includes the *move single system control register* instruction and the *move multiple system control registers* instruction. For the move single system control register instruction, external 32-bit operands may be immediate, in memory or an MPU register. For the move multiple system control register instructions, external operands may only be immediate or in memory (immediate addressing is only allowed if dr = 0). The encoding format for this class of instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
1	0	dr	REGISTER LIST			0	0	0	0	0	0	0	0	0	0

Table 4-16. Extension Field Encoding

Destination Format Encoding	Extension Encoding	External Operand Data Format	MPU Services
000	0000000	Long Word Integer	Notes 1 and 2
001	0000000	Single Precision Real	Notes 1 and 2
010	0000000	Extended Precision Real	Notes 1 and 2
011	kkkkkkk	Packed Decimal Real with a Static k-factor	Note 1
100	0000000	Word Integer	Notes 1 and 2
101	0000000	Double Precision Real	Notes 1 and 2
110	0000000	Byte Integer	Notes 1 and 2
111	rrr0000	Packed Decimal with a Dynamic k-factor	Note 3

NOTES:

1. Four service requests can be issued for this instruction type:
   a. Null (CA = 1, PC = x) can be first used to request the transfer of the PC to the FPIAR if exceptions are enabled.
   b. Null (CA = 1, IA = 1) is used to force the MPU to wait while the conversion takes place.
   c. Evaluate effective address and transfer data (CA = 1) is issued to request the transfer of the converted operand.
   d. Null (CA = 0) is used to terminate the dialog if no exception occurred. If an exception occurred, the take mid-instruction exception primitive is used to terminate the dialog.
2. The extension field should be all zeros, but no F-line emulator trap is taken if it is not. Assemblers and compilers should fill the extension field with zeros to ensure compatibility with future devices.
3. Bits 3–0 of the extension field should be zero, but no F-line emulator trap is taken if they are not. Assemblers and compilers should set these bits to zero to ensure compatibility with future devices. Four service requests are issued for this instruction:
   a. Transfer single main processor register (CA = 1, PC = x) is first used to request the transfer of the PC to the FPIAR (if exceptions are enabled) and to transfer the MPU data register containing the k-factor.
   b. Null (CA = 1, IA = 1) is used to force the MPU to wait while the conversion takes place.
   c. Evaluate effective address and transfer data (CA = 1) is issued to request the transfer of the converted operand.
   d. Null (CA = 0) is used to terminate the dialog if no exceptions occurred. If an exception occurred, the take mid-instruction exception primitive is used to terminate the dialog.

The dr bit set to one indicates a read of the FPCP; cleared to zero indicates a write to the FPCP. The register select field specifies the system control register or registers to be moved during the operation. Table 4-17 lists the dr and register list field encodings, instruction operation, operand size, allowed effective addressing modes, and services required of the MPU by the FPCP for this instruction type.

Bits 9–0 of the command word should be zero, although no F-line trap is taken if they are not. Assemblers and compilers should set these bits to zeros to ensure compatibility with future devices.

4.7.1.6 MOVE MULTIPLE FLOATING-POINT DATA REGISTERS INSTRUCTIONS. This class of instructions provides move multiple floating-point data register operations analogous to the M68000 move multiple address and data registers instructions. Unlike the integer counterpart, the floating-point register list can be specified either statically in the instruction or dynamically in an MPU data register.

The addressing modes for the move multiple from memory to floating-point data registers instruction are restricted to the control and address register indirect with postincrement effective addressing modes.

The addressing modes for the move multiple from floating-point data registers to memory instruction are restricted to the control alterable and address register indirect with predecrement effective addressing modes.

Table 4-17. Encoding for Move FPcr Operations

Register List	Instruction Operation	Transfer Size (in Bytes)	Allowed <ea>	MC68020/ MC68030 Services
Move Memory to Registers (dr=0)				
000	(Undefined, Reserved)	—	—	Notes 1 and 2
001	Move to FPIAR	4	Any	Note 1
010	Move to FPSR	4	Data	Note 1
011	Move to FPSR and FPIAR	8	Memory	Note 1
100	Move to FPCR	4	Data	Note 1
101	Move to FPCR and FPIAR	8	Memory	Note 1
110	Move to FPCR and FPSR	8	Memory	Note 1
111	Move to FPCR, FPSR, and FPIAR	12	Memory	Note 1
Move Registers to Memory (dr=1)				
000	(Undefined, Reserved)	—	—	Notes 1 and 2
001	Move from FPIAR	4	Alterable	Note 1
010	Move from FPSR	4	Data Alterable 1	
011	Move from FPSR and FPIAR	8	Memory Alterable 1	
100	Move from FPCR	4	Data Alterable 1	
101	Move from FPCR and FPIAR	8	Memory Alterable 1	
110	Move from FPCR and FPSR	8	Memory Alterable 1	
111	Move from FPCR, FPSR, and FPIAR	12	Memory Alterable 1	

NOTES:
1. This operation requires two primitives to be issued to the MPU. The first primitive is evaluate effective address and transfer data (CA=1), indicating the appropriate transfer size and allowed effective addressing mode. The second primitive is null (CA=0) to terminate the instruction dialog.
2. For the current implementation of the FPCP, this encoding is redundant with the 001 encoding of the register select field (i.e., it selects the FPIAR as the only register to be moved); however, this encoding is reserved for future use by Freescale.

NOTE

The effective addressing mode restrictions for this instruction are enforced by the MPU when the transfer multiple coprocessor registers response primitive is received (not by the FPCP when it receives the command word). If the encoding of the effective address field in the operation word is inconsistent with the encoding of the dr and mode fields in the command word, unexpected results occur. In some cases, the instruction is executed, but the order of the register transfer is the reverse of the appropriate order for the addressing mode. However, system integrity is preserved for all cases.

The encoding format for this class of instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
1	1	dr	MODE			0	0	0	REGISTER LIST						

The dr bit set to one indicates a read of the FPCP; dr cleared to zero indicates a write to the FPCP. The mode field specifies the order of the register transfer and the location of the register list. The definitions of the mode field bits are (bits shown as X may be either zero or one):

0X Transfer FP7 through FP0
1X Transfer FP0 through FP7
X0 Register List is Static
X1 Register List is Dynamic

The order of the register transfer that is selected affects the interpretation of the register list, because the list is always scanned starting with the most significant bit. Thus, for the 0X encoding of the mode field, the most significant bit of the register list corresponds to FP7, and the least significant bit corresponds to FP0. For the 1X encoding, this relationship is reversed.

The type of the register list also affects the interpretation of the register list field. If a static list is selected, then the register list field of the command word contains the register list. If a dynamic register list is selected, then the register list field of the command word contains the number of the MPU data register that contains the register list. The format of the register list field in the command word for the various mode field encodings is shown in the following table. If a bit in the register list is set, then the corresponding register is moved, otherwise the list is scanned for the next bit that is set. For the dynamic register list format, rrr specifies the MPU data register that contains the register list (X means either zero or one). The format of a dynamic register list is the same as the format of the appropriate static list, and it is contained in the least significant eight bits of the MPU data register.

Mode Field Encoding		Register List Field Format							
00	—	FP7	FP6	FP5	FP4	FP3	FP2	FP1	FP0
10	—	FP0	FP1	FP2	FP3	FP4	FP5	FP6	FP7
X1	—	0	r	r	r	0	0	0	0

Table 4-18 lists the dr and mode field encodings, instruction operation, allowed effective addressing modes, and services required of the MPU by the FPCP for this instruction type.

4.7.1.7 UNDEFINED, RESERVED COMMAND WORDS. The command word encoding shown below is undefined and reserved for future Freescale use. All undefined, reserved command word encodings generate an F-line emulator exception.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	x	x	x	x	x	x
0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x

4.7.2 FDBcc, FScc, and FTRAPcc Instruction Formats

For these instruction types, the MPU writes a conditional predicate to the FPCP condition CIR for evaluation. The FPCP determines whether the conditional predicate is true or false

Table 4-18. Encodings for Move Multiple FPn Operations

Mode Field	Instruction Operation	Allowed <ea> Modes	MPU Services
	Move Memory to Registers (dr=0)		
00	(Invalid Operation)	—	Note 1
01	(Invalid Operation)	—	Note 1
10	Move to Registers, Static Register List	(An)+ or Control	Note 2
11	Move to Registers, Dynamic Register List	(An)+ or Control	Note 3
	Move Registers to Memory (dr=1)		
00	Move from Registers, Static Register List	−(An)	Note 2
01	Move from Registers, Dyamic Register List	−(An)	Note 3
10	Move from Registers, Static Register List	Control Alterable	Note 2
11	Move from Registers, Dynamic Register List	Control Alterable	Note 3

NOTES:
1. These encodings cause the FPCP to perform an operation that is inconsistent with the M68000 Family move multiple operations. For these cases, the selected registers are transferred in the order that is appropriate for the predecrement addressing mode (i.e., FP7–FP0) using a static or dynamic register list, respectively. However, the MPU does not allow the predecrement addressing mode for a move from memory to multiple coprocessor registers operation. Thus, assemblers and compilers should not generate these encodings, or unexpected results may occur.
2. This instruction requires two primitives; the first is the transfer multiple coprocessor registers (CA=1) primitive to request that the MPU evaluate the effective address, read the register select CIR, and transfer the number of registers indicated by the mask (with an operand size of 12 bytes for each register). The second primitive is null (CA=0), which is used to terminate the dialog.
3. This instruction requires three primitives; the first is the transfer single main processor register (CA=1) primitive to request the transfer of the MPU data register that contains the dynamic register list. The second is the transfer multiple coprocessor registers (CA=1) primitive to request that the MPU evaluate the effective address, read the register select CIR, and transfer the number of registers indicated by the mask (with an operand size of 12 bytes for each register). The third primitive is null (CA=0) to terminate the dialog.

based on the floating-point condition codes as described in **4.4 CONDITIONAL TEST DEFINITIONS**. The true or false result is returned to the main processor with the null primitive.

These instructions all use the operation word type field encoding and command word format shown below. The instruction specific field of the operation word determines the instruction variation and is defined in Table 4-19 for each instruction type.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	INSTRUCTION SPECIFIC					
0	0	0	0	0	0	0	0	0	0	–	CONDITIONAL PREDICATE				

The conditional predicate field specifies the conditional test to be performed. Table 4-20 lists the conditional predicate encodings and the FPCP responses. For details of the calculation of the response, refer to **4.4 CONDITIONAL TEST DEFINITIONS**. Bits 15–6 of the command word are shown to be filled with zeros; however, no F-line trap is taken if they are not. To ensure compatibility with future devices, assemblers and compilers should fill this field with zeros.

The displacement, extension, or operand words follow immediately after the conditional predicate word. For the FDBcc instruction, the displacement is a 16-bit twos complement integer that indicates the relative distance in bytes from the displacement word (i.e., the

Table 4-19. Encodings for the FDBcc, FScc, and FTRAPcc Instructions

Instruction Specific Field	Instruction Operation	Selected <ea>	MPU Services
000 XXX	FScc <ea>	Dn	Note 1
001 XXX	FDBcc Dn,<label>	—	Note 2
010 XXX	FScc <ea>	(An)	Note 1
011 XXX	FScc <ea>	(An)+	Note 1
100 XXX	FScc <ea>	−(An)	Note 1
101 XXX	FScc <ea>	d_{16}(An)	Note 1
110 XXX	FScc <ea>	indexed/indirect	Note 1
111 000	(Undefined, Reserved)	—	Note 3
111 001	(Undefined, Reserved)	—	Note 3
111 010	FTRAPcc.W #<data>	—	Note 4
111 011	FTRAPcc.L #<data>	—	Note 4
111 100	FTRAPcc with No Parameter	—	Note 4
111 101	(Undefined, Reserved)	—	Note 3
111 110	(Undefined, Reserved)	—	Note 3
111 111	(Undefined, Reserved)	—	Note 3

NOTES:
1. The MPU evaluates the <ea> and writes the conditional predicate to the FPCP for evaluation. The null (CA = 0) primitive is used to return the true/false evaluation, and the appropriate value is then written to the <ea> by the MPU. The value of XXX specifies the MPU data or address register (Dn or An) used in the <ea> evaluation.
2. The MPU writes the conditional predicate to the FPCP for evaluation. The null (CA = 0) primitive is used to return the true/false evaluation. If the condition is true, the MPU proceeds to the next instruction. Otherwise, the counter register Dn.W (specified by the value of XXX) is decremented, and the new value is compared with −1. If it is equal to −1, the MPU proceeds to the next instruction; otherwise, the 16-bit displacement is sign extended and added to the PC.
3. The MPU takes an F-line emulation trap.
4. The MPU writes the conditional predicate to the FPCP for evaluation. The null (CA = 0) primitive is used to return the true/false evaluation. If the condition is true, then the cpTRAPcc exception is taken. Otherwise, the MPU proceeds to the next instruction, discarding the optional immediate operand if necessary.

PC value used in the branch destination calculation is the address of the displacement word). For the FScc instruction, the effective address extension words are formatted as required by the main processor. For the FTRAPcc instruction, a one or two word user-defined operand can be included with the instruction. Note that from the perspective of the FPCP, these instructions are identical to the branch type coprocessor instructions. The various operations are handled by the MPU in a manner that is transparent to the FPCP.

4.7.3 Conditional Branch Instruction Format

For this instruction type, the MPU writes a conditional predicate to the FPCP condition CIR for evaluation. The FPCP determines whether the conditional predicate is true or false based on the floating-point condition codes as described in **4.4 CONDITIONAL TEST DEFINITIONS**. The true or false result is returned to the main processor with the null primitive. The formats for this instruction type are shown below.

Table 4-20. Conditional Predicate Evaluation Responses

Conditional Predicate	Mnemonic	Definition	MC68881/ MC68882 Response
000000	F	False	Note 1
000001	EQ	Equal	Note 1
000010	OGT	Ordered Greater Than	Note 1
000011	OGE	Ordered Greater Than or Equal	Note 1
000100	OLT	Ordered Less Than	Note 1
000101	OLE	Ordered Less Than or Equal	Note 1
000110	OGL	Ordered Greater Than or Less Than	Note 1
000111	OR	Ordered	Note 1
001000	UN	Unordered	Note 1
001001	UEQ	Unordered or Equal	Note 1
001010	UGT	Unordered or Greater Than	Note 1
001011	UGE	Unordered or Greater Than or Equal	Note 1
001100	ULT	Unordered or Less Than	Note 1
001101	ULE	Unordered or Less Than or Equal	Note 1
001110	NE	Not Equal	Note 1
001111	T	True	Note 1
010000	SF	Signaling False	Note 2
010001	SEQ	Signaling Equal	Note 2
010010	GT	Greater Than	Note 2
010011	GE	Greater Than or Equal	Note 2
010100	LT	Less Than	Note 2
010101	LE	Less Than or Equal	Note 2
010110	GL	Greater Than or Less Than	Note 2
010111	GLE	Greater Than or Less Than or Equal	Note 2
011000	NGLE	Not (Greater Than or Less Than or Equal)	Note 2
011001	NGL	Not (Greater Than or Less Than)	Note 2
011010	NLE	Not (Less Than or Equal)	Note 2
011011	NLT	Not (Less Than)	Note 2
011100	NGE	Not (Greater Than or Equal)	Note 2
011101	NGT	Not (Greater Than)	Note 2
011110	SNE	Signaling Not Equal	Note 2
011111	ST	Signaling True	Note 2
1XXXXX	—	(Undefined, Reserved)	Note 3

NOTES:
1. Indicate the condition true or false result by using the null (CA=0) primitive.
2. If the NAN condition code bit is set, then set the BSUN bit in the FPSR. If the BSUN trap is enabled, then return the take pre-instruction exception primitive with the BSUN vector number; otherwise, indicate the condition true false result by using the null (CA=0) primitive.
3. Not used, redundant encodings with 0XXXXX. No F-line trap is taken if these bit patterns are used. To ensure compatibility with future devices, assemblers and compilers should use the 0XXXXX encodings.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	1	0	CONDITIONAL PREDICATE					
16-BIT DISPLACEMENT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	1	1	CONDITIONAL PREDICATE					
32-BIT DISPLACEMENT															

The conditional predicate field specifies the conditional test to be performed. Table 4-20 lists the conditional predicate encodings and the FPCP responses. For details of the response calculation, refer to **4.4 CONDITIONAL TEST DEFINITIONS**.

The displacement is a twos-complement integer that indicates the relative distance in bytes from the displacement word(s) (i.e., the PC value used in the branch destination calculation is the address of the displacement word(s)). A 16-bit displacement is sign extended before it is used in the branch destination calculation.

NOTE

From the perspective of the FPCP, the two forms of this instruction are identical. The size of the displacement is determined by the MPU and is transparent to the FPCP. Also, the FNOP instruction syntax that is recognized by Freescale assemblers generates an FBcc.W instruction with cc = F (false) and a displacement value of zero.

4.7.4 Save Instruction Format

The FSAVE instruction indicates that the FPCP must immediately suspend any current operation and save the internal state in memory. Effective addressing modes are restricted to control alterable and address register indirect with predecrement modes. The encoding format for this instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			1	0	0	EFFECTIVE ADDRESS MODE			REGISTER		

4.7.5 Restore Instruction Format

The FPCP restore instruction indicates that regardless of the current state of operation, a new internal state is to be loaded immediately. Effective addressing modes are restricted to control and address register indirect with postincrement modes. The encoding format for this instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			1	0	1	EFFECTIVE ADDRESS MODE			REGISTER		

4.8 INSTRUCTION FORMAT SUMMARY

The following paragraphs present a summary of the binary encodings for the FPCP instruction set. The unique encoding for each instruction is shown explicitly, with the encoded fields common to all of the instructions listed in a single table at the beginning of this section.

4.8.1 Coprocessor ID Field

This field of each instruction specifies which one of eight (seven, for the MC68030) possible coprocessors in a system is to perform the operation. There are no restrictions placed on the value of the ID field by the main processor in the system; however, certain conventions should be followed. Freescale assemblers default to coprocessor ID = 1 for the FPCP. although directives are available to change this default. Furthermore, due to the hardware implementation of the MC68851 Paged Memory Management Unit, that device must be assigned to coprocessor ID = 0 if used in a system. Thus, the FPCP should not be assigned to coprocessor ID = 0 if it is anticipated that an MC68851 may be used in the system, or in an MC68030 system.

4.8.2 Effective Address Field

This field specifies the M68000 Family addressing mode that is to be used to locate operands external to the FPCP {if required by the instruction). For some operations, restrictions are placed on which of the available addressing modes are allowed. These restrictions are enforced by hardware in the MPU and FPCP, and Freescale assemblers do not generate operation words with disallowed effective addressing mode field encodings. The encodings for this fields are shown in Table 4-21.

4.8.3 Register/Memory Field

This field is common to all of the arithmetic instructions and the FMOVE to FPn instruction. A zero in this field indicates that the operation is register-to-register, and a one in this field indicates that the source operand is external to the FPCP.

4.8.4 Source Specifier Field

This field is common to all of the arithmetic instructions and the FMOVE floating-point data register instruction. The definition of this field is affected by the value of the R M field:

If R/M=0, it specifies the source floating-point data register, FPm.
If R/M=1, it specifies the source operand data format:

000	L	Long Word Integer
001	S	Single Precision Real
010	X	Extended Precision Real
011	P	Packed Decimal Real
100	W	WordInteger
101	D	Double Precision Real
110	B	Byte Integer

Table 4-21. Effective Address Field Encoding Summary

Address Modes	Mode	Register	Data	Memory	Control	Alterable	Assembler Syntax
Data Register Direct	000	reg. no.	X	—	—	X	Dn
Address Register Direct	001	reg. no.	—	—	—	X	An
Address Register Indirect	010	reg. no.	X	X	X	X	(An)
Address Register Indirect with Postincrement	011	reg. no.	X	X	—	X	(An)+
Address Register Indirect with Predecrement	100	reg. no.	X	X	—	X	−(An)
Address Register Indirect with Displacement	101	reg. no.	X	X	X	X	(d_{16},An)
Address Register Indirect with Index (8-Bit Displacement)	110	reg. no.	X	X	X	X	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	110	reg. no.	X	X	X	X	(bd,An,Xn)
Memory Indirect Postindexed	110	reg. no.	X	X	X	X	([bd,An],Xn,od)
Memory Indirect Preindexed	110	reg. no.	X	X	X	X	([bd,An,Xn],od)
Absolute Short	111	000	X	X	X	X	(xxx).W
Absolute Long	111	001	X	X	X	X	(xxx).L
Program Counter Indirect with Displacement	111	010	X	X	X	—	(d_{16},PC)
Program Counter Indirect with Index (8-Bit Displacement)	111	011	X	X	X	—	(d_8,PC,Xn)
Program Counter Indirect with Index (Base Displacement)	111	011	X	X	X	—	(bd,PC,Xn)
PC Memory Indirect Postindexed	111	011	X	X	X	—	([bd,PC],Xn,od)
PC Memory Indirect Preindexed	111	011	X	X	X	—	([bd,PC,Xn],od)
Immediate	111	100	X	X	—	—	#<data>

4.8.5 Destination Register Field

This field is common to all of the arithmetic instructions and the FMOVE to FPn instruction. This field specifies the floating-point data register that is to be used as the destination. The result of an operation is always stored in this register, and one of the source operands is fetched from this register for dyadic instructions.

4.8.6 Conditional Predicate Field

This field is common to all of the conditional instructions and specifies the FPCP conditional test that is to be evaluated for the main processor. Table 4-22 shows the conditional predicate binary encodings for the 32 conditional tests supported by the FPCP.

Table 4-22. Conditional Predicate Field Encoding Summary

Conditional Predicate	Mnemonic	Definition
000000	F	False
000001	EQ	Equal
000010	OGT	Ordered Greater Than
000011	OGE	Ordered Greater Than or Equal
000100	OLT	Ordered Less Than
000101	OLE	Ordered Less Than or Equal
000110	OGL	Ordered Greater Than or Less Than
000111	OR	Ordered
001000	UN	Unordered
001001	UEQ	Unordered or Equal
001010	UGT	Unordered or Greater Than
001011	UGE	Unordered or Greater Than or Equal
001100	ULT	Unordered or Less Than
001101	ULE	Unordered or Less Than or Equal
001110	NE	Not Equal
001111	T	True
010000	SF	Signaling False
010001	SEQ	Signaling Equal
010010	GT	Greater Than
010011	GE	Greater Than or Equal
010100	LT	Less Than
010101	LE	Less Than or Equal
010110	GL	Greater Than or Less Than
010111	GLE	Greater Than or Less Than or Equal
011000	NGLE	Not (Greater Than or Less Than or Equal)
011001	NGL	Not (Greater Than or Less Than)
011010	NLE	Not (Less Than or Equal)
011011	NLT	Not (Less Than)
011100	NGE	Not (Greater Than or Equal)
011101	NGT	Not (Greater Than)
011110	SNE	Signaling Not Equal
011111	ST	Signaling True

4.9 INSTRUCTION FORMAT DIAGRAMS

The instruction formats are summarized in this section.

FMOVE to FPn

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	0	0

FINT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	0	1

FSINH

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	1	0

FINTRZ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	0	1	1

FSQRT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	1	0	0

FLOGNP1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	0	1	1	0

FETOXM1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	0	0	0

FTANH

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	0	0	1

FATAN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	0	1	0

FASIN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	1	0	0

FATANH

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	1	0	1

FSIN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER				DESTINATION REGISTER			0	0	0	1	1	1	0

FTAN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	0	1	1	1	1

FETOX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			OESTINATION REGISTER			0	0	1	0	0	0	0

FTWOTOX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	0	0	1

FTENTOX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	0	1	0

FLOGN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	1	0	0

FLOG10

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	1	0	1

FLOG2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	0	1	1	0

FABS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	0	0	0

FCOSH

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	0	0	1

FNEG

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	0	1	0

FACOS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	0	0

FCOS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	0	1

FGETEXP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	1	0

FGETMAN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	1	1

FDIV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	0	0

FMOD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	0	1

FADD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	1	0

FMUL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	0	1	1

FSGLDIV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	0	0

FREM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	0	1

FSCALE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	1	0

FSGLMUL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	0	1	1	1

FSUB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	0	1	0	0	0

FSINCOS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER, FPs			0	1	1	0	DESTINATION REGISTER, FPc		

FCMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	1	1	0	0	0

FTST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	1	1	1	0	1	0

FMOVECR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	0	0	0	0	0	0
0	1	0	1	1	1	DESTINATION REGISTER			ROM OFFSET						

ROM Offset Field — Specifies the offset in the FPCP Constant ROM where the desired constant is located.

FMOVE from FPn

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	1	1	DESTINATION FORMAT			SOURCE REGISTER			K-FACTOR (IF REQUIRED)						

Destination Format Field — Specifies the data format of the destination operand as follows:

000 — Long Word Integer
001 — Single Precision Real
010 — Extended Precision Real
011 — Packed Decimal Real, static k-factor
100 — Word Integer
101 — Double Precision Real
110 — Byte Integer
111 — Packed Decimal Real, dynamic k-factor

k-factor Field — Specifies the format of the packed decimal string to be generated (if the destination format field indicates packed decimal), or the number of the main processor data register that contains the format specification. The interpretation of the k-factor is:

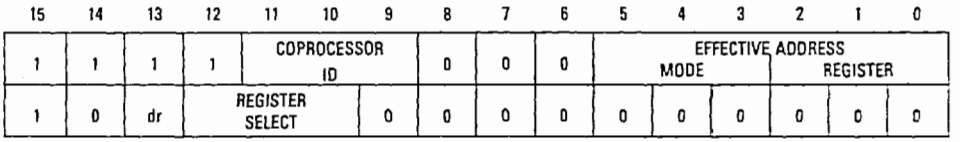
−64 to 0 — Number of significant digits to the right of the decimal point.
+1 to +17 — Number of significant digits in the mantissa.
+18 to +63 — Sets the OPERR bit, treated as +17.

The format of this field for a dynamic k-factor is:

r r r 0 0 0 0

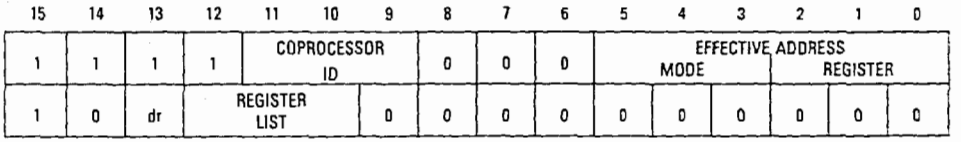
Where rrr is the number of the main processor data register that contains the k-factor.

FMOVE FPcr



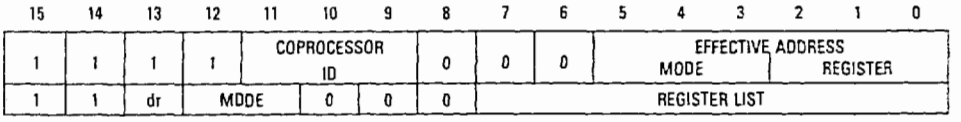
dr Field — Specifies the direction of the transfer:
0 — Move memory to system control register
1 — Move system control register to memory
Register Select Field — Specifies the system control register to be moved:
001 — FPIAR
010 — FPSR
100 — FPCR

FMOVEM FPcr



dr Field — Specifies the direction of the transfer:
0 — Move memory to system control registers
1 — Move system control registers to memory
Register List Field — Specifies the system control registers to be moved:
000 — (Undefined, reserved)
001 — FPIAR
010 — FPSR
011 — FPSR, then FPIAR
100 — FPCR
101 — FPCR, then FPIAR
110 — FPCR, then FPSR
111 — FPCR, then FPSR, then FPIAR

FMOVEM FPn



dr Field — Specifies the direction of the transfer
0 — Move the listed registers from memory to the FPCP
1 — Move the listed registers from the to memory
Mode Field — Specifies the type of the register list and addressing mode
00 — Static register list, predecrement addressing mode
01 — Dynamic register list, predecrement addressing mode
10 — Static register list, postincrement *or* control addressing mode
11 — Dynamic register list, postincrement or control addressing mode
Register List Field:
Static list — contains the select mask; if a register is to be moved, the corresponding bit in the list is set, otherwise it is clear.

Dynamic list — contains the main processor data register number, rrr, as shown below:

		Register List Format							
Static, −(An)	—	FP7	FP6	FP5	FP4	FP3	FP2	FP1	FP0
Static, (An)+ or Control	—	FP0	FP1	FP2	FP3	FP4	FP5	FP6	FP7
Dynamic	—	0	r	r	r	0	0	0	0

The format of the dynamic list mask is the same as for the static list and is contained in the least significant eight bits of the specified MPU data register.

FScc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	0	0	0	CONDITIONAL PREDICATE					

FDBcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	0	0	1	COUNT REGISTER		
0	0	0	0	0	0	0	0	0	0	CONDITIONAL PREDICATE					
16-BIT DISPLACEMENT															

Count Register Field — Specifies the main processor data register to be decremented

FTRAPcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	1	1	1	1	MODE		
0	0	0	0	0	0	0	0	0	0	CONDITIONAL PREDICATE					
16-BIT OPERAND OR MOST SIGNIFICANT WORD OF 32-BIT OPERAND (IF NEEDED)															
LEAST SIGNIFICANT WORD OF 32-BIT OPERAND (IF NEEDED)															

Mode Field — Specifies the form of the instruction:
010 — The instruction is followed by a 16-bit operand.
011 — The instruction is followed by a 32-bit operand.
100 — The instruction has no operand following it.

FNOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FBcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	1	SIZE	CONDITIONAL PREDICATE					
16-BIT DISPLACEMENT, OR MOST SIGNIFICANT WORD OF 32-BIT DISPLACEMENT															
LEAST SIGNIFICANT WORD OF 32-BIT DISPLACEMENT (IF NEEDED)															

Size Field — Specifies the size of the twos-complement displacement:
Size = 0 — Displacement is 16-bits (and is sign extended before it is used).
Size = 1 — Displacement is 32-bits.

FSAVE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			1	0	0	EFFECTIVE ADDRESS MODE			REGISTER		

FRESTORE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			1	0	1	EFFECTIVE ADDRESS MODE			REGISTER		

SECTION 5
COPROCESSOR PROGRAMMING

This section describes the guidelines for programming Freescale's floating-point coprocessors. The first portion of the section presents the guidelines for applications programming. It describes the concurrency with main processor instruction execution applicable to both coprocessors, and the coprocessor instruction concurrency provided by the MC68882 coprocessor. It also discusses the optimization of code for the MC68882 coprocessor.

The second portion of the section discusses systems programming considerations. It describes the state frame sizes, and lists the instructions required in the exception handlers for MC68881/MC68882 (FPCP) exceptions. The systems programming portion also describes the handling of exceptions by the MC68020/MC68030 (MPU) and FPCP combination, and code that detects and identifies the coprocessor.

This section primarily describes programming of the MC68882 coprocessor, since programs that run successfully in the MC68882 also run successfully in the MC68881. It is advisable to program for the MC68882 even if the MC68881 is currently being used, so that no program changes are required for upgrading to the MC68882.

5.1 APPLICATIONS PROGRAMMING

All applications programs that run successfully on the MC68881 can be used on the MC68882 without alteration, but optimization of code for the MC68882 provides significant reduction in execution time. The following paragraphs describe the concurrency available with the MC68881, the greater concurrency provided by the MC68882, and optimization techniques for MC68882 programs.

5.1.1 Concurrency

The M68000 coprocessor interface, the MC68020 and MC68030 microprocessors, and the MC68881 and MC68882 coprocessors are designed to provide the conventional sequential instruction execution models while instructions may actually be executed concurrently. Applications programs can be written with no provisions for concurrency; the system apparently executes the instructions in sequence. This apparent sequential execution is automatic, and the programmer need not be concerned about it.

5.1.1.1 CONCURRENT INTEGER AND FLOATING-POINT COMPUTATIONS. The M68000 coprocessor interface is designed to provide full support for the sequential instruction execution model. Although the M68000 coprocessor interface allows concurrency between coprocessor and main processor operations, the coprocessor must implement this concurrency while maintaining a programming model based on sequential instruction execution.

After the main processor initiates a floating-point instruction (by writing to the command CIR), it reads the response CIR. When the CA bit (bit [15] of the response CIR) is set, it

indicates that the main processor should perform the specified service and then read the response CIR again. The FPCP sets the CA bit to define portions of the floating-point instruction that cannot operate concurrently with main processor instruction execution. When the coprocessor can operate concurrently with main processor instructions, it clears the CA bit in the response primitive. Clearing the CA allows the main processor to proceed to the next instruction after it has read the response CIR and performed the specified service. This releases the main processor for concurrent operation. In the arithmetic floating-point instructions, the FPCP releases the main processor after the transfer phase is completed. Refer to **8.2 CONCURRENT INSTRUCTION EXECUTION** for further details of main processor/coprocessor concurrent instruction execution.

Within the boundaries of a floating-point instruction that does not allow concurrency with main processor instructions (response primitives return CA=1), the FPCP can allow the main processor to service pending interrupts. Bit [8] of the null primitive is the interrupts allowed (IA) bit, used by the FPCP to allow the main processor to check for pending interrupts and to service them before reading the response CIR again. This minimizes the worst case interrupt latency. Refer to **8.3 INTERRUPT LATENCY TIMES** for details.

As shown in Figure 5-1, once the MPU has initiated a floating-point instruction in the MC68881 and transferred the required operands to the coprocessor, the main processor is free to perform other instructions. Meanwhile, the coprocessor converts the operand to internal format, calculates the result, and rounds the result as required. The concurrency shown in Figure 5-1 for an MC68881 applies also to an MC68882.

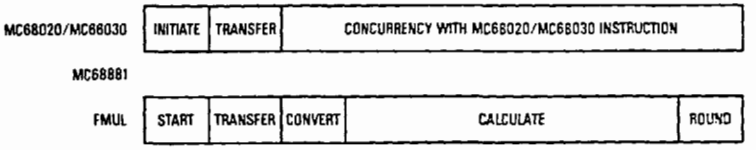


Figure 5-1. MC68881 Concurrency — FMUL Instruction

5.1.1.2 CONCURRENT FLOATING-POINT COMPUTATIONS. An FPCP arithmetic instruction with a floating-point data register destination releases the main processor when the coprocessor completes its transfer phase. If the next instruction is another floating-point instruction and if the main processor writes to the command CIR to initiate the next instruction while the APU is still busy with the previous instruction, the MC68881 returns a null (CA=1, IA=1) primitive in its response CIR. The MC68881 issues the null primitive because the APU can execute only one instruction at a time. Since the bus interface unit (BIU) cannot operate on any other instruction without the APU, no concurrency is possible. The MC68881 cannot begin to execute the second instruction, including the prefetch of necessary operands. The encoding of the response CIR remains unaltered until the instruction in the APU is completed. If the instruction terminates with an exception, a pre-instruction exception is taken.

With the MC68882, if the main processor initiates a second arithmetic instruction while a preceding instruction is executing in the APU as described in the preceding paragraph, the BIU transfers the instruction to the conversion unit (CU). Depending on the instruction, the

operand syntax, and the operand data format, the CU completes execution of the instruction in one of the following ways:

- The instructions that have operand data formats B, W, L, and P are listed in Table 5-1. When the CU receives an instruction with the operand data format of B, W, or L, it requests the BIU to transfer the necessary operand. Then, the CU waits for the APU to become idle so that it can hand off the instruction to the APU. If the instruction has an operand with data format P, the CU does not request the prefetch of the operand. In this case, it waits until the APU is idle to hand off the instruction to the APU.
- If the instruction is an FMOVE.X FPm,FPn instruction, the CU does the following:
  a. Releases the main processor.
  b. Prefetches the source operand from FPm, unless the instruction currently operating in the APU uses FPm as a destination. In that case, the CU waits until the APU is idle before prefetching.
  c. If the selected rounding precision is single or double, waits until the APU is idle and hands off the instruction to the APU.
  d. If the operand data type is NAN, denormalized, or unnormalized, waits until the APU is idle and hands off the instruction to the APU.
  e. If the instruction currently in the APU uses FPn, waits until the APU is idle before proceeding.
  f. Writes the source operand into the destination floating-point data register without involving the APU.
- If the instruction is an FMOVE <ea>, FPn with an operand format of S, D, or X, the CU does the following:
  a. Prefetches the source operand from memory by using the evaluate <ea> and transfer data primitive with CA = 0. This releases the main processor immediately after the source operand is written to the operand CIR.
  b. Converts the memory source operand to internal extended format. If the selected rounding precision is single or double, waits until the APU is idle and hands off the instruction to the APU.
  c. Creates a tag that represents the data type of the converted source operand (normalized, denormalized, zero, infinity, or NAN).
  d. If the operand data type is NAN, unnormalized, or denormalized, waits until the APU is idle and hands off the instruction to the APU.
  e. If the instruction currently in the APU uses FPn, waits until the APU is idle before proceeding.
  f. Writes the converted source operand into FPn without involving the APU.
- If the instruction is an FMOVE FPm,<ea> with a data format of S or D, the CU does the following:
  a. Prefetches the source operand from FPm, unless the instruction currently operating in the APU uses FPm as a destination. In that case, the CU waits until the APU is idle before prefetching.
  b. If the data type of the source operand is unnormalized, denormalized, or NAN, waits until the APU is idle and hands off the instruction to the APU.
  c. Converts the source operand to the destination data format.
  d. If the conversion results in an overflow or underflow, or if the INEX2 trap is enabled, waits until the APU is idle and hands off the instruction to the APU.
  e. Creates a tag that represents the data type of the converted source operand (normalized, denormalized, zero, infinity, or NAN).

f. Transfers the converted operand to the memory destination (without involving the APU) by using the evaluate <ea> and transfer data primitive with CA=0. This releases the main processor immediately after the operand is read from the operand CIR.

- If the instruction is an FMOVE.X FPm,<ea>, the CU does the following:
  a. Prefetches the source operand from FPm, unless the instruction currently operating in the APU uses FPm as a destination. In that case, the CU waits until the APU is idle before prefetching.
  b. If the data type of the source operand is unnormalized, denormalized, or NAN, waits until the APU is idle and hands off the instruction to the APU.
  c. Transfers the converted operand to the memory destination (without involving the APU) by using the evaluate <ea> and transfer data primitive with CA=0. This releases the main processor immediately after the operand is read from the operand CIR.

- If the instruction is listed in Table 5-4 and if the source (FPm) and destination (FPn or FPc:FPs) are all floating-point data registers, the CU does the following:
  a. Releases the main processor.
  b. Prefetches the source operand from FPm if possible.
  c. Waits until the APU is idle and hands off the instruction to the APU.
- If the instruction is listed in Table 5-4 and if the source is external to the MC68882, the CU does the following:
  a. Prefetches the source operand from memory by using the evaluate <ea> and transfer data primitive with CA=0. This releases the main processor immediately after the source operand is written to the operand CIR.
  b. If the source operand format is single or double precision, converts the source operand to the extended precision internal format.
  c. Creates a tag that represents the data type of the converted source operand (normalized, unnormalized, denormalized, zero, infinity, or NAN).
  d. Waits until the APU is idle and hands off the instruction to the APU.

Table 5-1 lists the minimum-concurrency instructions. The monadic operations, designated <mop> in Tables 5-1 and 5-4, are listed in Table 5-2. The dyadic operations, designated <dop> in Tables 5-1 and 5-4, are listed in Table 5-3. Table 5-4 lists the partially-concurrent instructions, and Table 5-5 lists the fully-concurrent instructions.

The instructions that have external operands and are listed in Tables 5-4 and 5-5 prefetch the source operand (if no register conflict exists) and release the main processor after the operand is transferred. When a third instruction is received in the command CIR while the APU is busy and the CU is either busy or waiting to hand off its instruction to the APU, the third instruction must wait. The BIU encodes a null (CA=1, IA=1) primitive in the response CIR until the CU becomes idle or hands off its instruction to the APU. Note that this situation is analogous to the situation in the MC68881 when a second instruction is received in the command CIR while the APU is still busy with a previous instruction. The difference is that the MC68881 waits until the APU is available, while the MC68882 waits until the CU is available.

The conditional instructions listed in Table 5-6 do not allow concurrency. These instructions are not executed unless both the CU and the APU are idle and all exception flags are cleared. An extreme case occurs if the MPU writes to the condition CIR of the MC68882

Table 5-1. Minimum-Concurrency Instructions

Instruction	Operand Syntax	Operand Format
FMOVE	<ea>,FPn FPm,<ea> FPm,<ea>{#k} FPm,<ea>{Dn} <ea>,FPcr FPcr,<ea>	B,W,L,P B,W,L P P L L
FMOVECR	#ccc,FPn	X
FMOVEM	<ea>,<list> <ea>,Dn <list>,<ea> Dn,<ea>	L,X X L,X X
FTST	FPm	B,W,L,P
F<mop>	<ea>,FPn	B,W,L,P
F<dop>	<ea>,FPn	B,W,L,P
FSINCOS	<ea>,FPc:FPs	B,W,L,P

5

Table 5-2. Monadic Instructions

Instruction	Function
FABS	Absolute Value
FACOS	Arc Cosine
FASIN	Arc Sine
FATAN	Arc Tangent
FATANH	Hyperbolic Arc Tangent
FCOS	Cosine
FCOSH	Hyperbolic Cosine
FETOX	e^x
FETOXM1	$e^x - 1$
FGETEXP	Extract Exponent
FGETMAN	Extract Mantissa
FINT	Extract Integer Part
FINTRZ	Extract Integer Part, Rounded-to-Zero
FLOGN	ln(x)
FLOGNP1	ln(x + 1)
FLOG10	$Log_{10}(x)$
FLOG2	$Log_2(x)$
FNEG	Negate
FSIN	Sine
FSINH	Hyperbolic Sine
FSQRT	Square Root
FTAN	Tangent
FTANH	Hyperbolic Tangent
FTENTOX	10^x
FTWOTOX	2^x

Table 5-3. Dyadic Operations

Instruction	Function
FADD	Add
FCMP	Compare
FDIV	Divide
FMOD	Modulo Remainder
FMUL	Multiply
FREM	IEEE Remainder
FSCALE	Scale Exponent
FSGLDIV	Single Precision Divide
FSGLMUL	Single Precision Multiply
FSUB	Subtract

Table 5-4. Partial-Concurrency Instructions

Instruction	Operand Syntax	Operand Format
FTST	<ea> FPm	S,D,X X
F<mop>	<ea>,FPn FPm,FPn	S,D,X
F<dop>	<ea>,FPn FPm,FPn	S,D,X
FSINCOS	<ea>,FPc:FPs FPm,FPc:FPs	S,D,X X

Table 5-5. Fully-Concurrent Instructions

Instruction	Operand Syntax	Operand Format	Degraded to No Concurrency	Degraded to Partial Concurrency
FMOVE	FPm,FPn	X	a	b,c,f
FMOVE	<ea>,FPn	S,D		b,c,f
FMOVE	<ea>,FPn	X		b,c,f
FMOVE	FPm,<ea>	S,D	a	b,d,e
FMOVE	FPm,<ea>	X	a	b

a. Register Conflict of FPm with preceding instruction's destination floating-point data register
b. NAN, Unnormalized or Denormalized Data Types
c. Rounding Precision in FPCR set to Single or Double
d. INEX2 bit in FPCR EXC byte is enabled
e. An Overflow or Underflow occurs
f. Register conflict of FPn with preceding instruction's destination floating-point data register

while both the CU and APU are busy, the appropriate exceptions are enabled, and the instructions in the CU and APU each report an exception. The MC68882 reports these exceptions, one at a time, until both exception handlers have been executed. As a consequence, the conditional instruction is re-started twice to ensure that the reported condition codes contain information reflecting the result of all previous instructions and related exception handlers. Therefore, a sequential execution model can be guaranteed. It is possible that a BSUN exception is reported by the conditional instruction, but the BSUN exception would be reported long after the instructions and related exceptions have been executed and completed.

Table 5-6. Conditional Instructions

Instruction	Operand Syntax	Operand Size or Format
FBcc	<label>	W,L
FDBcc	Dn,<label>	W
FNOP	None	None
FScc	<ea>	B
FTRAPcc	None #xxx	None W,L

Consider the case of two consecutive FMUL instructions followed by an FMOVE instruction, as shown in Figure 5-2. The following assumptions apply:

1. Both FMUL instructions have external operands (opclass 010),
2. The FMOVE instruction has a memory destination (opclass 011), and
3. No exceptions are enabled.

The main processor initiates the second FMUL instruction, and the MC68881 returns the null (CA=1) primitive as long as the APU is involved in the calculate phase of the first FMUL instruction. When the APU becomes available, the BIU returns the evaluate effective address and transfer operands (CA=0) primitive and begins the transfer phase of the second FMUL instruction. At this point, since the CA bit is clear, the main processor begins the execution of another instruction while the MC68881 converts the operand to internal format and begins the calculate phase of the second FMUL instruction. Since the next MPU instruction is an FMOVE instruction, the MPU initiates the FMOVE instruction, but the MC68881 returns the null (CA=1) primitive until the second FMUL instruction completes.

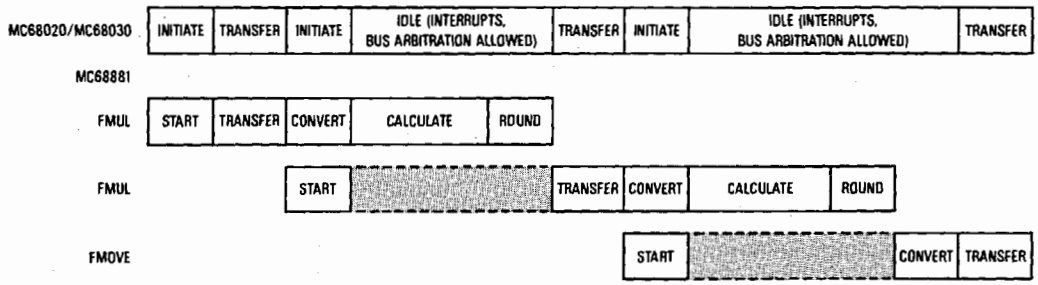


Figure 5-2. MC68881 Concurrency — FMUL Followed by FMUL and FMOVE

Then, the coprocessor completes the FMOVE by converting the operand and transferring it to the main processor.

In this example, the MPU continues to examine the response CIR of the MC68881 while it completes each of the two FMUL instructions. Should an interrupt occur during these times, the main processor services the interrupt but does not initiate any other instruction.

The MC68882 can execute floating-point instructions concurrently by performing conversions between external binary real data formats (S, D, and X) and the internal extended format in the conversion unit (CU) while the arithmetic processing unit (APU) is calculating the result of a preceding instruction. Additional concurrency is provided by making the floating-point data registers accessible to both the CU and APU simultaneously.

Figure 5-3 shows the same three floating-point instructions executing in an MC68882. As soon as the operand of the first FMUL instruction has been transferred to the coprocessor, the main processor begins executing the next instruction, another FMUL instruction with an external source operand. Provided the operand is a binary real operand and no register conflict occurs, the coprocessor can transfer and convert the operand while it continues to calculate the product of the first FMUL instruction. As soon as the operand transfer completes, the main processor begins executing the FMOVE instruction. Since the CU contains the converted operand for the second FMUL instruction at this time, it is not available to convert the source operand of the FMOVE instruction. However, when the APU completes the rounding phase for the first FMUL instruction, it accepts the operand from the CU and begins calculations for the second FMUL instruction. The CU now converts the source operand of the FMOVE instruction to the destination format. The bus interface unit (BIU) transfers the converted operand to the external destination, completing the second FMUL instruction.

The effect of the concurrency provided by the MC68882 is to execute three instructions during a time period equal to the execution time of the first instruction plus the computation time of the second instruction. Execution of the third instruction is completely overlapped by the second instruction.

In this example, execution of the second FMUL instruction is partially concurrent with execution of the first FMUL instruction, and execution of the FMOVE instruction is fully concurrent with execution of the second FMUL instruction. Some MC68882 instructions do not execute concurrently. Others execute partially concurrently, and some execute with full concurrency. However, little concurrency is possible when the operand is in integer or packed decimal format.

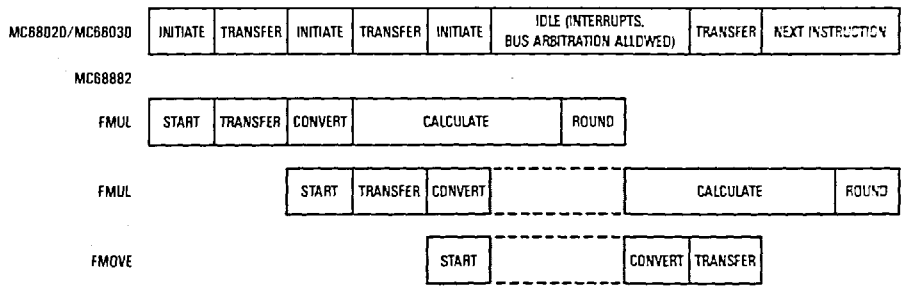


Figure 5-3. MC68882 Concurrency — FMUL Followed by FMUL and FMOVE

5.1.2 Optimization of Code for the MC68882

A program that runs successfully on the MC68881 runs on the MC68882 with improved performance. However, the code can be optimized to exploit the features of the MC68882 for the maximum performance improvement. Optimization requires the following steps:

1. Unroll any rolled loops to obtain at least a 2x unrolled version,
2. Eliminate register conflicts by rearranging FMOVE instructions, and
3. Rearrange FMOVE instructions so that the fastest FMOVE instructions follow the fastest arithmetic instructions, and the slowest FMOVE instructions follow the slowest arithmetic instructions.

5.1.2.1 UNROLLING LOOPS. A rolled loop consists of the instructions to perform the operations of the loop once using a single index value during each iteration. The performance of the MC68882 is improved by unrolling the loop, so that an iteration performs those operations more than once, using two or more index values. The recommended 2x unrolled version performs the operations twice.

The rolled version of a loop allows little optimization; a register conflict is inevitable. The 2x unrolled version can use different floating-point data registers for each repetition of the instructions. The FMOVE instructions can be placed in the optimum locations.

5.1.2.2 AVOIDING REGISTER CONFLICTS. The following rules define conflicts between floating-point data registers.

- A register conflict occurs when the destination register of an instruction is the source register of the following instruction, and that instruction is a fully-concurrent instruction listed in Table 5-5. For example:

```
FADD.D    ⟨ea⟩,FP0
FMOVE.D   FP0,⟨ea⟩        FP0 conflicts
```

- A register conflict occurs when the destination register of an instruction is the destination register of the following instruction, and that instruction is a fully-concurrent instruction listed in Table 5-5. For example:

```
FADD.D    ⟨ea⟩,FP0
FMOVE.D   ⟨ea⟩,FP0        FP0 conflicts
```

- No other combination of source and destination registers of two consecutive instructions cause a register conflict.

The second case (where an FMOVE instruction uses the same destination register as the preceding instruction) is an unlikely case, since the result of the first instruction is lost. However, the MC68882 provides the same result as the MC68881 even for this case.

5.1.2.3 ARRANGING FMOVE INSTRUCTIONS. The FMOVE instruction is fully concurrent when the operands are in binary real data format, no register conflicts exist, and the notes of Table 5-5 do not apply. However, the execution time of the FMOVE instruction is hidden completely only when the overlap time of the preceding instruction exceeds the execution time of the FMOVE instruction. Thus, the fastest FMOVE instructions should follow the fastest arithmetic instructions, FADD, for example. Also, the slowest FMOVE instructions should follow the slowest arithmetic instructions, such as FMUL. Refer to the tables of

execution times in **SECTION 8 INSTRUCTION EXECUTION TIMING** for arithmetic instruction execution times. Table 5-7 lists execution times for some FMOVE instructions.

Table 5-7. FMOVE Instruction Execution Times

Operand	Type	Execution Time (Clocks)
FMOVE.X	FPx,FPy	21
FMOVE.D	<ea>,FPy	39
FMOVE.D	FPy,<ea>	55

5.1.2.4 PERFORMANCE IMPROVEMENT EXAMPLE. The DAXPY subroutine inner loop of the Linpack benchmark (Linpack loop) is an appropriate example for illustrating optimization for the MC68882. Figure 5-4 shows the source code for the rolled version of the Linpack loop.


```
*           VECTOR TIMES A CONSTANT PLUS A VECTOR
*           X(i)=Y(i)*C+X(i)

            MOVE.L      #count,D0
            FMOVE.D     <ea_C>,FP0

LOOPTOP     FMOVE.X     FP0,FP1
            FMUL.D      <ea_Y(i)>,FP1
            FADD.D      <ea_X(i),FP1
            FMOVE.D     FP1,<ea_X(i]
            DBRA        D0,LOOPTOP
```

Figure 5-4. Rolled Version of Linpack Loop

Optimization of this code for the MC68882 consists of unrolling the loop, and rearranging the FMOVE instructions. Notice that FP1 contains the result of the computations using index i, and that FP2 contains the result of the computations using index i+1. Also notice that the two FMOVE instructions that move registers to registers are executed following FADD instructions, and that the FMOVE instructions that move registers to effective addresses are executed following FMUL instructions. Figure 5-5 shows the source code for the optimized Linpack loop.

5.2 SYSTEMS PROGRAMMING

The guidelines for systems programming relate to exception processing. The sizes of the state frames stored by exception handlers are discussed first. Next, the section discusses the FSAVE, BSET, and FRESTORE instructions required in exception handlers. Then, the handling of specific exceptions is discussed. Code that detects the presence of a floating-point coprocessor and identifies the coprocessor is also discussed.

5.2.1 State Frame Sizes

The sizes of the state frames stored by the FSAVE instruction differ for the MC68882 and MC68881, as shown in Table 5-8.

```
*           VECTOR TIMES A CONSTANT PLUS A VECTOR
*           X(i) = Y(i)*C + X(i)

            MOVE.L      #count,D0
            FMOVE.D     <ea_C>,FP0
            FMOVE.X
            FP0,FP1
            FMUL.D      <ea_Y(i)>,FP1
            BRA         LOOPSTRT

LOOPTOP     FMOVE.X     FP0,FP1
            FMUL.D      <ea_Y(i)>,FP1
            FMOVE.D     FP2,<ea_X(i + 1)>
LOOPSTRT    FADD.D      <ea_X(i),FP1

            FMOVE.X     FP0,FP2
            FMUL.D      <ea_Y(i + 1)>,FP2
            FMOVE.D     FP1,<ea_X(i)>

            FADD.D      <ea_X(i + 1)>,FP2

            DBRA        D0,LOOPTOP

            FMOVE.D     FP2,<ea_X(i + 1)>
```

Figure 5-5. Optimized Linpack Loop

Table 5-8. State Frame Sizes

Device	Null Frame	Idle Frame	Busy Frame
MC68881	4 Bytes	28 Bytes	184 Bytes
MC68882	4 Bytes	60 Bytes	216 Bytes

The size of the null state frame is four bytes for both coprocessors. The size of the other state frames is 32 bytes larger for the MC68882 than for the MC68881. The MC68882 uses the additional bytes to store the state of the conversion unit.

5.2.2 Exception Handler Code

The code for floating-point exception handlers for the MC68882 must include the following instructions:

1. An FSAVE instruction at the beginning of the handler (ahead of the first coprocessor instruction)
2. A BSET or similar instruction following the FSAVE instruction to set the EXC_PEND flag (bit 27) of the BIU flag in the idle state frame
3. An FRESTORE instruction immediately preceding the RTE instruction

Handlers for the following exceptions require these instructions even if the handlers contain no floating-point instructions:

Branch or Set on Unordered Condition
Inexact Result
Floating-Point Divide by Zero
Underflow
Operand Error
Overflow
Signaling NAN

Handlers for interrupts, F-line emulation, FTRAPcc instructions, and other exceptions must not set the EXC_PEND bit in the BIU flag long word, but any exception handler that contains one or more floating-point instructions must begin with an FSAVE instruction and have an FRESTORE instruction preceding the RTE instruction. No requirements are imposed on the floating-point protocol violation exception because it is considered to be a catastrophic exception from which no recovery is possible.

When a floating-point exception handler that does not begin with an FSAVE instruction executes in a system that uses an MC68882 coprocessor, one of two things happens. Either the next MC68882 instruction takes the same exception, producing an infinite loop, or it takes a protocol violation exception.

When a floating-point exception handler that begins with an FSAVE instruction but does not set the EXC_PEND bit executes in a system that uses an MC68882 coprocessor, the next MC68882 instruction takes the same exception, also producing an infinite loop.

When a floating-point exception handler that begins with an FSAVE instruction but does not end with an FRESTORE instruction is executed in a system that uses an MC68882 coprocessor, a partially-executed instruction following the exceptional instruction may never be completed. Figure 5-6 shows the required instructions in a minimum exception handler for an MC68882.

```
HANDLER   FSAVE     -(SP)       SAVE INTERNAL STATE
          MOVE.B    (SP),D0     FIRST BYTE OF STATE FRAME
          BEQ       NULL        BRANCH IF NULL FRAME
          CLR.L     D0          CLEAR DATA REGISTER
          MOVE.B    1(SP),D0    LOAD STATE FRAME SIZE
          BSET      #3,(SP,D0)  SET BIT 27 OF BIU
                    .
                    .
                    .
NULL      FRESTORE  (SP)+       RESTORE STATE
          RTE                   RETURN
```

Figure 5-6. Minimum Exception Handler

An exception handler can access the idle state frame to obtain information about the exception. The offsets of the exceptional operand, the operand register, and the BIU flags are different in the MC68881 and the MC68882. In the MC68882, these offsets are greater than those in the MC68881 by $20. For example, the offset for the exceptional operand is $08 in the MC68881 and $28 in the MC68882. However, the negative offsets (from the bottom of the state frame) are the same for both coprocessors. Figure 5-7 shows a code fragment that can be used to access the exceptional operand and the operand register image in an exception handler for either coprocessor.

5.2.3 Processing of Special Conditions

The designs of the MPU, the M68000 coprocessor interface, and the FPCP provide the performance benefits of concurrent operation while maintaining a conventional sequential instruction execution model. Processing of special conditions is also performed as if instructions were executed sequentially. Refer to the coprocessor interface section of the appropriate microprocessor user's manual for additional information.

```
XOPER     EQU     -16          FOR EXCEPTIONAL OPERAND
OPEREG    EQU     -4           FOR OPERAND REGISTER
            .
            .
            .
          MOVE.B  I(SP),D0     LOAD FRAME LENGTH INTO D0
            .
            .
            .
          MOVE.L  XOPER(SP,D0),<ea>   ACCESSES THE FIRST LONGWORD OF THE EXCEPTIONAL OPERAND
          MOVE.L  OPEREG(SP,D0),<ea>  ACCESSES THE OPERAND REGISTER IMAGE
```

Figure 5-7. Idle State Frame Access Example

5.2.3.1 INTERRUPTS. The main processor can process interrupts at any instruction boundary and during the execution of a general or conditional category coprocessor instruction under either of two conditions. When the main processor receives a null (CA = 1, IA = 1) primitive, the MPU services any pending interrupts prior to reading the response CIR. The MPU also services pending interrupts when the trace exception is enabled and the MPU receives a null (CA = 0, IA = 1, PF = 0) primitive.

The MPU uses the ten-word mid-instruction stack frame shown in Figure 7-16 when it services interrupts during the execution of a general or conditional category coprocessor instruction. Using this stack frame allows the MPU to perform all necessary processing and return to read the response CIR. Thus, the MPU services the interrupt while the FPCP continues to execute the coprocessor instruction.

During execution of an FSAVE instruction, when the MPU reads the not ready format word, it also services interrupts. After servicing any pending interrupts, the MPU returns and reinitiates the FSAVE instruction.

5.2.3.2 BUS ARBITRATION. During execution of a floating-point instruction, the MPU can relinquish control of the bus through bus arbitration. If the FPCP has released the MPU and is completing execution of the instruction, relinquishing the bus has no effect on the coprocessor. If the MPU is involved in a dialog with the coprocessor, relinquishing the bus delays the execution of the instruction in the FPCP. However, since the coprocessor communicates with the MPU by placing a response primitive in the response CIR for the MPU to read, no adverse effect occurs. The only effect of the bus arbitration is a longer delay while the coprocessor awaits the services of the MPU.

5.2.3.3 CONTEXT SWITCHING. In a multi-tasking environment, the context of the FPCP may be changed asynchronously with respect to coprocessor operations. The coprocessor may be interrupted at any point during the execution of an instruction. The FSAVE and FRESTORE instructions are used to save and restore the context of the coprocessor during context switches.

An FSAVE instruction stops execution of the instruction in the coprocessor at the earliest interruptable point, and stores the state of the coprocessor. The coprocessor is now available to the program executing in the new context. When the interrupted program resumes, an FRESTORE instruction loads the saved state of the coprocessor, restoring the coprocessor to its previous state. The coprocessor continues from the point at which it was interrupted.

The state frames defined for the null, idle, and busy states of the coprocessor contain all the information the coprocessor requires to resume operation. Inclusion of the coprocessor version number in the format word and the checking of that version number during execution of the FRESTORE instruction prevent restoration of an incompatible context (e.g., an MC68881 context in an MC68882).

5.2.3.4 BUS ERRORS. A bus error can occur during initiation of a coprocessor instruction or while the MPU is accessing memory or CPU address space during execution of a coprocessor instruction. A bus error during initiation of an instruction is used as an indication that the coprocessor is not present, and the MPU takes an F-line emulator exception. A bus error during a memory access indicates that some fault (e.g., parity error or page fault) prevents the memory system from providing the requested operand. The coprocessor interface, being asynchronous, does not require the MPU to service the bus error exception at once. No time restrictions on the main processor's response to a bus error exception exist. After the exception handler has corrected the cause of the bus error, the MPU returns to the point in the coprocessor instruction dialog at which the fault occurred.

5.2.3.5 EXCEPTION PROCESSING. During the execution of a coprocessor instruction, the coprocessor releases the main processor after the main processor has completed all the services the coprocessor requires to execute the instruction. Any exception processing the main processor performs after being released and before initiation of another coprocessor instruction has no effect on the coprocessor.

Either the main processor or the coprocessor can detect an exception during execution of a floating-point instruction. The handlers for these exceptions are bracketed with FSAVE and FRESTORE instructions as previously described to ensure that coprocessor state information about concurrently executing instructions is properly restored after execution of the exception handler completes.

5.2.3.6 SIMULTANEOUS FLOATING-POINT EXCEPTION AND TASK SWITCH INTERRUPT. Since an interrupt signal can occur at any time, a task switch interrupt can occur simultaneously with a floating-point exception detected by the coprocessor. The FPCP and the coprocessor interface with the MPU are designed to preserve the sequential instruction execution model in this case. Figure 5-8 shows an FMUL instruction executing in an MC68882, followed by an FADD instruction. A task switch interrupt occurs as the main processor responds to the exception. The sequence of events is as follows:

1. The MC68882 is executing the two instructions concurrently.
2. The FMUL instruction is executing in the APU, the CU has performed the conversion of the source operand, and the CU is waiting to hand off the FADD instruction when the APU becomes idle. The MC68882 is returning null (CA=0, IA=0) primitives to synchronize the main processor. The main processor is reading the primitives, responding to any pending interrupts.
3. The FMUL instruction detects an exception, which is reported by the FADD instruction with a take mid-instruction exception primitive.
4. The main processor recognizes a pending interrupt as it reads the take mid-instruction exception primitive. Because of the internal timing, however, the MPU processes the floating-point exception first.

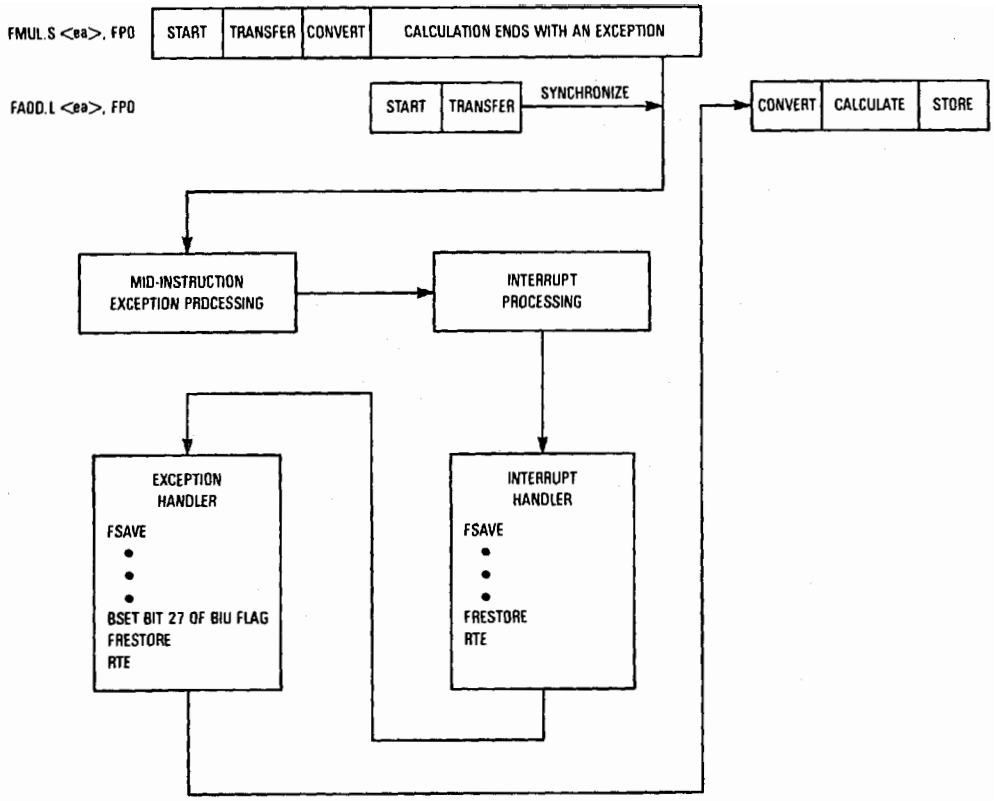


Figure 5-8. Simultaneous Task Switch Interrupt and Floating-Point Exception

5. The processing of the exception completes, and the main processor begins processing the interrupt before executing the first instruction in the floating-point exception handler.
6. The main processor executes the interrupt handler. Because the interrupt handler does not contain a BSET instruction that sets bit 27 of the BIU flag word, the state restored by the FRESTORE instruction in the handler indicates that the floating-point exception has not been serviced. The FADD instruction is not allowed to continue.
7. The floating-point exception handler is executed. This handler includes a BSET instruction that sets bit 27 of the BIU flags word. When the FRESTORE instruction restores the state frame, the FADD instruction continues.

In this example, if the interrupt handler allowed the FADD instruction to continue, the FADD instruction would have overwritten the contents of FP0, and the results would have been incorrect. The MC68882 exception model handles this worst-case situation correctly.

5.2.4 Detecting Coprocessor Presence

A program or an exception handler may need to know if a floating-point coprocessor is available, or which type coprocessor is present. The code fragment in Figure 5-9, which executes at the supervisor privilege level, detects and identifies the coprocessor.

```
                CLR.B       FLAG        CLEAR NO PROCESSOR FLAG
                FNOP                    DETECT COPROCESSOR (SEE NOTE)
                MOVE.B      FLAG,D0     LOAD FLAG
                BNE         NOCOP       NO COPROCESSOR BRANCH
                FSAVE       -(SP)       SAVE INTERNAL STATE
                CLR.L       D0          ZERO INDEX
                MOVE.B      1(SP),D0    OBTAIN STATE FRAME SIZE
                CMPI        #$18,D0     MC68881?
                BEQ         ONE         YES
                  .                     CODE FOR MC68882
                  .
                  .
                BRA         START       END OF MC68882 CODE
ONE               .                     CODE FOR MC68881
                  .
                  .
START           ...                     START OF CODE COMMON TO BOTH COPROCESSORS
```

NOTE: When no coprocessor is present, an exception handler executes at this point. See text.

Figure 5-9. Coprocessor Identification Code

The FNOP instruction takes an F-line emulation exception when no floating-point coprocessor is available. The F-line emulation exception handler must set the no coprocessor flag and increment the stacked PC value by four. The BNE instruction branches around this code when no coprocessor is present. The instructions immediately following the BEQ instruction are executed for an MC68882 coprocessor; those at label ONE are executed for an MC68881.

SECTION 6
EXCEPTION PROCESSING

This section describes how the MC68881/MC68882 (FPCP) and the main processor handle exceptional conditions during the processing of floating-point instructions. These exceptional conditions may be detected internally by the FPCP, internally by the main processor, or externally by the main processor.

The MC68020/MC68030 (MPU) processes exceptions by treating any coprocessor in an M68000 system as an extension to the main processor; the fact that a coprocessor is separate from the main processor is transparent to the programmer. Thus, the exception processing for all coprocessors in a system is coordinated by the main processor in a manner that is consistent across all exception types, whether detected during the execution of an instruction native to the main processor or during a coprocessor instruction.

The processing of an exception detected during the execution of an FPCP instruction involves the following basic steps:

1. Detect the exception
2. Determine the exception vector number and report the exception to the main processor (if detected by the FPCP)
3. Change processing states if needed (user to supervisor)
4. Save the old context of the main processor (performed automatically by the MPU)
5. Load a new context from the address contained in the exception vector table
6. Execute the exception handler
7. Return to the previous context

The first two steps involve slightly different operations for exceptions detected by the main processor and those detected by the FPCP, but the manner in which these operations are performed is consistent with noncoprocessor related exceptions. The major difference in the processing of exceptions detected by the FPCP and the main processor is the point at which exception processing starts. For all main-processor-detected exceptions and some coprocessor-detected exceptions, processing for the exception begins during the execution of the coprocessor instruction by the main processor. However, for many of the coprocessor-detected exceptions, processing for the exception does not begin until after the main processor completes execution of the offending instruction and attempts execution of a new floating-point instruction. The manner of handling this type exception supports a sequential instruction programming model.

The action of the processor during step 7 depends upon the type of exception that was previously taken. When the exception handler completes execution, a return from exception (RTE) instruction is executed, and the previously interrupted program resumes execution at one of the following points:

1. The beginning of the instruction that was pre-empted by an exception detected by or reported to the MPU (pre-instruction exception)
2. The point where the exception occurred during the execution of an instruction (mid-instruction exception)

3. The beginning of the instruction immediately following the instruction that caused or detected the exception (post-instruction exception). Note that neither the MC68881 nor the MC68882 reports the post-instruction exception.

The following paragraphs describe the causes of various coprocessor-related exceptions and how they are handled by the FPCP and the main processor. Throughout this discussion, the main processor is assumed to be an MC68020 or MC68030, although any other processor can be programmed to emulate the M68000 Family coprocessor interface that is implemented on the MPU.

6.1 COPROCESSOR-DETECTED EXCEPTIONS

Coprocessor-detected exceptions fall into two categories: those related to communications with the main processor (F-line traps and protocol violations) and those related to the execution of floating-point instructions (computational errors such as divide by zero, or instructions designed to cause a trap such as the FTRAPcc instruction). The protocol for handling each of these exception types is described in detail in this section.

The main processor coordinates all exception processing. Therefore, when the FPCP detects an exception, it cannot always force exception processing immediately but must wait until the main processor is ready to start exception processing. The main processor is always prepared to process an exception whenever it reads the response coprocessor interface register (CIR). For the MC68881 and, in most cases, for the MC68882, if a coprocessor-detected exception occurs during the calculation phase of an instruction, it is held pending within the FPCP until the next write to the command or condition coprocessor interface register (CIR). Then, instead of returning the first primitive of the dialog for the new instruction, the FPCP returns the take pre-instruction exception primitive to start exception processing for the offending instruction. (For the MC68881, the offending instruction is always the previous floating-point instruction, since no multiple floating-point concurrency is allowed; for the MC68882, the offending instruction may not necessarily be the previous instruction.)

The FPCP may also report an exception after writing an operand to memory. In this case, a take mid-instruction exception primitive is issued after the operand is stored in memory (if a conversion error occurred). The mid-instruction exception allows the exception handler to more easily determine the address of the exceptional operand, since the MC68020 includes the results of the effective address calculation for the destination operand in the mid-instruction stack frame (the long word at offset +$10).

It is possible for the MC68882 to report a mid-instruction exception as a result of an exception created by a previous instruction. This occurs when the instruction in the APU reports an exception while a second instruction in the conversion unit (CU) is waiting to be handed off to the arithmetic processing unit (APU). Consider the case of two FMUL instructions:

```
FMUL.X FP0,FP1 (which results in an exception)
FMUL.B <ea>,FP2
```

At the time the second FMUL instruction is initiated, the first FMUL instruction is still executing in the APU. The CU instructs the bus interface unit (BIU) to fetch the program counter, and prefetch the byte operand. Since the CU cannot convert the byte operand, it instructs the BIU to encode a null (CA=1, IA=1) in the response CIR, and waits to hand

off the instruction to the APU. When the first FMUL instruction finally finishes in the APU and reports an exception, the second FMUL instruction is in the middle of the instruction, hence a take mid-instruction exception is taken. Note that in this case, the destination operand is a floating-point register, and therefore the effective address calculation for the destination operand in the mid-instruction stack frame of the MPU is undefined.

The third point at which the FPCP can indicate an exception to the main processor is in response to a protocol violation. If an unexpected access to a coprocessor interface register causes a protocol violation, the FPCP immediately encodes the response CIR to the take mid-instruction exception primitive with the protocol violation vector number. This allows the protocol violation handler to determine the cause of the violation (either an illegal primitive from the FPCP or an illegal access by the MPU) and perform necessary action. Since an FPCP protocol violation is a catastrophic error, and the FPCP cannot return an illegal primitive, the only appropriate action is to abort the task that detected the protocol violation.

The basic protocol followed in response to a coprocessor-detected exception is:

1. The FPCP encodes the appropriate take exception primitive (pre- or mid-instruction), along with the vector number, in the response CIR.
2. The MPU reads the response CIR (usually in an attempt to initiate the next instruction) and receives the take exception request.
3. The MPU acknowledges the request by writing an exception acknowledge to the control CIR. The appropriate stack frame is then stored in memory, and control is transferred to the exception handler routine.
4. The response to the exception acknowledge differs for the type of exception and for the FPCP, as follows:
    a. Protocol violation:
        MC68881 — Aborts all internal operations that may be active and enters the idle state.
        MC68882 — Same as MC68881.
    b. BSUN and F-line (detected by the coprocessor):
        MC68881 — Clears the exception and enters the idle state.
        MC68882 — Same as MC68881.
    c. Arithmetic (Operr, Overflow, Underflow, Divide by zero, Inexact result):
        MC68881 — Clears the exception and enters the idle state.
        MC68882 — Refer to **5.2.2 Exception Handler Code**.

The following paragraphs discuss the exception vector assignments used by the FPCP, and each of the exception types that can be detected by the FPCP.

The M68000 Family of processors uses a data structure called the exception vector table as a localized dispatching point for all exceptional conditions that may occur in a system. The exception vector table is a 1024-byte structure made up of 256 long word entries. Each entry in the table is a pointer to the routine that services a specific exceptional occurrence. When an exception occurs, the processor supplies an index that selects the vector entry for the exception. The index, called the vector number, is an 8-bit value that is multiplied by four to calculate an offset into the vector table. Of the 256 possible vector numbers, 64 are reserved by Freescale for definition by M68000 Family devices; the remaining 192 are for definition by system designers.

Of the 64 reserved vectors, the MPU defines all but 25. The FPCP utilizes three of the same vector entries defined by the MPU and defines seven additional vectors to support floating-point exceptions. The vectors defined by the FPCP are shown in Table 6-1. The vector number is the value (shown in decimal) that is encoded in the appropriate take exception response primitive (except for the FTRAPcc vector number, which is generated internally by the MPU). The vector offset is the location of the corresponding entry in the vector table. The MPU adds the vector offset to the value contained in the vector base register to calculate the absolute address of the vector. Refer to the appropriate main processor user's manual for further information on the exception processing operations performed by the MPU and for the full definition of the exception vector table.

Table 6-1. MC68881/MC68882 Exception Vector Assignments

Vector Number (Decimal)	Vector Offset (Hexadecimal)	Assignment
7	$01C	FTRAPcc Instruction
11	$02C	F-Line Emulator
13	$034	Coprocessor Protocol Violation
48	$0C0	Branch or Set on Unordered Condition
49	$0C4	Inexact Result
50	$0C8	Floating-Point Divide by Zero
51	$0CC	Underflow
52	$0D0	Operand Error
53	$0D4	Overflow
54	$0D8	Signaling NAN

The following paragraphs describe the causes for each exception, what information is available to the trap handler, and what results occur if traps are enabled or disabled. FPCP instruction exceptions arise from the detection of abnormal conditions during coprocessor instruction execution. All coprocessor-detected instruction exceptions are enabled or disabled via the FPCR ENABLE byte.

Any of eight exception conditions can be detected during the execution of a floating-point instruction. The location of the exception bits in the EXC and ENABLE bytes (contained in the FPSR and FPCR registers, respectively) is shown in Figure 6-1. If more than one enabled exception occurs during the same instruction, then the highest priority instruction trap is taken (BSUN is the highest; INEX2/INEX1 is the lowest). When multiple exceptions occur, the FPCP traps to the highest priority exception that is enabled, and the lower priority exception does not cause a second trap. It is the programmer's responsibility to determine if any of the exception bits that have lower priority than the exception taken are set.

FPCP instruction exceptions that arise from the move floating-point data register to external destination instructions are reported to the MPU as mid-instruction exceptions. All other MC68881-detected instruction exceptions are reported as pre-instruction exceptions, and all other MC68882-detected instruction exceptions are reported as either pre-instruction or mid-instuction exceptions. The FPCP move multiple and move system control register instructions cannot generate coprocessor detected instruction exceptions. The FSAVE instruction can generate a coprocessor-detected exception only when it interrupts an FSAVE

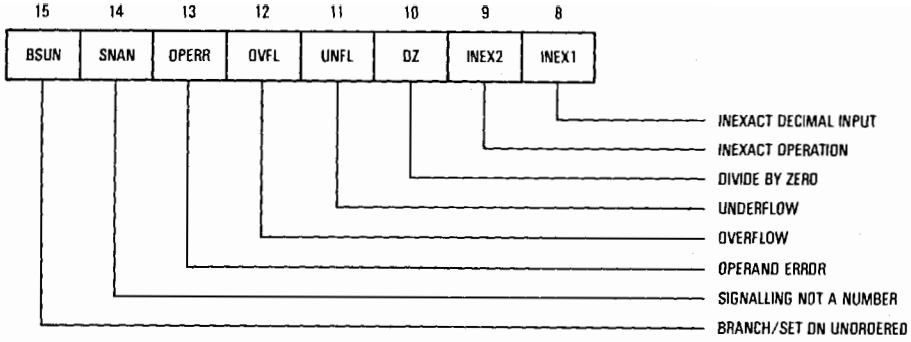


Figure 6-1. EXC and ENABLE Byte Bit Assignments

or FRESTORE operation in progress. The FRESTORE instruction can generate coprocessor-detected instruction exceptions only when the state frame format written to the coprocessor is not recognized.

In the following exception descriptions, the term "intermediate result" is used frequently. During the execution of a floating-point operation, the FPCP arithmetic processing unit (APU) contains a 67-bit mantissa (for rounding purposes) and a 17-bit exponent (to ensure that overflow or underflow can never occur during the main algorithm). At the end of the operation, this intermediate result must be stored in a floating-point data register, in an MPU data register, or in memory. This intermediate result is checked for underflow, rounded, and checked for overflow to obtain the final result.

6.1.1 Branch/Set on Unordered (BSUN)

The BSUN exception is the result of performing a conditional test associated with the FBcc, FDBcc, and FTRAPcc instructions when an unordered condition is present. (An unordered condition occurs when an input to an arithmetic operation is a NAN.) The BSUN exception can only occur during FPCP conditional instructions with the following IEEE nonaware branch condition predicates:

GT	Greater Than	GL	Greater Than or Less Than
NGT	Not Greater Than	NGL	Not Greater Than or Less Than
GE	Greater Than or Equal	GLE	Greater Than or Less Than or Equal
NGE	Not Greater Than or Equal	NGLE	Not Greater Than or Equal Less Than or Equal
LT	Less Than	SF	Signaling False
NLT	Not Less Than	ST	Signaling True
LE	Less Than or Equal	SEQ	Signaling Equal
NLE	Not Less Than or Equal	SNE	Signaling Not Equal

If the APU is busy (MC68881) or if the CU (MC68882) is busy, a null (CA = 1, IA = 1) primitive is returned, and the MPU continues to reexamine the response CIR. If an exception is pending, a take pre-instruction exception primitive is returned. After the appropriate exception handler is executed, the conditional instruction is restarted. When either the APU is idle (MC68881) or the APU and CU are idle (MC68882) and no exceptions are pending, FPCP checks for a BSUN exception, evaluates the conditional predicate, and reports the result to the MPU.

The MPU can write to the condition CIR of the MC68882 when both the CU and APU are busy. If exceptions are enabled and if each of the instructions report an exception, the MC68882 reports the exceptions and executes the handlers, one at a time. The MC68882 restarts the conditional instruction after returning from each exception handler; that is, the MC68882 restarts the instruction twice. It is important to note that the coprocessor completes all previous instructions and the MPU completes any executing exception handler before the conditional instruction checks for a BSUN exception, evaluates the conditional predicate, and reports the result to the MPU.

The FPCP detects a BSUN exception if the conditional predicate is one of the IEEE nonaware branches, and the NAN condition code bit is set. When the FPCP detects this exception, it sets the BSUN bit in the FPSR exception status byte.

Trap Disabled Results: The FPCP evaluates the condition and reports the result to the MPU in the response CIR.

Trap Enabled Results: The FPCP reports a pre-instruction exception to the MPU with the BSUN vector number instead of a true or false indication.

The BSUN exception is unique in that the trap is taken before the conditional predicate is evaluated. Furthermore, the instruction that caused the BSUN exception is re-executed following return from the BSUN trap handler. Therefore, it is the responsibility of the trap handler to prevent the conditional instruction from taking the BSUN trap again. Four ways are available to prevent taking the trap again.

The first way involves incrementing the stored program counter in the stack to bypass the conditional instruction. This technique applies to situations where a fall-through is desired. Be aware that accurate calculation of the program counter increment requires detailed knowledge of the size of the conditional instruction being bypassed.

The second method is to clear the NAN bit of the FPSR condition code byte. However, this alone cannot deterministically control the result indication (true or false) which would be returned when the conditional instruction re-executes.

The third method is to disable the BSUN trap. Like the second method, this method cannot control the result indication (true or false) which would be returned when the conditional instruction re-executes.

The fourth method involves examining the condition predicate and setting the condition code in the FPSR accordingly. This technique gives the most control since it is possible to pre-determine the direction of program flow. Bit 7 of the F-line operation word indicates where the conditional predicate is located. If bit 7 is set, the conditional predicate is the lower six bits of the F-line operation word. Otherwise, the conditional predicate is the lower six bits of the instruction word, which immediately follows the F-line operation word. Using the conditional predicate and the table in **4.4.1 IEEE NonAware Tests**, the condition codes can be set to return a known result indication when the conditional instruction is re-executed.

6.1.2 Signaling Not-a-Number

An SNAN is used as an escape mechanism for a user defined, non-IEEE data type. The FPCP never creates an SNAN as a result of an operation; a NAN created by an operand error exception is always a nonsignaling NAN.

When an SNAN is an operand involved in an arithmetic instruction, the SNAN bit is set in the FPSR exception byte. Since the FMOVEM, FMOVE FPcr, and FSAVE instructions do not modify the status bits, they cannot generate exceptions. Therefore, these instructions are useful for manipulating SNANs.

Trap Disabled Results: If the destination data format is S, D, X, or P, then the SNAN bit in the NAN is set to one and the resulting nonsignaling NAN is transferred to the destination. No bits other than the SNAN bit of the NAN are modified, although the input NAN is truncated if necessary. If the destination data format is B, W, or L, then the 8, 16, or 32 most significant bits of the SNAN significand, with the SNAN bit set, are written to the destination.

Trap Enabled Results: For memory or MPU data register destinations, the result is written in the same manner as if the trap were disabled, and then a mid-instruction exception is signaled. If desired, the trap handler can overwrite the result.

For floating-point data register destinations, instruction execution is terminated, and the floating-point data registers are not modified. In this case, the SNAN trap handler should supply the result.

Note that the trap handler should use only the FMOVEM instruction to read or write the floating-point data registers, since FMOVEM cannot generate further exceptions. Also, only an FMOVEM instruction can write a SNAN into a floating-point data register.

6.1.3 Operand Error

The operand error category encompasses problems arising in a variety of operations, and includes those errors not frequent or important enough to merit a specific exception condition. Basically, an operand error occurs when an operation has no mathematical interpretation for the given operands. The possible operand errors are listed in Table 6-2. When an operand error occurs, the OPERR bit is set in the FPSR exception status byte.

Trap Disabled Results: For a memory or MPU data register destination, several possible results can be supplied, depending on the destination size and error type. (An operand error is never generated when the destination is an MPU data register or memory and the destination format is S, D, or X.)

If the operand error is caused by an integer overflow or if the floating-point data register to be stored contains infinity, the result is the largest positive or negative integer that can fit in the specified destination format size. If the destination is B, W, or L and the floating-point number to be stored is a NAN, then the 8, 16, or 32 most significant bits of the NAN significand are stored as the result.

For packed decimal results, if the k factor is greater than +17, the result returned is a packed decimal string that assumes a k factor equal to +17. For packed decimal results where the absolute value of the exponent is greater than 999, the decimal string is returned with the three least significant exponent digits in EXP2, EXP1, and EXP0. The fourth digit, EXP3, is supplied in the most significant four bits of the third byte in the string. Refer to **3.6 DATA FORMAT DETAILS** for the packed decimal string format.

If the destination is a floating-point data register, an extended precision nonsignaling NAN (with all ones mantissa) is stored in the destination floating-point data register.

Table 6-2. Possible Operand Errors

Instruction	Condition Causing Operand Error
FACOS	Source is ±infinity, >+1, or <−1
FADD	(+infinity)+(−infinity) or (−infinity)+(+infinity)
FASIN	Source is ±infinity, >+1, or <−1
FATANH	Source is >+1, or <−1, Source=±infinity
FCOS	Source is ±infinity
FDIV	0/0 or infinity/infinity
FGETEXP	Source is ±infinity
FGETMAN	Source is ±infinity
FLOG10	Source is <0, Source=−infinity
FLOG2	Source is <0, Source=−infinity
FLOGN	Source is <0, Source=−infinity
FLOGNP1	Source is <−1, Source=−infinity
FMOD	Floating-Point Data Register is ±infinity or Source is 0, Other Operand is Not a NAN
FMOVE to B,W, or L	Integer Overflow/Underflow, Source is Non-Signaling NAN, or Source is ±infinity
FMOVE to P	Result Exponent >999 (Decimal) or k-Factor >+17
FMUL	One Operand is 0, Other Operand is ±infinity
FREM	Floating-Point Data Register is ±infinity or Source is 0, Other Operand is Not a NAN
FSCALE	Source is ±infinity, Other Operand is Not a NAN
FSGLDIV	0/0 or infinity/infinity
FSGLMUL	One operand is 0, Other Operand is infinity
FSIN	Source is ±infinity
FSINCOS	Source is ±infinity
FSQRT	Source <0, Source=−infinity
FSUB	Source and floating-point data register are +infinity or source and FPn are −infinity
FTAN	Source is ±infinity

Trap Enabled Results: For memory or MPU data register destinations, the destination operand is written as if the trap were disabled, and then a take exception primitive is returned to the MPU. This can only occur for the FMOVE FPm,<ea> instruction, and the exception is reported as a mid-instruction exception. If desired, the trap handler can overwrite the result generated by the FPCP.

If the destination is a floating-point data register, the register is not modified by the FPCP. In this case, the trap handler should generate the appropriate result.

To enable the trap handler to return a result for memory or MPU data register destinations, the MPU and the FPCP supply:

1. The address of the instruction where the error occurred (in the FPIAR). By examining the instruction, the trap handler may determine the operation being performed, the value of the second operand (for dyadic instructions), and the destination location.
2. The address of the destination in the mid-instruction stack frame (at offset +$10). This allows the trap handler to overwrite the NAN, if necessary, without recalculating the effective address.

To enable the trap handler to return a result for floating-point data register destinations, the MPU and the FPCP supply:

1. The address of the instruction where the error occurred (in the FPIAR). By examining the instruction, the trap handler may determine the operation being performed, the value of the second operand (for dyadic instructions), and the destination location.

2. The exceptional operand in the FPCP idle state frame. For additional FSAVE state frame information, refer to **6.4.2 State Frames**. When an SNAN trap occurs, the exceptional operand is the source input argument converted to extended precision.

Note that the trap handler should use only the FMOVEM instruction to read or write the floating-point data registers since FMOVEM cannot generate further exceptions or change the condition codes.

6.1.4 Overflow

An overflow occurs when the intermediate result of an arithmetic operation is too large to be represented in a floating-point data register using the selected rounding precision. A store-to-memory operation overflows when the value in the source floating-point data register is too large to be represented in the destination format.

Overflow is detected for arithmetic operations where the destination is a floating-point data register when the intermediate result exponent is greater than or equal to the maximum exponent value of the selected rounding precision. (Refer to **2.2.2 FPCR Mode Control Byte**.) Overflow is detected for store-to-memory operations when the intermediate result exponent is greater than or equal to the maximum exponent value of the destination data format. Overflow can only occur when the destination is in the S, D, or X format. Overflows when converting to the B, W, or L integer and packed decimal formats are included as operand errors. Refer to **3.6 DATA FORMAT DETAILS** for the maximum exponent value for each format. At the end of any operation that could potentially overflow, the intermediate result is checked for underflow, rounded, and checked for overflow before it is stored to the destination. If overflow occurs, the OVFL bit is set in the FPSR exception byte.

NOTE

An overflow can occur when the destination is a floating-point data register and the selected rounding precision is single or double even if the intermediate result is small enough to be represented as an extended precision number. The intermediate result is rounded to the selected precision (both the mantissa and the exponent), and then the rounded result is stored in extended precision format. If the magnitude of the intermediate result exceeds the range of the selected rounding precision format, an overflow occurs. The FSGLMUL and FSGLDIV instructions are the exceptions in that, although the mantissa of the intermediate result is rounded to single precision, the exponent remains an extended format exponent. Therefore, those instructions can never report an overflow as long as the intermediate result is small enough to be represented in extended precision format.

Trap Disabled Results: The current rounding mode determines the value to be stored at the destination, as follows:

Rounding Mode	Result
RN	Infinity, with the sign of the intermediate result
RZ	Largest magnitude number, with the sign of the intermediate result
RM	For positive overflow, largest positive number For negative overflow, − infinity
RP	For positive overflow, + infinity For negative overflow, largest negative number

Trap Enabled Results: The result stored in the destination is the same as the result stored when the trap is disabled, and a take exception primitive is returned to the MPU. If the destination is memory or an MPU data register, the operand is stored, and then a take mid-instruction exception primitive is issued. If the destination is a floating-point data register, a take exception primitive is returned when the MPU reads the response CIR of the FPCP. Since the MC68881 does not allow multiple floating-point concurrency, a take pre-instruction exception is reported when the MPU attempts the next floating-point instruction. The MC68882 can report an exception as a mid-instruction exception on a subsequent floating-point instruction.

The address of the instruction that causes the overflow is available to the trap handler in the FPIAR. By examining the instruction, the trap handler can determine the arithmetic operation type and destination location. The trap handler can execute an FSAVE instruction to obtain additional information. When an FSAVE is executed, the exceptional operand is stored in the state frame. Refer to **6.4.2 State Frames** for details of the FSAVE instruction state frames. When an overflow occurs, the exceptional operand is defined differently for various destination types:

1. Memory or MPU data register destination — the value in the exceptional operand is the intermediate result mantissa rounded to the destination precision, with a 15-bit exponent biased as a normal extended precision number. In the case of a memory destination, the evaluated effective address of the operand is available in the MPU mid-instruction stack frame (at offset +$10). This allows the trap handler to overwrite the default result, if necessary, without recalculating the effective address.
2. Floating-point data register destination — the value in the exceptional operand is the intermediate result rounded to extended precision, with an exponent bias of $3FFF–$6000 rather than $3FFF. The additional bias of −$6000 is used to "wrap" the 17-bit intermediate value into a value that can be represented in 15 bits. To recover the 17-bit twos-complement exponent of the intermediate result, the 15-bit exponent of the exceptional operand should be sign extended to at least 17 bits (i.e., if it is manipulated in an MPU data register, it is sign extended to a *long word* value), and then the bias of $3FFF–$6000 should be subtracted from that number. Note that for most operations, the intermediate exponent value does not exceed 32,767 and thus can be contained in a 16-bit integer. However, a completely general exception handler should calculate a 17-bit exponent value.

   In addition to normal overflow, the exponential instructions implemented by the FPCP (e^x, 10^x, 2^x, SINH, COSH, and FSCALE) may generate results that overflow the 17-bit exponent used for intermediate results. For example, the e^x function can easily overflow the 17-bit intermediate exponent if the source value is a large number ($x \geq +18{,}192$). When such an overflow occurs (called a catastrophic overflow), the exceptional operand exponent value is set to $0000. This value is easily distinguished from the exceptional operand exponent values produced by normal overflow processing. The

smallest exceptional operand exponent value that can be produced by a normal overflow is $1FFF ($04000 + $3FFF–$6000, truncated to 15 bits), while the largest exceptional operand exponent value is $7FFF ($0A000 + $3FFF–$6000, truncated to 15 bits). The catastrophic overflow exceptional operand exponent value of $0000 is produced any time the unbiased exponent of the calculated intermediate result is a value greater than $0A000.

Note that the trap handler should use only the FMOVEM instructions to read or write the floating-point data registers since FMOVEM cannot generate further exceptions or change the condition codes.

6.1.5 Underflow

An underflow occurs when the intermediate result of an arithmetic operation is too small to be represented as a normalized number in a floating-point data register using the selected rounding precision. A store-to-memory operation underflows when the value in the source floating-point data register is too small to be represented in the destination format as a normalized number. Underflow is detected for arithmetic operations where the destination is a floating-point data register when the intermediate result exponent is less than or equal to the minimum exponent value of the selected rounding precision (refer to **2.2.2 FPCR Mode Control Byte**).

Underflow is detected for store-to-memory operations when the intermediate result exponent is less than or equal* to the minimum exponent value of the destination data format.

Underflow can only occur when the destination format is S, D, or X. When the destination format is packed decimal, underflows are included as operand errors. When the destination format is B, W, or L, the conversion underflows to zero without causing either an underflow or an operand error. See **3.6 DATA FORMAT DETAILS** for the minimum exponent value for each format.

At the end of any operation that could potentially underflow, the intermediate result is checked for underflow, rounded, and checked for overflow before it is stored at the destination. If an underflow occurs, the UNFL bit is set in the FPSR exception status byte.

NOTE

An underflow can occur when the destination is a floating-point data register and the selected rounding precision is single or double even if the intermediate result is large enough to be represented as an extended precision number. The intermediate result is rounded to the selected precision (both the mantissa and the exponent), and then the rounded result is stored in extended precision format. If the magnitude of the intermediate result is too small to be represented in the selected rounding precision format, an underflow occurs. The FSGLMUL and FSGLDIV are exceptions in that, although the mantissa of the intermediate result is rounded to single precision, the exponent remains an extended precision format

*Underflow is NOT detected for intermediate result exponents that are equal to the extended precision minimum exponent, since the explicit integer part bit of extended precision permits representation of normalized numbers with a minimum extended precision exponent.

exponent. Therefore, these instructions can never report an underflow as long as the intermediate result is large enough to be represented in the extended precision format.

Trap Disabled Results: The result that is stored in the destination is either a denormalized number or zero. The intermediate result is always normalized because the FPCP ALU and temporary registers use a 17-bit exponent. Denormalization is accomplished by shifting the mantissa of the intermediate result to the right while incrementing the exponent until it is equal to the denormalized exponent value for the destination format. The denormalized intermediate result is rounded to the selected rounding precision or destination format.

If, in the process of denormalizing the intermediate result, all of the significant bits are shifted off to the right, the selected rounding mode determines the value to be stored at the destination, as follows:

Rounding Mode	Result
RN	Zero, with the sign of the intermediate result
RZ	Zero, with the sign of the intermediate result
RM	For positive underflow, + zero For negative underflow, smallest denormalized negative number
RP	For positive underflow, smallest denormalized positive number For negative underflow, − zero

Trap Enabled Results: The result stored in the destination is the same as the result stored when traps are disabled, and a take exception primitive is returned to the MPU. If the destination is memory or an MPU data register, the operand is stored, and then a take mid-instruction exception primitive is issued. If the destination is a floating-point data register, a take exception primitive is returned when the MPU reads the response CIR of the FPCP. Since the MC68881 does not allow multiple floating-point concurrency, the exception is always reported as a pre-instruction exception when the next floating-point instruction is attempted. The MC68882, however, may report an exception as a mid-instruction exception on a subsequent floating-point instruction.

The address of the instruction that caused the underflow is available to the trap handler in the FPIAR. By examining the instruction, the trap handler can determine the arithmetic operation type and destination location. The trap handler can execute an FSAVE instruction to obtain additional information. When an FSAVE instruction is executed, the exceptional operand is stored in the state frame. Refer to **6.4.2 State Frames** for details of FSAVE state frames. The exceptional operand is defined differently for various destination types:

1. Memory or MPU data register destination — the value in the exceptional operand is the intermediate result mantissa rounded to the destination precision, with a 15-bit exponent biased as a normal extended precision number. In the case of a memory destination, the evaluated effective address of the operand is available in the MPU mid-instruction stack frame (at offset +$10). This allows the trap handler to overwrite the default result, if necessary, without recalculating the effective address.
2. Floating-point data register destination — the value in the exceptional operand is the intermediate result mantissa rounded to extended precision, with an exponent bias of $3FFF + $6000 rather than $3FFF. The additional bias of +$6000 is used to "wrap" the 17-bit intermediate value into a value that can be represented in 15 bits. To recover the 17-bit twos-complement exponent of the intermediate result, the 15-bit exponent

of the exceptional operand is sign extended to at least 17 bits (i.e., if it is manipulated in an MPU data register, it is sign extended to a long-word value), and then the bias of $3FFF + $6000 is subtracted from that number. Note that for most operations, the intermediate exponent value is not less than −32,768, and thus can be contained in a 16-bit integer. However, a completely general exception handler should calculate a 17-bit exponent value.

In addition to normal underflow, the exponential instructions implemented by the FPCP (e^x, 10^x, 2^x, SINH, COSH, and FSCALE) may generate results that underflow the 17-bit exponent used for intermediate results. For example, the e^x function can easily underflow the 17-bit intermediate exponent if the source value is a large number ($x \leq -8{,}192$). When such an underflow occurs (called a catastrophic underflow), the exceptional operand exponent value is set to $0000. This is the smallest exception operand exponent value that can be produced by a normal underflow ($16001 + $3FFF + $6000, truncated to 15 bits), while the largest underflow exponent value is $5FFF ($1C000 + $3FFF + $6000, truncated to 15 bits). The catastrophic underflow exceptional operand exponent value of $0000 is produced any time the unbiased 17-bit exponent of a calculated intermediate result has a value less than or equal to $16001.

Note that the trap handler should use only the FMOVEM instructions to read or write to the floating-point data registers since FMOVEM cannot generate further exceptions or change the condition codes.

NOTE

The IEEE standard defines two causes of an underflow:

1. When a result is very small, the absolute value of the number is less than the minimum number that can be represented by a normalized number in a specific format.
2. When loss of accuracy occurs while attempting to calculate a very small number (a loss of accuracy also causes an inexact exception).

The IEEE standard specifies that if the underflow trap is disabled, an underflow should only be signaled when both of these cases are satisfied (i.e., the result is too small to represent with a given format, and there is a loss of accuracy during the calculation of the final result). If the trap is enabled, the underflow should be signaled any time a tiny result is produced, regardless of whether accuracy is lost in calculating it.

The FPCP UNFL bit in the AEXC byte of the FPSR implements the IEEE trap disabled definition, since it is only set when a very small number is generated and accuracy has been lost when calculating that number. The UNFL bit in the EXC byte implements the IEEE trap enabled definition, since it is set anytime a tiny number is generated. Thus, if the FPCP underflow trap is enabled, a trap occurs when very small size alone is detected (as the IEEE standard specifies) to support the emulation of machines that underflow to zero, rather than using the IEEE gradual underflow method (i.e., denormalized numbers). If the underflow trap is disabled, the UNFL bit in the AEXC byte may be examined at the end of a calculation to determine if any result produced during the operation required representation as a denormalized number, and accuracy was lost when denormalizing and rounding that result.

6.1.6 Divide-by-Zero

This exception occurs when a zero divisor occurs in a division, or when a transcendental function is asymptotic with infinity as the asymptote. Table 6-3 lists the instructions that can generate the divide-by-zero exception. When a divide-by-zero is detected, the DZ bit is set in the FPSR exception status byte.

Table 6-3. Possible Divide-by-Zero Exceptions

Instruction	Input Operand Value
FATANH	Source Operand = ±1
FDIV	Source Operand = 0 and FPn is Not a NAN, Infinity, or Zero
FLOG10	Source Operand = 0
FLOG2	Source Operand = 0
FLOGN	Source Operand = 0
FLOGNP1	Source Operand = −1
FSGLDIV	Source Operand = 0 and FPn is Not a NAN, Infinity, or Zero

Trap Disabled Results: Store the following results in the destination floating-point data register:

- For the FDIV and FSGLDIV instructions, return an infinity with the sign set to the exclusive OR of the signs of the input operands.
- For the FLOGx instructions, return minus infinity.
- For the FATANH instruction, return a +infinity if the source operand is −1; or a −infinity if the source operand is +1.

Trap Enabled Results: The destination floating-point data register is not modified, and a take exception primitive is returned when the MPU reads the response CIR of the FPCP. Since the MC68881 does not allow multiple floating-point concurrency, the exception is always reported as a pre-instruction exception when the next floating-point instruction is attempted. The MC68882, however, may report an exception as a mid-instruction exception on a subsequent floating-point instruction. The trap handler must generate a result to store in the destination.

To assist the trap handler in this function, the FPCP supplies:

1. The address of the instruction where the divide-by-zero occurred (in the FPIAR). By examining this instruction, the trap handler can determine the operation being performed, the value of the source operand (for dyadic instructions), and the destination floating-point register number.
2. The FSAVE instruction that places the exceptional operand in a state frame. For additional FSAVE state frame information, refer to **6.4.2 State Frames**. The exceptional operand is the source input argument converted to extended precision.

Note that the trap handler should use only the FMOVEM instruction to read or write the floating-point data registers, since FMOVEM cannot generate further exceptions or change the condition codes.

6.1.7 Inexact Result

The FPCP provides two inexact bits (INEX1 and INEX2) to help distinguish between inexact results generated by decimal input (INEX1) and other inexact results (INEX2). Two inexact bits are useful in instructions in which both types of inexacts can occur, such as:

```
FDIV.P      #7E-1,FP3
```

In this case, the packed decimal to extended precision conversion of the immediate source operand causes an inexact error to occur which is signaled as INEX1. Furthermore, the subsequent divide might also produce an inexact result and cause INEX2 to be set. Therefore, the FPCP provides two inexact bits in the FPSR exception status byte to distinguish these two cases.

Note that only one inexact exception vector number is generated by the FPCP. If either of the two inexact exceptions is enabled, the MPU fetches the inexact exception vector, and the exception handler routine is initiated. Refer to **6.1.8 Inexact Result on Decimal Input** for a discussion of INEX1.

In a general sense, INEX2 is the condition that exists when any operation, except the input of a packed decimal number, creates a floating-point intermediate result whose infinitely precise mantissa has too many significant bits to be represented exactly in the selected rounding precision (refer to **2.2.2 FPCR Mode Control Byte**) or in the destination data format. If this condition occurs, the INEX2 bit is set in the FPSR exception status byte, and the infinitely precise result is rounded as described in the next paragraph.

The FPCP supports the four rounding modes specified by the IEEE standard. These modes are round to nearest (RN), round toward zero (RZ), round toward plus infinity (RP), and round toward minus infinity (RM). The rounding definitions are:

Rounding Mode	Result
RN	The representable value nearest to the infinitely precise intermediate value is the result. If the two nearest representable values are equally near (a tie), then the one with the least significant bit equal to zero (even) is the result. This is sometimes referred to as "round nearest, even".
RZ	The result is the value closest to, and no greater in magnitude than, the infinitely precise intermediate result. This is sometimes referred to as the "chop mode", since the effect is to clear the bits to the right of the rounding point.
RM	The result is the value closest to and no greater than the infinitely precise intermediate result (possibly minus infinity).
RP	The result is the value closest to and no less than the infinitely precise intermediate result (possibly plus infinity).

The RM and RP rounding modes are often referred to as "directed rounding modes" and are useful in interval arithmetic. Rounding is accomplished using the intermediate result format shown in Figure 6-2.

Depending on the selected rounding precision or destination data format in effect, the location of the least significant bit of the fraction and the locations of the guard, round, and sticky bits in the 67-bit intermediate result mantissa varies.

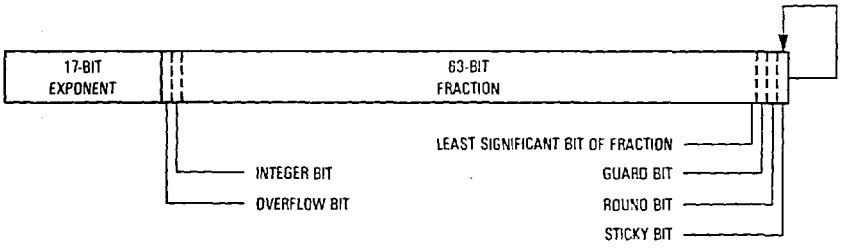


Figure 6-2. Intermediate Result Format

The guard and round bits are always calculated exactly. The sticky bit is used to create the illusion of an infinitely wide intermediate result mantissa. As shown by the arrow in Figure 6-2, the sticky bit is the logical OR of all the bits in the infinitely precise result to the right of the round bit. During the calculation stage of an arithmetic operation, any nonzero bits generated that are to the right of the round bit set the the sticky bit (which is used in rounding) to one. Because of the sticky bit, the rounded intermediate result for all required IEEE arithmetic operations in the round-to-nearest mode is in error by no more than one-half unit in the last place. For transcendental instructions, the result may not be this accurate (refer to **4.3 COMPUTATIONAL ACCURACY**).

NOTE

When the FPCP is programmed to operate in the single or double precision rounding mode, a method referred to as "range control" is used to assure correct emulation of a machine that only supports single or double precision arithmetic. When the FPCP performs any calculation, the intermediate result is in the format shown in Figure 6-2, and a rounded result stored into a floating-point data register is always in the extended precision format. However, if the single or double precision rounding mode is in effect, the final result generated by the FPCP is within the range of the format (except for the FSGLDIV and FSGLMUL instructions, as described in **4.5.5.2 UNDERFLOW, ROUND, OVERFLOW**).

Range control is accomplished by not only rounding the intermediate result mantissa to the specified precision, but also checking the 17-bit intermediate exponent to ensure that it is within the representable range of the selected rounding precision format. If the intermediate exponent exceeds the range of the selected precision, the exponent value appropriate for an underflow or overflow is stored as the result in the 15-bit extended precision format exponent. For example, if the rounding precision and mode is single/RM and the result of an arithmetic operation overflows the magnitude of the single precision format, the largest normalized single precision value is stored as an extended precision number in the destination floating-point data register (i.e., an unbiased 15-bit exponent of $00FF and a mantissa of $FFFFFF0000000000). If an infinity is the appropriate result for an underflow or overflow, the infinity value for the destination data type is stored as the result (i.e., an exponent with the maximum value and a mantissa of zero).

Figure 6-3 shows the algorithm that is used to round an intermediate result to the selected rounding precision or destination data format. If the destination is a floating-point register,

```
BEGIN
    IF GUARD, ROUND AND STICKY=0
        THEN (RESULT IS EXACT)
            DON'T SET INEX2
            DON'T CHANGE THE INTERMEDIATE RESULT
        ELSE (RESULT IS INEXACT)
            SET INEX2 IN THE FPSR EXC BYTE
        SELECT THE ROUNDING MODE
            RM: IF INTERMEDIATE RESULT IS NEGATIVE
                    THEN ADD 1 TO LSB
            RN: IF GUARD=1 AND ROUND AND STICK=0 (TIE CASE)
                    THEN IF LSB=1 ADD 1 TO LSB
                    ELSE IF GUARD=1 ADD 1 TO LSB
                    ENDIF
            RP: IF INTERMEDIATE RESULT IS POSITIVE
                    THEN ADD 1 TO LSB
            RZ: (FALL THROUGH; GUARD, ROUND AND STICKY ARE CHOPPED)
        END SELECT
        IF OVERFLOW=1
            THEN
                SHIFT MANTISSA RIGHT BY ONE BIT
                ADD 1 TO THE EXPONENT
        END IF
        SET GUARD, ROUND AND STICK TO 0
    END IF
END
```

Figure 6-3. Rounding Algorithm

the rounding boundary is determined by the selected rounding precision in the FPSR. If the destination is external memory or an MPU data register, the rounding boundary is determined by the destination data format. If the rounded result of an operation is not exact, then the INEX2 bit is set in the FPSR exception status byte.

Trap Disabled Results: The rounded result is delivered to the destination.

Trap Enabled Results: The rounded result is delivered to the destination, and an exception is reported to the MPU. If the destination is memory or an MPU data register, a take mid-instruction exception primitive is returned immediately after the operand is stored. If the destination is a floating-point data register, a take exception primitive is returned when the MPU reads the response CIR of the FPCP. Since the MC68881 does not allow multiple floating-point concurrency, the exception is always reported as a pre-instruction exception when the next floating-point instruction is attempted. The MC68882, however, may report an exception as a mid-instruction exception on a subsequent floating-point instruction.

The address of the instruction that generated the inexact result is available to the trap handler in the FPIAR. The trap handler can determine the location of the operand(s) by examining the instruction. In the case of a memory destination, the evaluated effective address of the operand is available in the MPU mid-instruction stack frame (at offset +$10). When an FSAVE is executed by an inexact trap handler, the value of the exceptional operand in the state frame is not defined (refer to **6.4.2 State Frame**). An inexact exception differs from the other exceptions in this respect. If an inexact condition is the only exception that occurred during the execution of an instruction, the value of the exceptional operand is invalid. If multiple exceptions occur during an instruction, the exceptional operand value is related to a higher priority exception.

Note that the trap handler should use only the FMOVEM instruction to read or write the floating-point data registers, since FMOVEM cannot generate further exceptions or change the condition codes.

NOTE

The IEEE standard specifies that inexactness should be signaled on overflow as well as for rounding. The FPCP implements this via the INEX bit in the FPSR AEXC byte. However, the standard also indicates that the inexact trap should be taken if an overflow occurs with the overflow trap disabled and the inexact trap enabled. Therefore, the FPCP takes the inexact trap if this combination of conditions occurs, even though the INEX1 or INEX2 bits may not be set in the FPSR EXC byte. In this case, INEX is set in the AEXC byte and OVFL is set in both the EXC and AEXC bytes.

6.1.8 Inexact Result on Decimal Input

In a general sense, inexact result 1 (INEX1) is the condition that exists when a packed decimal operand cannot be converted exactly to extended precision in the current rounding mode. If this condition occurs, the INEX1 bit is set in the FPSR exception status byte, and the result of the decimal-to-binary conversion is rounded to extended precision (regardless of FPSR mode byte rounding precision) as shown in Figure 6-3. The FPCP provides two inexact bits (INEX1 and INEX2) to help distinguish between inexact results generated by decimal input conversions (INEX1) and other inexact results (INEX2).

Trap Disabled Results: If the instruction is an FMOVE to a floating-point data register, the rounded result is stored in the floating-point data register. If the instruction is not an FMOVE, the rounded result is used in the calculation.

Trap Enabled Results: The result is generated in the same manner as if traps were disabled, except that a take exception primitive is returned when the MPU reads the response CIR of the FPCP. Since the MC68881 does not allow multiple floating-point concurrency, the exception is always reported as a pre-instruction exception when the next floating-point instruction is attempted. The MC68882, however, may report an exception as a mid-instruction exception on a subsequent floating-point instruction.

The address of the instruction that caused the inexact decimal conversion is available to the trap handler in the FPIAR. The trap handler can determine the location of the decimal string by examining the instruction, although the effective address of the string must be recalculated (if possible) by the trap handler. When an FSAVE is executed by an inexact trap handler, the value of the exceptional operand in the state frame is not defined (refer to **6.4.2 State Frame**). An inexact exception differs from the other exceptions in this respect. If the inexact conversion is the only exception that occurs during the execution of an instruction, the value of the exceptional operand is invalid. If multiple exceptions occur during an instruction, the exceptional operand value is related to a higher priority exception.

Note that the trap handler should use only the FMOVEM instruction to read or write the floating-point data registers, since FMOVEM cannot generate further exceptions or change the condition codes.

6.1.9 Multiple Exceptions

Dual and triple instruction exceptions may be generated by a single instruction in a few cases. When multiple exceptions occur with traps enabled for more than one exception class, only the highest priority exception trap is taken; the other enabled exceptions do not cause a trap. The higher priority trap handler must check for multiple exceptions. The priority of the traps is as follows:

BSUN	Highest Priority
SNAN	
OPERR	
OVFL	
UNFL	
DZ	
INEX2/INEX1	Lowest Priority

The multiple instruction exceptions that can occur are:

SNAN and INEX1
OPERR and INEX2
OPERR and INEX1
OVFL and INEX2 and/or INEX1
UNFL and INEX2 and/or INEX1
INEX2 and INEX1

6.1.10 IEEE Exception and Trap Compatibility

The IEEE standard defines only five exceptions. The FPCP FPSR AEXC byte contains bits representing these five exceptions, which are defined to function exactly as the standard specifies the exceptions. However, it may be more useful to differentiate the IEEE required exceptions into the eight exceptions represented in the FPSR EXC byte. Since the FPCP uses the bits in the FPSR EXC byte and the FPCR ENABLE byte to determine when to trap, there are seven possible instruction traps defined (INEX1 and INEX2 share one exception vector) instead of the five defined by the standard.

If it is necessary to write an application program that only supports the five IEEE specified traps, the BSUN, SNAN, and OPERR exception vectors should be set to point to the same handler routine. This allows the FPCP to support the invalid operation exception defined in the IEEE standard, which is represented by the invalid operation (IOP) bit in the AEXC byte.

To satisfy other requirements in the IEEE standard, the FPCP does the following:

1. A one is ORed into the AEXC byte IOP bit if the BSUN, SNAN, or OPERR bit is set in the EXC byte.
2. A one is ORed into the AEXC byte UNFL bit only if both the UNFL and the INEX2 bits of the EXC byte are set. However, per the IEEE standard, the underflow trap is based only on the UNFL bit in the EXC byte,.
3. A one is ORed into the AEXC byte INEX bit if the INEX1, INEX2 or OVFL bit is set in the EXC byte.
4. The IEEE standard requires that an inexact trap be taken if it is enabled, an overflow occurs, and the overflow trap is disabled. Thus, if the OVFL bit is set in the EXC byte, the OVFL bit is not set in the ENABLE byte, and the INEX2 bit is set in the ENABLE byte, then the inexact trap is taken.

The equations for items 1, 2, and 3 are:

AEXC(IOP) = AEXC(IOP)vEXC(BSUNvSNANvOPERR)
AEXC(UNFL) = AEXC(UNFL)vEXC(UNFLΛINEX2)
AEXC(INEX) = AEXC(INEX)vEXC(INEX1vINEX2vOVFL)

The equation for item 4 (inexact trap taken) is:

Inexact Trap =
[[EXC(OVFL)vEXC(INEX2)]ΛENABLE(INEX2)]v[EXC(INEX1)ΛENABLE(INEX1)]

where:

"v" = logical OR
"Λ" = logical AND

6.1.11 Illegal Command Words

Illegal coprocessor commands are coprocessor command word bit patterns that are not implemented by the FPCP. The FPCP reports illegal coprocessor commands as pre-instruction exceptions, using the F-line emulator vector number. The specific illegal command word bit patterns are defined in **4.7 INSTRUCTION ENCODING DETAILS.**

FPCP instructions consist of an operation word, a coprocessor command word (if any), and extension words (if any). The MPU detects an illegal operation word and the FPCP detects an illegal command word.

For the case where a coprocessor-detected instruction trap is pending when the MPU writes an illegal coprocessor command to the FPCP command CIR, the coprocessor first reports the pending instruction exception as a pre-instruction exception. Following exception processing of the instruction exception, the MPU resumes execution of the main program at the beginning of the illegal coprocessor command, by writing to the command CIR again. The illegal instruction exception is then reported by the FPCP.

6.1.12 Coprocessor-Detected Protocol Violation

All interprocessor communications in the coprocessor interface occur as standard M68000 bus cycles. A failure in this communication results in the FPCP reporting a mid-instruction exception with the coprocessor protocol violation vector number. When a protocol violation has been detected by the FPCP, the response CIR is encoded to the take mid-instruction primitive and the next read of the response CIR by the main processor terminates the dialog.

The MC68881 signals a protocol violation when unexpected accesses of the command, condition, register select, or operand CIRs occur. Coprocessor detected protocol violations occur when:

1. The MC68881 is expecting a write to the command or condition CIR, and instead an access of the register select or operand CIR occurs.
2. The MC68881 is expecting a read of the register select or operand CIR, and instead a write to the command, condition, or operand CIR occurs.
3. The MC68881 is expecting a write to the operand CIR, and instead either a write to the command or to the condition CIR or a read of the register select or of the operand CIR occurs

The MC68882 signals a protocol violation when unexpected accesses of the command, condition, register select, operand, or instruction address CIRs occur. For the MC68882, coprocessor-detected protocol violations occur when:

1. The MC68882 is expecting a write to the command or condition CIR, but a read or write operation to the register select CIR or to the operand CIR or a write operation to the instruction address CIR occurs instead.
2. The MC68882 is expecting a read of the register select CIR or of the operand CIR, but a write operation to the command CIR, the condition CIR, the operand CIR, the instruction address CIR, or the register select CIR occurs instead.
3. The MC68882 is expecting a write operation to the operand CIR, but a write operation to the command CIR or to the condition CIR or a read of the register select CIR, the operand CIR, or instruction address CIR occurs instead.
4. The MC68882 is expecting a write operation to the instruction address CIR, but a write operation to the command CIR, the condition CIR, the operand CIR, or the register select CIR or a read of the operand CIR or the register select CIR occurs instead.

For these violations, the FPCP maps the 16-bit register select CIR onto the upper word of the 32-bit operand register. Thus, inconsistent data is read from the operand CIR, and write cycles cannot store the correct value. Of course, this is of no consequence since the protocol violation invalidates any operation being attempted by the FPCP or the main processor.

During normal operation, the FPCP synchronizes interprocessor communication by delaying the assertion of DSACKx, if necessary. However, upon detection of a protocol violation, the MC68881 always terminates the access by immediately asserting DSACKx.

Note that in certain cases resulting from serious system programming errors, an unrecoverable protocol violation may occur when using the MC68882. This particular case of the protocol violation occurs during the coprocessor interface dialog for the FMOVE and FMOVEM instructions if a read of the operand CIR occurs before the evaluate <ea> and transfer data (DR=1) or the transfer multiple coprocessor registers (DR=1) primitive is issued. In this case, the protocol violation is not reported via the take mid-instruction primitive as is the normal case. Instead, the MC68882 ignores the access completely, and it is the responsibility of the system watchdog timer to abort the access to the operand CIR by asserting the bus error signal to the main processor. The MC68020 and MC68030 cannot cause this protocol violation to occur except through misuse of the MOVES instruction.

Spurious coprocessor-detected protocol violations may also be caused by hardware timing design errors. When using an MC68020 or MC68030, a common oversight is to use a buffered address strobe ($\overline{AS}$) to the FPCP while not buffering the lower address lines supplied to the FPCP. If the $\overline{AS}$ buffer delay used is long enough, it is possible to violate FPCP specification #7. This problem is usually indicated by protocol violations during FMOVEM instructions.

A protocol violation cannot occur as a result of an access to the reserved register locations, a read of a write-only register, or a write to a read-only register (a read of a reserved or write-only register always returns a value of all ones). One exception to this rule is that a write access to the register select CIR causes a protocol violation. Reads of the save or response CIR are always valid as are writes to the restore or control CIR.

While the MC68881 can request that the MPU write the instruction address CIR (by setting the PC bit in a primitive response), accesses of this register are neither expected or unexpected. Thus, when the MC68881 is utilized as a peripheral processor where no concurrent instruction execution occurs, requests to transfer the PC may be ignored without incurring a protocol violation. When the instruction address CIR is written by the main processor, the MC68881 updates the FPIAR with the written value without regard to "correct" protocol.

Since the MC68882 provides concurrent execution of multiple floating-point instructions, it requires program counter values to be transferred when requested to guarantee a valid FPIAR for a concurrently-executed instruction which reports an exception. Whenever the MC68882 requests the PC value, it reports a protocol violation if the main processor does not transfer the PC value by writing the instruction address CIR.

A protocol violation is the highest priority coprocessor-detected exception. It is also considered to be a fatal exception, since the MPU acknowledgment of the protocol violation exception clears any pending FPCP instruction exceptions and aborts any instruction in progress.

NOTE

To distinguish between a protocol violation detected by the MPU or the FPCP, an exception handler can read the response CIR and evaluate the returned primitive. If the protocol violation is detected by the FPCP due to an unexpected access, the operation being executed previously is aborted, and the FPCP assumes the idle state when the exception acknowledge is received. Therefore, the primitive read from the response CIR is null (CA=0). If the protocol violation is detected by the MPU due to an illegal primitive, the FPCP response CIR contains that primitive when the exception handler reads it. (Since the FPCP cannot internally generate an illegal primitive, an MPU detected protocol violation indicates a hardware failure.)

To read the response CIR in a hardware independent manner, the trap handler should use the move alternate address space (MOVES) instruction. For example, the following instruction sequence reads the response CIR of the coprocessor with CPID=1 into an MPU data register:

```
MOVE.B    #7,D0          Prepare the SFC register
MOVEC     D0,SFC         for a CPU space cycle. . .
MOVES.W   $00022000,D0   Execute a "coprocessor" cycle.
```

6.1.13 Recovery from Exceptions

When a coprocessor-detected exception occurs, enough information is made available to the trap handler to perform the necessary corrective action and then resume execution of the program that caused the exception. Of course, in some instances, it may not be valid to resume execution of the program; recovery is not possible for protocol violations. The information available to an exception handler is described in the previous sections, and the following paragraphs describe the methods used to resume execution of a program after an exception is appropriately handled.

In all cases, the stack frame generated by the MPU in response to a coprocessor-detected exception contains a program counter value that points to the instruction to be executed

upon return from the exception handler. In the case of pre-instruction exceptions, the instruction to be executed upon return is the FPCP instruction that was attempted, but preempted by a pending exception. For mid-instruction exceptions (other than interrupts), two pointers are saved: the address of the FPCP instruction that caused the exception and the address of the instruction immediately following that FPCP instruction. Furthermore, the FPIAR contains a pointer to the FPCP instruction that caused the exception in both cases. Thus, an exception handler can always locate the instruction that caused an exception, and identify the next instruction to be executed upon return from the handler.

When the MPU executes a return from exception (RTE) instruction, it reads the stack frame from the top of the active system stack and restores that context. In the case where the stack frame was generated by an FPCP pre-instruction exception, the context that is restored is the MPU context of the pre-empted FPCP instruction. The FPCP instruction begins execution in the normal manner, with the MPU writing the coprocessor command word to the FPCP.

In the case where the RTE stack frame is generated by a coprocessor-detected mid-instruction exception, the context restore operation is slightly different. In this case, the MPU must complete execution of the instruction that was suspended by the exception. When the RTE instruction completes execution, the MPU first reads the response CIR of the FPCP to determine the next appropriate action.

NOTE

Since the MC68881 always finishes execution of the instruction that causes this type of exception before reporting it, the response that is returned is null (CA = 0, PF = 1), which releases the main processor to continue with the execution of the next instruction. Note that after a take mid-instruction exception primitive is returned, the main processor is not required by the MC68881 to perform a read from the response CIR before initiating the next floating-point instruction, but the MPU always performs this action when processing a mid-instruction stack frame.

An MC68881 arithmetic exception handler (i.e., a handler for any exception other than the BSUN exception) routine is not required to perform any action to clear the cause of an exception. In fact, an MC68881 arithmetic exception handler may consist of a single RTE instruction (which produces the same logical effect as disabling an exception). This is because the main processor acknowledges the exception by writing to the control CIR when the coprocessor signals an exception to the MPU, and the exception acknowledge clears any pending exceptions in the MC68881. Thus, the MC68881 arithmetic exception handler is not required to clear any status bits or read any MC68881 registers in order to prevent the reocurrence of an exception when an RTE instruction is executed. However, an RTE instruction alone does not prevent the reoccurrence of an MC68882 exception. The MC68882 does not clear the pending floating-point exception in response to the exception acknowledge. An MC68882 arithmetic exception handler must meet certain requirements in order to clear the cause of the exception. Refer to **5.2.2 Exception Handler Code** for the MC68882 exception handler requirements. In the case of the BSUN exception handler, some action must be taken (as described in **6.1.1 Branch/Set on Unordered (BSUN)** by the exception handler to avoid an infinitely executing loop.

For the MC68881, if an exception handler includes any FPCP instruction other than an FMOVEM, an FSAVE should be the first FPCP instruction to be executed. This assures that an exception handler cannot generate any exceptions related to, or modify the context of,

the program that caused the exception. For the MC68882, all exception handlers must begin with an FSAVE instruction, even when they do not contain any floating-point instructions. The FPIAR value must be saved before any instruction other than an FMOVEM is executed, so that the address of the instruction that caused the exception is not lost. When the exception handler completes the error recovery and is prepared to return to the suspended program, an FRESTORE is executed as the last FPCP instruction; this restores the previous context of the program that caused the exception. Refer to **5.2.2 Exception Handler Code** for other requirements of the MC68882 exception handler.

6.2 MAIN PROCESSOR DETECTED EXCEPTIONS

The following paragraphs describe exceptions that are detected by the MPU during FPCP instruction execution. Refer to the main processor user's manual for additional information on these exceptions, and the pre- and mid-instruction exception main processor stack frames.

6.2.1 Trap on Coprocessor Condition Instruction

The FPCP trap on condition instruction is initiated when the MPU writes a conditional predicate to the FPCP for evaluation and reads a true/false result in the FPCP response primitive. If the FPCP indicates that the condition is true, the MPU takes a post-instruction exception using the TRAPV/TRAPcc vector number.

The stack frame generated by the MPU in response to this exception contains two pointer values:

1. A pointer to the FTRAPcc instruction that caused the exception
2. A pointer to the instruction that follows the FTRAPcc (the pointer to which the processor returns if an RTE instruction is executed)

6.2.2 Illegal Instructions

The FPCP instructions consist of an operation word, a coprocessor command word (if any), and extension words (if any). The MPU detects illegal operation words, and the FPCP detects illegal command words. When the MPU detects an illegal operation word for a coprocessor instruction, it takes a pre-instruction exception using the F-line emulator vector number. Refer to **4.7 INSTRUCTION ENCODING DETAILS** for specific bit patterns that are illegal coprocessor operation words.

In addition to detecting an illegal operation word, the MPU can detect an illegal instruction even though the operation word is valid. This occurs when the addressing mode of the instruction is not valid. When the FPCP returns a primitive response to the MPU that requests a data transfer to or from the effective address, the FPCP either implicitly or explicitly indicates the valid addressing modes for an instruction. Thus, the MPU can determine that properly formed FPCP operation words and primitive responses are invalid if they specify operations that are illegal, such as writing to a nonalterable effective address.

When the MPU detects an invalid instruction in this manner, it terminates the FPCP execution of the instruction by writing an abort to the control CIR. (The MC68882 only aborts

the instruction with the invalid effective address without disturbing concurrently-executed instructions. The MPU then takes a pre-instruction exception using the F-line emulator vector number. Termination of the FPCP instruction execution in this manner does not alter any visible processor or coprocessor registers or status (such as pending coprocessor exceptions). Use of the F-line emulator trap allows the operating system to emulate any extensions to the FPCP that are not supported by a specific processor.

6.2.3 Main-Processor-Detected Protocol Violations

If the MPU reads an FPCP response primitive that it interprets as an illegal primitive, it does not terminate the FPCP execution of the instruction by writing to the coprocessor interface control register. Instead, the MPU takes a mid-instruction exception using the coprocessor protocol violation vector number.

Since the FPCP never issues an illegal response primitive, this feature of the MPU serves to detect a failure of interprocessor communications. If a protocol violation is taken on an FPCP instruction, whether detected by the FPCP or the MPU, a system failure may be assumed. Refer to **6.1.12 Coprocessor-Detected Protocol Violation** for an example of how an exception handler can determine the cause of a protocol violation.

6.2.4 Trace Exceptions

To aid in program development, the MPU includes a facility to allow instruction-by-instruction tracing. In the single-step trace mode, after each instruction is executed, the MPU takes a post-instruction exception using the trace vector number. This allows a debugging program in the trace exception handler to monitor the execution of a program under test. Refer to the main processor user's manual for a complete description of the trace mode.

Many FPCP instructions can operate concurrently with MPU instructions, and defer the reporting of coprocessor detected instruction exceptions until the next FPCP instruction is dispatched by the MPU. This provides a sequential instruction execution model even though concurrent instruction execution may occur. To guarantee that pending exceptions are always reported at the same point in an instruction sequence, regardless of whether tracing is enabled, the FPCP always releases the MPU at the end of an instruction that allows concurrency before reporting the exception. This sequence is important, because the MPU (when in the trace mode) waits for an instruction to complete before proceeding.

To provide consistent reporting of exceptions, the FPCP always returns the null (CA = 0, PF = 1) primitive when it completes execution of an instruction that allows concurrency, and then reports a pending exception only after a write to the command or condition CIR.

The synchronization of the two devices in the trace mode is accomplished through the PF bit in the null primitive (see **7.1 CHIP-SELECT DECODE**). When the trace mode is enabled, the MPU repeatedly reads the response CIR to determine when the FPCP completes instruction execution. If the null (CA = 0, IA = 1, PF = 0) primitive is read, then the MPU checks for pending interrupts, and if none are pending, reads the response CIR again. This process continues until the MPU receives a null (CA = 0, PF = 1) primitive from the FPCP, at which time it performs the trace exception processing.

In order for a trace exception to be transparent to normal program execution, the trace handler routine must take certain precautions to prevent disturbing the context of the FPCP. When the main processor detects an exception, it automatically saves the most volatile portion of the current context and processes the exception immediately; thus, the trace handler routine is not required to perform any MPU context save in order for the system to operate properly. The FPCP does not operate in this manner, since it cannot initiate exception processing until the MPU attempts to execute a new floating-point instruction. Also, the context information that must be saved for the FPCP is more extensive than that of the main processor; thus, the software must perform the save only when necessary. The important consideration for a trace exception handler is that it must perform a more extensive context save for the FPCP than for the MPU (since part of the MPU context save is automatic). Also, it should not execute any FPCP instruction that may cause a pending exception to be reported, or a new exception to occur.

Because of these constraints, the first and last FPCP instructions of a trace exception handler should be the FSAVE and FRESTORE instructions, respectively. By executing the FSAVE instruction before any other floating-point instruction, the FPCP saves any pending exceptions in a state frame and then clears them internally; thus, an exception generated by the main program cannot be reported while the trace exception handler is executing. After the FSAVE instruction is executed, the FMOVEM instruction can be used to save the user-visible portion of the FPCP context. Then the trace handler is free to utilize the coprocessor as desired, without affecting the main program context. When the trace handler is ready to return to the main program, the FMOVEM instruction is used to restore the user-visible context, followed by an FRESTORE instruction to reinstate the exact context of the FPCP prior to the trace exception processing. Note that since the MPU is forced to wait until the completion of an FPCP instruction before processing a pending trace exception, the execution of the FSAVE instruction by the trace handler always results in an idle state frame being saved. The user-visible registers contain the results of the last floating-point instruction. This would not be the case if the trace exception handler were allowed to begin execution before the FPCP instruction is completed. Processors other than the MPU must implement the trace synchronization mechanism in software (by polling the PF bit) in order to assure these conditions.

6.2.5 Interrupt

When the FPCP is busy executing an instruction, it may issue a null (CA=1, IA=1) primitive response, which requests the MPU to continue polling the response register. (This only occurs if the FPCP requires additional services from the MPU for the current instruction.) This response also indicates to the MPU that it may sample interrupts between reads of the response CIR. If there is no interrupt pending, the MPU reads the response CIR again. If there is an interrupt pending, the MPU takes an interrupt exception using the mid-instruction stack frame. Upon exiting from the interrupt handler, the MPU repolls the FPCP response CIR to continue the suspended instruction dialog.

In the trace mode, an interrupt can temporarily break the synchronization of the MPU and the FPCP. This can occur when the MPU receives a null (CA=0, IA=1, PF=0) primitive. In this case, the MPU checks for interrupts before reading the response CIR again; if an interrupt is pending, the interrupt exception is processed immediately. In response to the interrupt, the MPU saves a 10-word mid-instruction stack frame with the trace pending status saved as part of the previous context information. When the interrupt handler completes execution and performs an RTE instruction, the MPU returns to the trace pending

mode and reads the response CIR to determine if the previous coprocessor instruction is completed. In this manner, the exception processing for the interrupt is completely transparent to the handling of the trace exception by the MPU and FPCP pair.

If an interrupt handler for a system using an FPCP requires the use of the FPCP, or if a task switch requires that the context be saved, an FSAVE instruction should be the first floating-point instruction executed by the routine. To restore the original context, an FRESTORE must be executed by the routine before the RTE instruction. If an interrupt handler does not interact with the FPCP, no context save operations are required.

Many FPCP instructions require a fairly long time to execute, and the MPU may be forced to wait until the FPCP execution is complete before proceeding to the next instruction (either because the instruction does not allow concurrency or the main processor is in the trace mode). Normally, the MPU can only process pending interrupts when it reaches an instruction boundary, but this might adversely affect interrupt latency if it is not allowed to process interrupts while waiting on the FPCP. To reduce interrupt latency as much as possible, the FPCP always sets the interrupts allowed (IA) bit in the null (CA = 1) and null (CA = 0, PF = 0) primitives; thus allowing interrupts to be processed while the MPU is waiting for the coprocessor to complete an operation. In fact, most FPCP instructions, regardless of their overall execution time, provide for very small interrupt latency times. The worst-case interrupt latency instruction for the FPCP is the FRESTORE with a busy state frame (see **8.3 INTERRUPT LATENCY TIMES** for more information).

6.2.6 Address and Bus Errors

Bus cycle faults may occur while processing FPCP instructions during the MPU accesses of the coprocessor interface registers, or during memory cycles run by the MPU to access instructions or data. If the MPU receives a fault while running the bus cycle which initiates an FPCP instruction (i.e., the initial write to the command or condition CIR), it assumes that no FPCP is present in the system, and takes a pre-instruction exception using the F-line emulator vector number. Thus, an MPU system may utilize software emulation of the FPCP or provide hardware floating point, and the actual configuration is transparent to the application program. If any other access to the FPCP is faulted, it is assumed that the coprocessor has failed, and the MPU takes a bus error exception.

If the MPU has a memory fault while executing an FPCP instruction, it takes an address error or bus error exception. After the fault handler corrects the fault condition, it may return and communication with the FPCP continues as if the fault had not occurred.

6.2.7 Privilege Violations

The MPU operates at one of two privilege levels: the user level or the supervisor level. The privilege level determines which operations are legal, and the S bit in the MPU status register determines the privilege level. Most programs execute at the user level where accesses are controlled, and effects on other parts of the system are limited. The operating system executes at the supervisor level, has access to all resources, and may execute all instructions; hence, it performs the overhead tasks for the user-level programs.

The FPCP FSAVE and FRESTORE instructions are privileged instructions; all others are nonprivileged. An attempt to execute the FSAVE or FRESTORE instructions while at the

user privilege level results in the MPU taking a pre-instruction exception using the privilege violation vector number.

6.2.8 Format Error Exceptions

When the FRESTORE instruction is executed, the FPCP checks the validity of the format word written to the restore CIR by the MPU. Refer to **6.4.2 State Frames** for information on the format word. The FPCP returns an invalid format word ($02XX) in the restore CIR when the format word from MPU is not valid. The MPU then takes a pre-instruction exception using the format error vector number. Refer to **7.5.4.7 FORMAT EXCEPTION, FRESTORE INSTRUCTION** for further information on the FRESTORE format error exception.

When an FSAVE instruction is initiated while the FPCP is executing a previous FSAVE or FRESTORE instruction, the FPCP returns an invalid format word ($02XX) in the save CIR. The MPU then takes a pre-instruction exception using the format error vector number. Refer to **7.5.4.6 FORMAT EXCEPTION, FSAVE INSTRUCTION** for further information on the FSAVE format error exception.

6.3 MC68882 EXCEPTION HANDLERS

MC68882 exception handlers can be derived by modifying existing MC68881 handlers. The required modifications are discussed in **5.2.2 Exception Handler Code.** Note that if the guidelines in the referenced text are met, the resulting MC68882 handlers can be used with no adverse effects for systems that use the MC68881. Since the MC68882 is pin-compatible and user-software-compatible with the MC68881, the exception handlers can be written to meet the system software requirements of both the MC68881 and the MC68882. When this is done, systems that only use the MC68881 at present can replace the MC68881 with the MC68882 using the same socket, without changing either applications or systems software.

6.4 CONTEXT SWITCHING

In most types of multitasking systems, it is often necessary to take control from one program and give control to another program. This requires the operating system to extract (from the FPCP) data corresponding to one program context and load the context corresponding to the next program to be executed. The information that must be exchanged is divided into two categories:

1. Programmer's model consists of data accessible by the programmer using nonprivileged instructions. This data is saved and restored using the FMOVEM instructions.

2. Internal state consists of various internal flags and registers that are vital in restoring the FPCP to the proper state. The application program need not be concerned with the internal state. These internal flags and registers are accessed by the privileged FSAVE and FRESTORE instructions.

The following paragraphs describe how this context information is manipulated.

6.4.1 FSAVE and FRESTORE Instruction Overviews

The basic mechanism for performing a context switch on the FPCP is provided by the FSAVE and FRESTORE instructions. These instructions are a logical extension to the instruction continuation mechanism that is used by the MC68010, MC68020, and MC68030 processors to support virtual memory. The FSAVE instruction is treated much like a microcode-level interrupt to the FPCP, instructing it to suspend any operation that is being executed (at the earliest possible boundary) and make a complete copy of the internal state of the machine into memory. This is similar to the effect of the assertion of bus error to the main processor. To restore the internal state saved by the FSAVE instruction, the FRESTORE instruction is used, which is similar to the RTE instruction on the main processor.

The internal state information that is stored in memory by the FSAVE instruction contains the image of the flags and registers not visible to the user, including the address in the microprogram counter, temporary register values, and pending exception information. After the execution of an FSAVE, the FPCP enters the idle state, and any pending exceptions are cleared. To perform a complete context save, FMOVEM instructions must be used to save the user-visible portion of the machine; and then a new context may be loaded. When it is necessary to reload the context that was previously saved, these steps are reversed: first, the FMOVEM instructions load the user-visible context, followed by an FRESTORE instruction, which loads the nonuser-visible context. After the execution of the FRESTORE, the FPCP returns to the exact context that existed just before the FSAVE instruction was executed, and execution continues from that point.

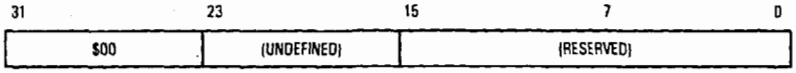
Depending on the state of the FPCP when an FSAVE instruction is executed, the format of the internal state information written to memory may be in one of three forms: idle, null, or busy. Also, the FPCP may force the MPU to wait for a short time while the internal state is prepared for the save operation. During execution of an FRESTORE instruction, the FPCP interprets the state information read from memory (and written to the restore CIR) to determine the appropriate response action. The FRESTORE is destructive in that the FPCP immediately stops any operation that it may be performing and begins to load the next context; thus there is no need for a mechanism in the FRESTORE instruction to allow the FPCP to make any service requests to the MPU. The protocol of the FSAVE and FRESTORE instructions is described in detail in a subsequent paragraph.

6.4.2 State Frames

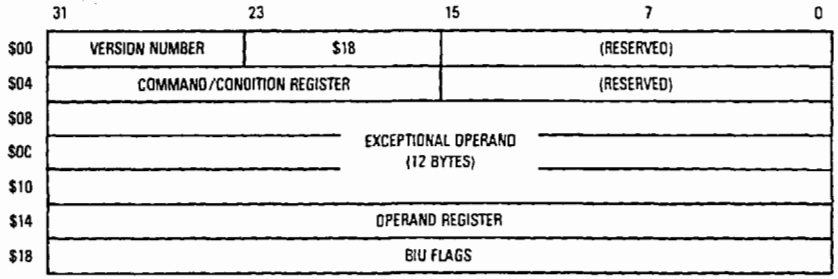
The three state frame formats that are generated by the MC68881 are shown in Figure 6-4. In all three state frames, the first long word of the frame has the same format. The least significant word of this long word is reserved for future definition by Freescale; it is included to allow long word alignment of a state frame in memory. The most significant word of the first long word (called the format word) contains the version number of the coprocessor that generated the state frame (in the most significant byte) and the size of the internal state stored in the frame (in the least significant byte). For the null state frame, the size value is undefined. Although the version number and frame size values are defined by the MC68881, the M68000 Family coprocessor interface defines the null format word which is the one format word value that must be recognized by any coprocessor as described in a subsequent paragraph.

Two of the state frame formats for the MC68882, shown in Figure 6-5, differ from the corresponding state frames for the MC68881 in two respects. First, the idle and busy state

NULL STATE FRAME



IDLE STATE FRAME



BUSY STATE FRAME

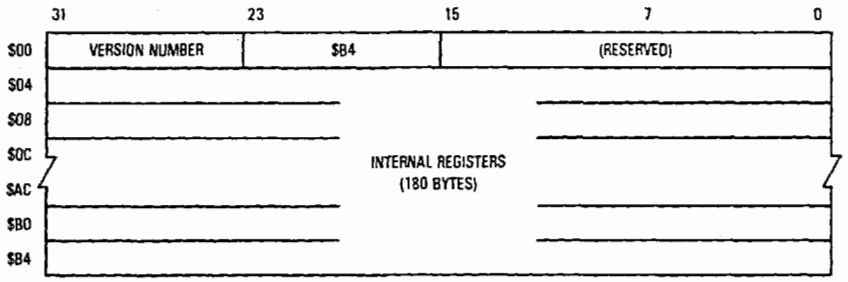


Figure 6-4. MC68881 State Frame Formats

frames each contain 32 additional bytes, which store the CU internal state. Second, the saved CU internal state is saved at the top of the frame, immediately following the format word. This results in offsets to the APU information that are greater than those for corresponding data in the MC68881 state frames by $20. The null state frame consists only of the format word in both coprocessors.

When an FSAVE instruction is executed, the format word is the first data item transferred to the MPU, and the main processor uses the size value to perform the correct address

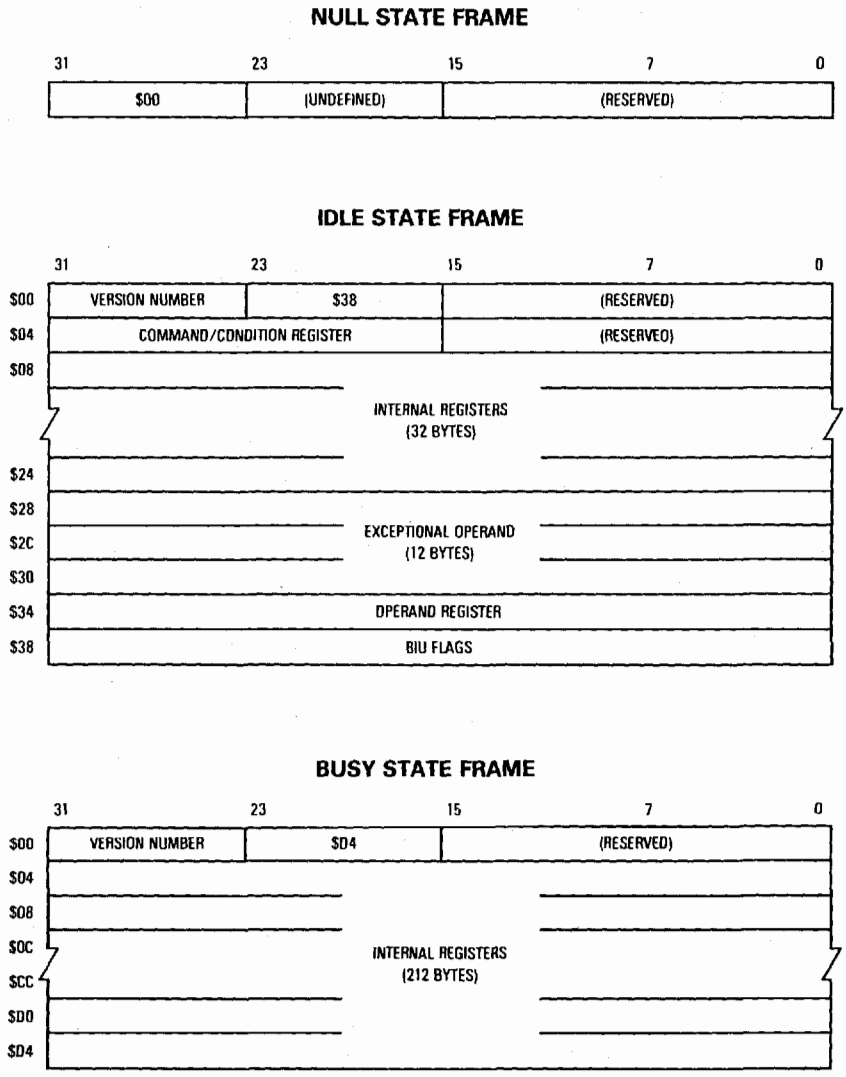


Figure 6-5. MC68882 State Frame Formats

calculations. During an FRESTORE instruction, the format word is written to the FPCP to initiate the restore operation. When this occurs, the FPCP checks the version number and frame size values for validity and signals a format exception if they are not valid for this particular device. The version number is an 8-bit value that identifies the microcode version of the FPCP, and the format of this number is defined internally by the FPCP. Future devices will use a unique combination of the version number and frame size values in order to guarantee that various revisions of the device cannot incorrectly utilize an internal state frame that is not valid for that revision.

In addition to being used by the FPCP to validate a state frame before it is used in a restore operation, the format word can be used by a user program to identify the format of a state frame and the saved state of the FPCP. In the following descriptions of the three state frames, the data format within a frame is guaranteed only for those version number and frame size values given in the accompanying tables. Routines that utilize state frame information must examine the format word to correctly identify any data formats that are subject to change by Freescale.

NOTE

The state size value in the format word indicates the size (in bytes) of the FPCP internal state information. This size value does not include the format word or the reserved word.

6.4.2.1 NULL STATE FRAME. As shown in Figures 6-4 and 6-5, no internal state information is saved in the null state frame. Only the coprocessor version number (0) is indicated. Version number 0 is a wild card number, allowing this state frame type to be restored to a coprocessor of any version. The size value of a null state frame is not assumed to be valid during a save operation and is ignored by the FPCP during a restore operation. A restore of the null state performs the reset function with all floating-point data registers loaded with nonsignaling NANs and with the FPCR and FPSR set to zero. A save of the null state results when no FPCP instructions have been executed since the last null state restore or hardware reset. Note that a save of a null state indicates that the FPCP programmer's model is empty, and thus does not need to be saved with a FMOVEM instruction.

6.4.2.2 IDLE STATE FRAME. As shown in Figure 6-4, 24 bytes of internal state are saved in the idle state frame for the MC68881. For the MC6882, the idle state frame consists of 56 bytes (see Figure 6-5). The format word indicates the coprocessor version number and state size (24 or 56 bytes in addition to the format word). An idle state frame is produced if an FSAVE occurs when a floating-point instruction is not being executed, or when the current instruction is in the end phase (refer to **6.4.3.5 END PHASE** for a definition of the end phase).

In addition to being used for context switching, the idle state frame contains information that is useful to most floating-point exception handlers. First, it contains the exceptional operand value, which can be evaluated by an exception handler to determine the cause of an exception. Second, it contains the BIU flag word that indicates the status of the FPCP at the time of an FSAVE instruction. For example, this information can be used by the trace exception handler in a debug monitor to display the pending exception status along with the register state of the machine.

As shown in Figure 6-4, the idle state frame for the MC68881 contains four data items: the command/condition register image, the exceptional operand, the operand register image, and the BIU flags. A reserved word is also included in order to align the state frame to a long-word boundary; it is written as $FFFF and ignored during restore operations. The command/condition word and operand register may contain temporary information, as indicated by the BIU flags.

The idle state frame for the MC68882 contains 32 words of CU internal register and state information between the command condition register and the exceptional operand. It is otherwise identical to the idle state frame for the MC68881.

The format of the BIU flag word is shown in Figure 6-6. Only the 16 most significant bits in the BIU flag word are defined; the undefined bits are written as ones during save operations and ignored during restore operations.

The definitions of the 16 flag bits are:

Bits 16–19 — These bits contain internal state information about the CU and should not be modified.

Bits 20–23 — These bits are set when valid data is contained in the operand register image of the state frame. There is one flag bit for each byte in the 32-bit operand register image; if a bit is one, there is valid data in the corresponding byte. If a bit is zero, the data in the corresponding byte is assumed to be invalid. These bits can be used to qualify the image of the operand register and should not be modified.

Bits 24–25 — These bits contain internal state information about the CU and should not be modified.

Bit 26 — This bit indicates that the FPCP has completed any necessary operand conversions and is ready to write an operand to memory. If this bit is a zero, an operand transfer to memory is pending. This bit should not be modified.

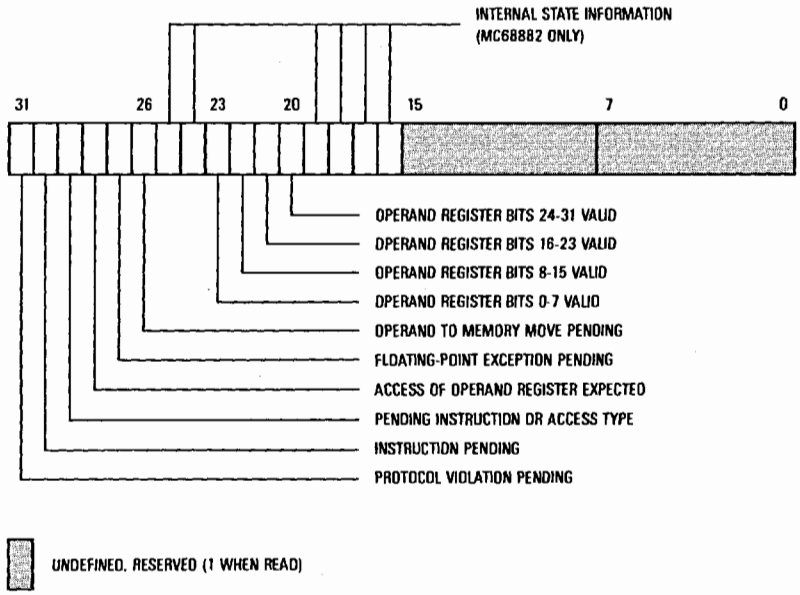


Figure 6-6. BIU Flag Format

Bit 27 This bit indicates that a floating-point exception is pending, which is reported when the MPU attempts to initiate the next floating-point instruction (after a FRESTORE of this state frame). If this bit is zero, an exception is pending, and the logical AND of the FPSR EXC and FPCR ENABLE bytes indicates the type of the pending exception. This bit may be read by an exception handler (particularly a trace routine) to determine the exception status of the FPCP. As described in a subsequent paragraph, a user program can modify this bit and the FPSR EXC and FPCR ENABLE byte images to create a software generated pending exception.

NOTE

This bit must be set by the exception handler immediately before an FRESTORE and RTE instruction. When this bit is not set in the exception handler, the MC68882 re-executes the handler.

Bit 28 This bit indicates that the FPCP is expecting the next coprocessor interface register access to be to the operand CIR. This bit is used by the BIU as part of the protocol violation checking hardware and should not be modified. If this bit is a zero, an access of the operand CIR is pending, and the state of bit 29 determines whether the expected access is a read or write cycle. Bits 28–30 combine to define the pending operation as listed in Table 6-4.

Bit 29 This bit defines the type of pending operand access that is expected or the type of pending operation that is saved in the command/condition register image. This bit should not be modified. Bits 28–30 combine to define the pending operation as listed in Table 6-4.

Bit 30 This bit indicates that the FPCP has received a new command word or conditional predicate from the MPU, but has not been able to begin execution of that operation. If this bit is zero, the command word or conditional predicate that was received is contained in the command/condition register images of the state frame. This bit should not be modified. Bits 28–30 combine to define the pending operation as listed in Table 6-4.

Bit 31 This bit indicates that a protocol violation has been detected by the FPCP, and the MPU has not responded with an exception acknowledge or abort operation. If this bit is a one, a protocol violation is pending. This bit should not be modified.

NOTE

The formats of the idle state frame and the BIU flags shown are for the initial production versions of the FPCP; this format is identified by the format word values ($1F18 and $3F18 for the MC68881, and $1F38 for the MC68882). Freescale reserves the right to utilize different state frame formats and format word values to support future revisions to the FPCP.

Table 6-4. BIU Flag Bit Definitions

30	29	28	Definition
0	0	0	(Undefined, Reserved)
0	0	1	Conditional Instruction Pending
0	1	0	(Undefined, Reserved)
0	1	1	General Instruction Pending
1	0	0	Write of Operand CIR Pending
1	0	1	(Undefined, Reserved)
1	1	0	Read of Operand CIR Pending
1	1	1	No Pending Instruction or Operand CIR Access

The only bit in the BIU flag word that can be modified by software is bit 27, the exception pending bit. If this bit is zero, an exception is pending and may be cleared by changing it to a one. Alternatively, the type of the pending exception can be changed by modifying the FPSR EXC byte and/or the FPCR ENABLE byte before executing an FRESTORE. Finally, if the pending exception bit is one (indicating that no exception is pending), it can be changed to make an exception pending; the type of exception pending is defined by the FPSR EXC and FPCR ENABLE bytes. In all of these cases, the change in the exception status takes effect when the state frame is utilized by an FRESTORE instruction.

The exception pending bit (referred to as EXC PEND) in the BIU flag word is the image of the exception pending signal internal to the FPCP. Normally, EXC PEND is negated by the FPCP execution unit when an instruction (other than an FMOVEM, FMOVE control register, FSAVE, FRESTORE) begins execution, and is asserted if an exception occurs during the instruction. The bus interface unit uses EXC PEND to determine the primitive response that is encoded in the response CIR after a write to the command or condition CIRs, or after the completion of the transfer of a floating-point operand to memory. If EXC PEND is true when an attempt is made to initiate an FPCP instruction (other than an FMOVEM, FMOVE control register, FSAVE, or FRESTORE), the response CIR is encoded to the take pre-instruction exception primitive (or the take mid-instruction primitive when the instruction in the CU is reporting an exception caused by the instruction in the APU); otherwise, the dialog for the instruction is started. If EXC PEND is true at the end of the move of a floating-point operand to memory, the response CIR is encoded to the take mid-instruction exception primitive; otherwise it is encoded to the null (CA = 0, PF = 1) primitive. The vector number that is encoded in the take exception primitive is determined by the state of the FPSR EXC and FPCR ENABLE bytes and corresponds to the highest priority exception that is enabled. The MPU responds to the take exception primitive by writing an exception acknowledge to the control CIR. When the MC68881 detects the exception acknowledge, it clears EXC PEND. However, the MC68882 does not clear the EXC PEND bit. It is the responsibility of the exception handler to clear EXC PEND, using the instructions listed in **5.2.2 Exception Handler Code**.

With this understanding of how EXC PEND (and its image in the BIU flag word) affects the operation of the FPCP, a programmer can make exceptions pending in the FPCP under software control. Or, conversely, a pending exception type may be changed or cleared if necessary.

6.4.2.3 BUSY STATE FRAME. As shown in Figure 6-4, 180 bytes of internal state are saved in the MC68881 busy state frame. The format word indicates the coprocessor version

number and state size (180 bytes). The busy state is produced if an FSAVE occurs when a floating-point instruction is in the initial or middle phase. Due to the volatile nature of the FPCP internal state during calculation, this state frame does not contain any information useful to applications programs, and the frame should not be modified in any way.

The MC68882 busy state frame contains 212 bytes, including 32 bytes of CU internal state information (refer to Figure 6-5). The format word contains the coprocessor version number and the state size. Otherwise the MC68882 busy state frame is identical to the busy state frame of the MC68881.

6.4.3 FSAVE Protocol

Table 6-5 lists five possible phases of the execution of a floating-point instruction that can apply at the time an FSAVE instruction is executed. For each phase, the table shows the response time and the state frame type.

Table 6-5. MC68881/MC68882 Responses to Save Command

Phase Name	Response Time	State Frame Type
Reset	Immediate	Null
Idle	Immediate	Idle
Initial	Immediate	Busy
Middle	Periodic	Busy
End	Delayed	Idle

When the MPU decodes an FSAVE instruction, it attempts to initiate a save operation in the FPCP by reading from the save CIR. If the FPCP is ready to perform the save, it responds with a valid state frame format word. The format word informs the MPU that the coprocessor is ready to transfer the state frame and also what size frame is to be saved. If the FPCP is not ready to begin the transfer of the state frame, it returns the come-again format word, forcing the MPU to wait. When the MPU receives the come-again format word, it checks for pending interrupts and processes them if necessary. Otherwise, it repeatedly reads the save CIR until a format word other than come again is returned. When the FPCP receives a valid format word, it reads the number of bytes indicated by the format word, four bytes at a time, from the operand CIR and writes them to memory.

The FPCP always returns one of five format words in the save CIR. Table 6-6 shows the five format word values and their meanings. In this table, the version number of the idle and busy format words corresponds to the version number of the initial production versions of the MC68881; future revisions of the device will utilize different version numbers to identify unique state frame formats. If the format of the idle or busy state frame of a future version of the FPCP differs from that of versions $1F and $3F for the MC68881 or $1F for the MC68882, **Freescale** will provide the new format information when the new version is available.

The come-again format word is returned by the FPCP to force the MPU to wait, as previously described. When the FPCP is ready to complete a save operation, it returns one of the other valid format words (null, idle, or busy) to the main processor and then transfers the

Table 6-6. MC68881/MC68882 Format Word Definitions

Format Word	Definition and Frame Size
$00xx*	Null State
$01xx*	Come Again
$02xx*	Illegal, Format Error
$vv18**	Idle State (MC68881)
$vvB4**	Busy State (MC68881)
$vv38**	Idle State (MC68882)
$vvD4**	Busy State (MC68882)

*The frame size byte for these format words is undefined for the M68000 Family coprocessor interface. The value encoded by each version is consistent; however, different versions are not guaranteed to use the same values.

**Each different version encodes a unique version number in "vv" while using the same frame size value.

appropriate state frame. The only time that the FPCP uses the illegal format word is when a read of the save CIR occurs while the FPCP is performing a state save or state restore. Normally, this only occurs when the execution of an FSAVE or FRESTORE instruction is suspended (e.g., due to a page fault during the save or restore operation) and an attempt is made to execute a new FSAVE instruction. If this happens, the illegal format word is returned to cause a format exception to be taken by the main processor. When the MPU receives the illegal format word, it writes an abort to the control CIR and initiates exception processing. In this case, the format error handler routine examines the instruction that was being executed when the format error occurred and can determine whether the second FSAVE instruction failed due to "nesting" of save or restore operations. Such an error is considered to be a catastrophic system error since the FPCP context is lost and cannot be recovered.

When the MPU receives an idle or busy format word, the bytes in the frame (four bytes at a time) are transferred from the operand CIR to memory. First, the format word is written to memory at the evaluated effective address. For the predecrement addressing mode, the value of the specified address register is saved in a temporary register, the size of the state frame is subtracted from the address register, and the format word is pushed to that address. (Thus, the required stack space is allocated before the save operation is started.) The state frame is then filled, from higher addresses to lower addresses, using the temporary register as a pointer. For the control alterable addressing modes, the format word is written to the specified address; then the address of the last word of the frame is calculated (in a temporary register) and the frame is filled from higher addresses to lower addresses. After the last byte of the state frame is written to memory, the FPCP is in the idle state with no pending exceptions, and the MPU executes the next instruction (it does not read the save or response CIR after the save operation).

The following paragraphs describe the response of the FPCP to an FSAVE instruction for the various phases of instruction execution.

6.4.3.1 RESET PHASE. In this phase, no FPCP instructions have been executed since the last hardware reset or FRESTORE of a null state frame. When the FPCP is in this state and an FSAVE is executed, a null format word is returned immediately.

6.4.3.2 IDLE PHASE. In this phase, the FPCP is not executing an instruction, but at least one instruction has been executed since the last hardware reset or FRESTORE of a null state frame. When the MC68881 is in this state and an FSAVE is executed, an idle format word is returned immediately, and an idle state frame is stored.

6.4.3.3 INITIAL PHASE. In this phase, the FPCP is acquiring instruction and operand words from the MPU. In a virtual memory system, a memory fault can occur during this phase due to an attempt to access an operand that is not resident in main memory. In this case, the MPU traps to a fault handler to initiate a transfer from secondary storage, typically involving one or more disk accesses. After initiating the transfer, the operating system usually switches the main processor and coprocessor(s) to another program, thus necessitating a save of the coprocessor state and restoration of the state of the coprocessor relative to the next program. To facilitate saving and restoring the coprocessor, the FPCP responds immediately to a save command during the initial phase by storing a busy state frame.

6.4.3.4 MIDDLE PHASE. The middle phase occurs only in FPCP instructions that take significant processing time (i.e., remainder, transcendental functions, and BCD conversions). During this phase, the internal microcode sequence of the FPCP provides for periodic checkpoints to determine if the MPU has issued a save command. If the MPU initiates a save command to the FPCP between check points, the FPCP sets an internal flag to denote the receipt of the command and returns a come-again format word to the MPU. The MPU repeatedly reads the save CIR until it receives a valid format word. The FPCP continues internal processing up to the next checkpoint, at which time processing stops, and the next read of the save CIR acquires the appropriate format word to start the save operation. At this point, the save command proceeds to completion, and the FPCP supplies a busy state frame.

6.4.3.5 END PHASE. This phase begins when the FPCP is almost finished with a long instruction. The length of the end phase is approximately equal to the amount of time required to perform a save of a busy state frame. When the FPCP reaches the end phase, it takes less time to complete execution of the instruction and save an idle frame than to immediately save a busy state. During this phase, the FPCP uses the come-again format word to force the MPU to wait for the completion of the instruction, and then saves an idle state frame.

Note that most of the FPCP instructions proceed directly from the initial phase to the end phase, and thus, most state frames generated by the FPCP are idle frames.

6.4.4 FRESTORE Protocol

When the MPU decodes an FRESTORE instruction, it evaluates the effective address to locate the format word for the state frame, and writes that format word to the restore CIR of the FPCP. In response to this write cycle, the FPCP aborts any operation that may be in progress and prepares to load a new internal state. The format word that is written to the restore CIR is checked for validity (it must be a null, idle, or busy format word with a version number that matches that of the specified device) before the restore operation begins. After the MPU writes the format word to the FPCP, it then reads the restore CIR to verify

that the format word is valid. If the format word is valid, the FPCP returns the same format word that was written; if the format word is not valid, an illegal format word ($02xx) is returned. If the format word is successfully verified, the MPU begins to transfer the state frame, four bytes at a time, from memory to the FPCP operand CIR.

When transferring the state frame from memory to the FPCP, the MPU first transfers the format word and, after it is verified, transfers the remainder of the state frame. The order of transfer is the same for both the postincrement and the control addressing modes. The long word at the lowest address is transferred first, followed by the long words at successively higher addresses. For the postincrement addressing mode, the specified address register is not updated by the MPU until the entire frame has been successfully transferred. Thus, a fault during an FRESTORE instruction generates a stack frame that does not overwrite any part of the FPCP state frame.

After the entire state frame has been transferred to the FPCP, the MPU continues with the execution of the next instruction (it does not read the response CIR). If an exception related to the FPCP caused the suspension of the task earlier, an RTE instruction is eventually executed to return to the original context. Depending on the exception type, the RTE may re-establish the MCU/coprocessor protocol of the suspended operation or begin the execution of a new FPCP instruction.

6.4.5 Context Switching Summary

To perform a complete context save or restore operation, three FPCP instructions are required. First, the FSAVE and FRESTORE instructions are used to transfer the nonuser-visible portion of the machine state between the FPCP and memory. Second, the FMOVEM instruction may be used to transfer the user-visible portion of the machine, including the floating-point data and control registers.

An important aspect of the FMOVEM instruction is that it cannot cause an exception or report a pending exception; thus the context of the FPCP, including pending exceptions, can be saved and restored in a manner that is completely transparent to a user program. Note that if an FSAVE instruction stores a null state frame, the floating-point data and control registers are reset to their default states, and the FMOVEM instructions are not needed. Figure 6-7 illustrates the manner in which a full context switch might be performed.

INSTRUCTION SEQUENCE TO STORE THE PREVIOUS CONTEXT

```
        FSAVE       -(An)                SAVE MC68881 STATE FRAME
        TST.B       (An)                 CHECK FOR A NULL FRAME
        BEQ         NULL_SV              SKIP PROGRAMMER'S MODEL; SAVE IF NULL
        FMOVEM      FP0-FP7,-(An)        ELSE, SAVE DATA REGISTERS
        FMOVEM      FPCR/FPSR/FPIAR,-(An) AND SAVE CONTROL REGISTERS
        MOVE.L      #-1,-(An)            PLACE NOT-NULL FLAG ON STACK
NULL_SV.
```

INSTRUCTION SEQUENCE TO LOAD THE NEXT CONTEXT

```
        TST.B       (An)                 CHECK FOR NULL FRAME OR NOT-NULL FLAG
        BEQ         NULL_RST             SKIP PROGRAMMER'S MODEL; RESTORE IF NULL
        ADDQ.L      #4,An                ELSE, THROW AWAY THE NOT-NULL FLAG
        FMOVEM      (An)+,FPCR/FPSR/FPIAR RESTORE THE CONTROL REGISTERS
        FMOVEM      (An)+,FP0-FP7        RESTORE THE DATA REGISTERS
NULL_RST FRESTORE   (An)+                RESTORE THE FPCP STATE FRAME
```

Figure 6-7. Full Context Save/Restore Instruction Sequences

SECTION 7
COPROCESSOR INTERFACE

This section describes the coprocessor interface with respect to the communication protocol utilized by the MC68881/MC68882 (FPCP) and MC68020/MC68030 (MPU). This communication protocol includes electrical and command-level mechanisms that allow a coprocessor to act as an extension to the main processor.

The connection between the MPU and the FPCP is an extension of the M68000 bus interface, with the FPCP connected as an auxiliary device to the MPU. The FPCP is selected by a chip-select ($\overline{CS}$) signal that is decoded from the MPU function code and address bus lines.

The FPCP contains a set of coprocessor interface registers (CIRs) by which the main processor and coprocessor communicate. These registers are not related to the programming model implemented by the FPCP. Rather, they are used as communication ports that have specific functions associated with each register. When the FPCP is used as a coprocessor to the MPU, the programmer is never required to explicitly access these interface registers, since the coprocessor interface is implemented in the hardware and microcode of the MPU. A main processor other than an MPU explicitly accesses the FPCP CIRs using a software routine that simulates the behavior of the MPU with respect to the coprocessor interface.

For more information on the electrical interconnection between the main processor and the FPCP, refer to **SECTION 11 INTERFACING METHODS**.

7.1 CHIP-SELECT DECODE

The MPU does not require any special bus signals, beyond the normal M68000 Family bus control signals, for connection to the FPCP. The former MC68000 interrupt acknowledge address space (function code 111) is extended in the MPU to become the CPU address space. A portion of this space, identified by the MPU address bus, is dedicated to coprocessor devices. Figure 7-1 illustrates the information presented on the MPU address bus for coprocessor accesses in the CPU address space.

During CPU space cycles, address bits A19–A16 indicate the CPU space function that the main processor is performing. The MPU utilizes four of the possible 16 encodings of A19–A16 as listed in Table 7-1.

The coprocessor identification (Cp-ID), A15–A13, is taken from the coprocessor instruction operation word (refer to **7.4 COPROCESSOR INSTRUCTIONS**). The coprocessor interface

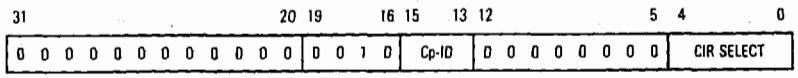


Figure 7-1. MPU Address Bus Encoding for Coprocessor Accesses

Table 7-1. MPU CPU Space Type Field Encoding

CPU Space Type Field (A19–A16)	CPU Space Transaction
0000	Breakpoint Acknowledge
0001	Access Level Control
0010	Coprocessor Communications
1111	Interrupt Acknowledge

register (CIR select) field, A4–A0, is decoded by the FPCP to select the appropriate CIR. For a map of the FPCP coprocessor interface registers in the CPU address space, refer to Figure 7-2. Since address bits A31–A20 are not present on all implementations of M68000 processors, these bits are not essential for decoding CPU space transactions and therefore are don't care bits.

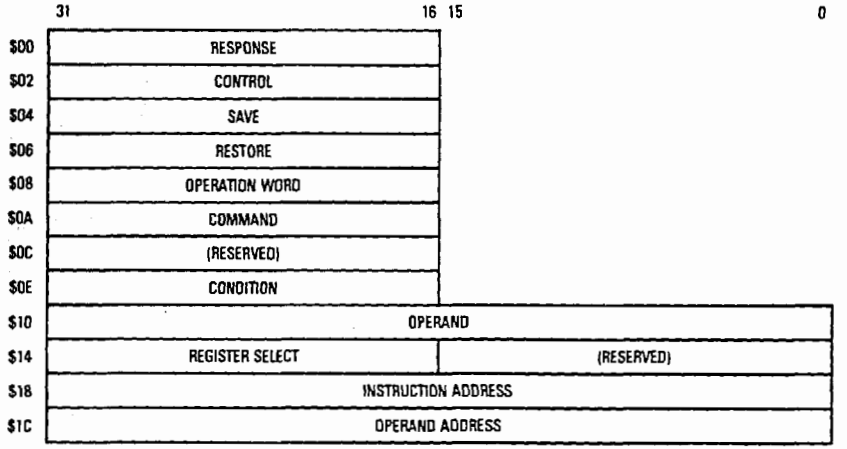


Figure 7-2. FPCP Coprocessor Interface Register Map

The FPCP chip-select decode, therefore, uses the MPU function codes (FC2–FC0), the CPU space type field (A19–A16), and the Cp-ID field (A15–A13). The FPCP decodes the address bits A4–A0 to determine the function (as defined by the selected CIR) of any coprocessor access.

7.2 COPROCESSOR INTERFACE REGISTERS

Table 7-2 identifies the FPCP coprocessor interface register (CIR) locations in the CPU space that are used for communications between the MPU and the FPCP. Figure 7-2 illustrates the memory map of the CIRs on a 32-bit bus. When $\overline{CS}$ is asserted, the FPCP decodes the CIR select field of the address bus (A4–A0) to select the appropriate coprocessor interface register.

When the FPCP is used on a 32-bit bus, the coprocessor interface registers appear at the logical addresses shown in Figure 7-2 and Table 7-2. The M68000 dynamic bus sizing

protocol is used to place all word registers on the upper word of the data bus (D31–D16). This is accomplished by asserting $\overline{\text{DSACK1}}$ and leaving $\overline{\text{DSACK0}}$ negated when any word register (other than the register select CIR) is accessed, regardless of the value of A1.

The following paragraphs describe the characteristics of each of the coprocessor interface registers as implemented by the FPCP. In each description, the read/write attribute of each register is included. If a register is read only, write accesses to that location are ignored; while read accesses of a write-only register always return all ones. In all cases, the FPCP asserts $\overline{\text{DSACKx}}$ in response to the assertion of $\overline{\text{CS}}$ in order to terminate the bus cycle.

Table 7-2. Coprocessor Interface Register Characteristics

Register	A4-A0	Offset	Width	Type
Response	0000x	$00	16	Read
Control	0001x	$02	16	Write
Save	0010x	$04	16	Read
Restore	0011x	$06	16	R/W
Operation Word*	0100x	$08	16	R/W
Command	0101x	$0A	16	Write
(Reserved)	0110x	$0C	16	—
Condition	0111x	$0E	16	Write
Operand	100xx	$10	32	R/W
Register Select	1010x	$14	16	Read
(Reserved)	1011x	$16	16	—
Instruction Address	110xx	$18	32	Write
Operand Address*	111xx	$1C	32	R/W

*These CIRs are optionally implemented by a coprocessor only if they are needed; since they are not used by the MC68881, they are not implemented. Writes to these locations are ignored, and reads always return all ones.

7.2.1 Response CIR ($00)

This 16-bit read-only register is used to communicate service requests from the FPCP to the main processor. A read of the response CIR is always legal, regardless of the state of an instruction dialog. The formats of the response primitives that are returned through this register are detailed in **7.4.2 Response Primitives**.

The execution of an instruction by the FPCP does not start until the main processor reads the response CIR for the first time after a write to the command CIR. Furthermore, a read of a primitive from the response CIR usually causes the FPCP to proceed to the next state in an instruction dialog. For example, if an evaluate effective address and transfer data primitive is encoded in the response CIR and the main processor reads that primitive, it is assumed that the primitive was successfully transferred (and saved for later use, if necessary) and that the requested service is performed. In this case, the FPCP then changes the encoding of the response CIR to the null primitive and waits for an access of the operand CIR to transfer the operand.

7.2.2 Control CIR ($02)

This 16-bit write-only register is utilized by a main processor to issue an exception acknowledge or instruction abort to the FPCP. Figure 7-3 illustrates the format of this register. Only two of the 16 bits are defined: the exception acknowledge (XA) and abort (AB) bits.

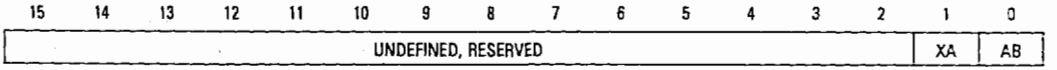


Figure 7-3. Control CIR Register

The implementation of the MC68881 does not utilize these two bits; instead, it interprets a write to this CIR address as an abort command, regardless of the data pattern written. Thus, an exception acknowledge (in response to a take exception primitive) or abort (in response to an illegal format word or an invalid primitive request) issued during any MC68881 instruction protocol, or an explicit write (e.g., with the MOVES instruction) to the control CIR always has the same effect on the MC68881. Also, write cycles to this register are never illegal since the MC68881 always responds in the same manner.

The response of the MC68881 to a write of the control CIR is:

1. To immediately terminate processing for any instruction that may be in progress. If an arithmetic instruction is in progress when an abort is issued, the content of the destination floating-point data register is undefined. No other user-visible registers are disturbed.
2. To clear any pending exceptions.
3. To reset the bus interface unit to the idle condition. Thus, the MC68881 is ready to begin a new instruction protocol following the write cycle.

Unlike the MC68881, the MC68882 distinguishes a write to the AB bit from a write to the XA bit. A write to the AB bit is interpreted as an abort of the last instruction received. However, an abort is only recognized during a certain window, which begins when the main processor writes an instruction to the command CIR and extends to the last CIR read or write required for that instruction. If the write to the AB bit of the control CIR occurs during this abort window, an abort function is initiated. Otherwise, a write to the AB bit is undefined and produces an undefined result. The response of the MC68882 to a write to the AB bit is:

1. To immediately terminate the instruction to which the abort window applies. Any concurrent instruction in progress within the MC68882 is allowed to complete.
2. To reset the bus interface unit to the idle condition, leaving the MC68882 ready to begin a new instruction protocol following the write cycle.

The write to the XA bit signals the MC68882 that the main processor is responding to an MC68882-detected exception. This write operation is necessary to clear any pending exceptions. However, the write operation alone does not guarantee that the exception is cleared. For the floating-point exception traps, it is the responsibility of the exception handler to clear the exception. If the handler does not clear the exception, the MC68882 continues reporting the same exception every time the main processor reads the response CIR. Refer to **5.2.2 Exception Handler Code** for additional exception handler requirements.

7.2.3 Save CIR ($04)

This 16-bit read-only register is used by the main processor to issue a context save command to the FPCP, and by the FPCP to return the format word of the FPCP state frame to the main processor. A read of this register causes any operation that may be executing (except a state save or restore) to be suspended, and a state save operation is initiated.

Following the read of a not-ready, come-again format word from the save CIR, the next expected access is a read of the save CIR. After the read of an idle or busy format word, the next expected access is to the operand CIR (to transfer the state frame). After the read of a null format word, the FPCP is in the reset state, and the next expected access is to the command or condition CIR.

The only time that a read of this register is illegal is when the FPCP is executing a state frame transfer for an FSAVE or FRESTORE instruction; a read of the save CIR is legal at any other time. If the main processor reads the save CIR at an illegal time, the invalid format word is returned. In response to the invalid format word, the main processor can write an abort to the FPCP to return it to the idle state.

7.2.4 Restore CIR ($06)

This 16-bit read/write register is used by the main processor to issue a context restore command to the FPCP and to validate the format word of a state frame. A write of this register causes the FPCP to immediately stop any operation that may be executing and prepare to load a new internal state context from a memory-resident state frame.

After the main processor writes a format word to the restore CIR, it must read the restore CIR to receive the result of the format word verification. If the written format word is valid, that format word is read back from the restore CIR to indicate the successful verification. If the format word is invalid, the invalid format, take exception value is placed in the restore CIR to indicate the verification failure. After a successful verification is signaled, the next expected access is to the operand CIR (to transfer the state frame). After a verification failure is signaled, the main processor should write an abort to the control CIR in order to return the FPCP to the idle state. (The MPU does this automatically.)

7.2.5 Operation Word CIR ($08)

This 16-bit write-only register is not used by the FPCP. The only time that this CIR location is used by the M68000 Family coprocessor interface is when a coprocessor issues the transfer operation word primitive, in which case the main processor writes the F-line word of the instruction to the operation word CIR. Since the FPCP never issues the transfer operation word primitive, the operation word CIR location should never be written by the main processor. If a write to this location occurs, it is ignored; it does not cause a protocol violation.

7.2.6 Command CIR ($0A)

This 16-bit write-only register is used by the main processor to initiate the dialog for a general type coprocessor instruction. When the FPCP detects a write to this CIR location,

the data value is latched from the data bus. If the MC68881 is executing a previous instruction in the APU or if the CU of the MC68882 is still busy when the command CIR is written, the latched command word is saved for later use, and the response CIR is encoded with the null (CA=1, IA=1) primitive. If the FPCP is in the idle or reset state when a write to the command CIR occurs, it encodes the first primitive for the selected instruction dialog in the response CIR in order to begin the execution of the new instruction.

A write to this CIR location is legal at any time except when the FPCP is in the initial phase of a general instruction or before the read of the conditional evaluation for a previous conditional instruction. If a write to the command CIR occurs when it is not expected, a protocol violation occurs, and the command word that is written is not saved by the FPCP.

7.2.7 Condition CIR ($0E)

This 16-bit write-only register is used by the main processor to initiate the dialog for a conditional type coprocessor instruction. When the FPCP detects a write to this CIR location, the data value is latched from the data bus. If the FPCP is executing a previous instruction when the condition CIR is written, the latched conditional predicate is saved for later use, and the response CIR is encoded with the null (CA=1, IA=1) primitive. If the FPCP is in the idle or reset state when a write to the condition CIR occurs, it evaluates the selected condition and returns the null (CA=0, TF=x) primitive (where the TF bit indicates whether the conditional evaluation is true (1) or false (0)).

A write to this CIR location is legal at any time except when the FPCP is in the initial phase of a general instruction, or before the read of the conditional evaluation for a previous conditional instruction. If a write to the condition CIR occurs when it is not expected, a protocol violation occurs, and the conditional predicate that is written is not saved by the FPCP.

7.2.8 Operand CIR ($10)

This 32-bit read/write register is used by the main processor to transfer data to and from the FPCP. The FPCP transfers data through this CIR location in the following cases:

1. Following an evaluate effective address and transfer data primitive
2. Following the read of the register select CIR after a transfer multiple coprocessor registers primitive
3. Following a transfer single main processor register primitive
4. Following a read of an idle or busy format word from the save CIR
5. Following a write of an idle or busy format word to the restore CIR

These five cases are the only times when an access to the operand CIR is legal. At any other time, an access to this CIR location causes a protocol violation.

The FPCP expects all operands that are to be transferred through this CIR location to be aligned with the most significant byte of the register. Any operand larger than four bytes is transferred through this register using a sequence of long-word transfers. If the operand is not a multiple of four bytes in size, the portion remaining after the initial long-word

transfers is aligned with the most significant byte of the operand CIR. Figure 7-4 illustrates the operand CIR data alignment expected by the FPCP when transferring data through the operand CIR.

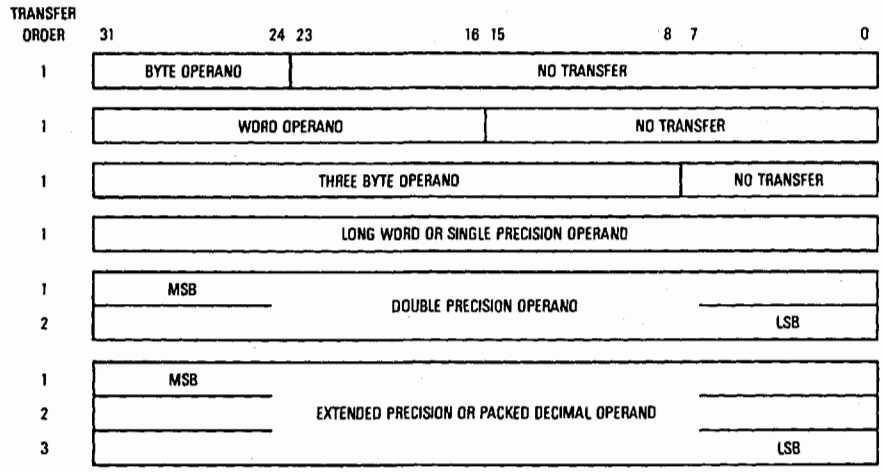


Figure 7-4. Operand CIR Data Alignment

7.2.9 Register Select CIR ($14)

This 16-bit read-only register is read by the main processor to transfer the register mask from the FPCP during a move multiple floating-point data registers operation. The only time that an access to this register is legal is immediately following the issue of a transfer multiple coprocessor registers primitive to the main processor; at any other time, an access of this CIR location causes a protocol violation.

Although a 16-bit register, the FPCP only utilizes the most significant eight bits; the least significant eight bits are always read as zeros. The most significant eight bits contain the register mask for the multiple register transfer, with each bit set if the corresponding floating-point register is to be transferred. The main processor should not interpret the order of the bits in the register mask, but rather count the number of ones in the mask to determine the number of registers to transfer. Each FPCP floating-point data register is 12 bytes long, and thus requires three long-word transfers.

7.2.10 Instruction Address CIR ($18)

This 32-bit write-only register is used by the main processor to transfer the address of the FPCP instruction being executed when the PC bit of any primitive is set. The FPCP only sets the PC bit in the first primitive returned during the dialog for an instruction that can cause an exception, or a take pre-instruction exception primitive for a BSUN exception. When the coprocessor is an MC68881, the main processor may optionally transfer the program counter value to the instruction address CIR at that time or ignore the request. (This choice is left to the discretion of the system designer in order to support exception

handlers in the most efficient manner.) When the coprocessor is an MC68882, the main processor must transfer the program counter value. The MC68882 issues a protocol violation when the main processor fails to transfer the program counter value. The MPU always transfers the PC when needed.

For the MC68881, accesses to the instruction address CIR are neither expected nor unexpected at any point in an instruction dialog; thus, an access to this CIR location never causes a protocol violation. A write to the instruction address CIR updates the FPIAR register in the MC68881 programming model; a read always returns all ones.

Internally, the MC68882 has three instruction address registers. One register is associated with each of the stages of the MC68882 pipeline (BIU, CU, and APU). The instruction address register associated with the arithmetic processing unit (APU) is the floating-point instruction address register (FPIAR). Since only the APU can report an exception, the FPIAR always points to the instruction that causes the exception whenever an exception occurs. When the instruction address CIR is written (whenever the program counter value is passed), the program counter value is also written to the instruction address register associated with the bus interface unit (BIU) stage of the pipeline. The MC68882 interprets this program counter value as the address of the instruction currently in the BIU. The instruction and its address are moved up the pipeline until the instruction reaches the APU stage of the pipeline. If that instruction causes an exception, its address is in the FPIAR (since the FPIAR is the instruction address register for the APU). This implementation is necessary to ensure that an exception handler can point to the correct instruction, the one that causes the exception. However, this implementation requires that the instruction address CIR be written whenever the MC68882 requests it. The MC68882 issues a protocol violation whenever the main processor fails to supply the requested program counter value. A read of the instruction address CIR always returns all ones.

7.2.11 Operand Address CIR ($1C)

This 32-bit read/write register is used by the main processor to transfer an operand address in response to the evaluate and transfer effective address or take address and transfer data primitives. Since the FPCP does not utilize either of these primitives, this CIR is not required for operation and is not implemented. An access to this CIR location does not cause a protocol violation; read cycles always return all ones, and the data is ignored during write accesses.

7.3 INTERPROCESSOR TRANSFERS

All interprocessor transfers are initiated by the MPU. During the processing of an FPCP instruction, the MPU transfers instruction information and data to the FPCP using standard M68000 write bus cycles. The MPU also receives data, requests for service, and status information from the FPCP using standard M68000 read bus cycles. A detailed description of the electrical characteristics of the FPCP bus interface is contained in **SECTION 10 BUS OPERATION** and **SECTION 12 ELECTRICAL SPECIFICATIONS**.

7.4 COPROCESSOR INSTRUCTIONS

FPCP instructions are from one to eight words in length. The first word of the instruction is called the operation word, and the second word of the instruction is called the coprocessor command word. Additional words specify the operands and are either extensions

to the effective addressing mode specified in the operation word or immediate operands that are part of the instruction. The general format of an FPCP instruction is illustrated in Figure 7-5.

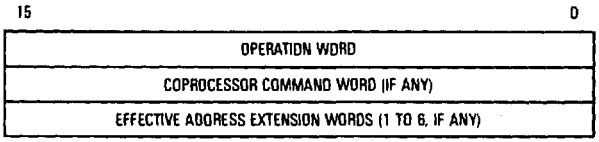


Figure 7-5. Coprocessor Instruction General Format

All coprocessor operations are based on the F-line operation codes (i.e., operation words with bits [15:12] = $F) which instruct the MPU to call upon a coprocessor for execution of the instruction. Figure 7-6 illustrates the format of this word.

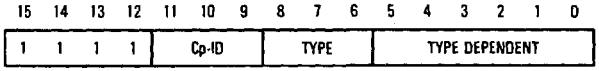


Figure 7-6. FPCP Instruction Operation Word

The Cp-ID field indicates which coprocessor is to be selected. Cp-IDs of 000-101 are reserved by Freescale, and Cp-IDs 110 and 111 are reserved for user definition. The Freescale MPU and FPCP assembler supplies 001 as the Cp-ID for FPCP instructions by default. The type field indicates to the MPU which type of coprocessor operation is selected: general, branch, conditional, save, or restore. The type and type-dependent fields and the coprocessor command word for all FPCP instructions are described in **4.7 INSTRUCTION ENCODING DETAILS.**

7.4.1 Instruction Protocol

All FPCP instructions have a typical protocol which the MPU and FPCP use. This communication protocol is as follows:

1. When the MPU detects an F-line operation word, communication is initiated by writing information (a command, condition selector, or restore format word) to the appropriate FPCP coprocessor interface register location. (The FPCP save instruction is initiated by a read operation.)

2. The MPU then reads the coprocessor response to the previous write operation. The response may indicate any of the following:
   a. The FPCP is busy. MPU checks for interrupts, processes them if any are pending, and then queries the coprocessor again. This allows synchronizing the main processor and coprocessor.
   b. An exception condition exists, and the FPCP instructs the MPU to take an exception using a specific exception vector. The MPU acknowledges the exception and initiates exception processing.

c. There is an FPCP service request (for example, to evaluate the effective address and transfer data between the effective address and the FPCP). The FPCP may also request that the MPU query the coprocessor after the service is performed.
d. The MPU is not needed for further processing of the coprocessor instruction. Communication is terminated, and the MPU is free to begin execution of the next instruction. If the MPU is in the trace mode, the MPU does not take the trace exception until the FPCP completes the processing of the coprocessor instruction.

Each FPCP instruction type has specific requirements based upon this simplified protocol. The main processor requests required for each FPCP instruction are described in **4.7 INSTRUCTION ENCODING DETAILS.** All FPCP main processor service requests (response primitives) are described in the following paragraphs. In addition, the dialog used by the MPU and the FPCP during the execution of each instruction is detailed in **7.5 INSTRUCTION DIALOGS.**

7.4.2 Response Primitives

Data read from the FPCP coprocessor interface response register is referred to as a primitive. Although the M68000 Family coprocessor interface defines 18 response primitives, the FPCP only uses six of those primitives. For additional information on the complete set of response primitives and how they are serviced; refer to the appropriate processor user's manual. The following paragraphs summarize all FPCP response primitives and how they are used.

The M68000 coprocessor response primitives are encoded in a 16-bit word that is transferred to the main processor through the response CIR. Figure 7-7 illustrates the general format of a response primitive.

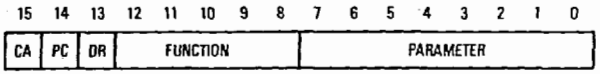


Figure 7-7. M68000 Coprocessor Response Primitive General Format

The encoding of bits [12-0] of a coprocessor response primitive is dependent on the individual primitive being implemented. Bits [15-13], however, are used to specify particular attributes of the response primitive that can be utilized in most of the primitives defined for the M68000 coprocessor interface.

Bit [15] in the primitive format, denoted by CA, is used to specify the come-again operation of the main processor. Whenever the main processor receives a response primitive from the FPCP with the CA bit set to one, it should perform the service indicated by the primitive and then return to read the response CIR again.

Bit [14] in the primitive format, denoted by PC, is used to specify the pass-program-counter operation. If the main processor reads a primitive from the FPCP that has the PC bit set, the main processor should immediately pass the current value of the program counter to the instruction address CIR as the first operation when servicing the primitive request. The value of the program counter passed from the main processor is usually the address of the operation word of the coprocessor instruction executing when the primitive is received.

(This is always the case if the main processor is the MPU.) The FPCP always sets the PC bit in the first primitive of a general type instruction that might cause an exception (i.e., all of the arithmetic and move single floating-point data register instructions when exceptions are enabled), or the take pre-instruction exception primitive for a BSUN trap during a conditional instruction. By updating the FPIAR in this manner, the FPCP can release the main processor for concurrent execution after all operands are fetched, and exception handlers can later locate an instruction that causes an exception. It should be noted that the PC bit is set in only one primitive response during any instruction dialog, and that the MC68881 does not issue a protocol violation if the main processor ignores the request to transfer the PC. The MC68882 issues a protocol violation if the main processor fails to transfer the PC in response to a request.

Bit [13] in the primitive format, denoted by DR, is the direction bit; it controls the direction of operand transfers between the main processor and the FPCP. If DR is zero, the direction of the transfer is from the main processor to the FPCP (a main processor write). If DR is one, the direction of the transfer is from the FPCP to the main processor (a main processor read). If the operation indicated by a given response primitive does not involve an explicit operand transfer, the value of this bit is dependent on the particular primitive encoding.

NOTE

All primitives issued by the MC68881, with the exception of the null primitive, have the CA bit equal to one, causing the MPU to check the response CIR after any service is performed. This allows the MC68881 to assure correct internal operation and to report exceptions immediately after a service is performed. However, the MC68882 may occasionally issue an evaluate <ea> and transfer data primitive with CA equal to zero. This is done in cases where internal operations are not adversely affected by the omission of the read of the response CIR after the operand transfer.

The following paragraphs describe the response primitive encodings used by the FPCP and the expected main processor response to each one in detail.

7.4.2.1 NULL PRIMITIVE. This primitive is used by the FPCP to synchronize operation with the main processor and to allow concurrent execution by the main processor. The format of the null primitive is shown in Figure 7-8. In addition to the variable bits CA and PC previously discussed, the null primitive uses three other variable bits to identify the required action to be taken by the main processor. Bit [8], denoted by IA, is used to specify that the main processor may process pending interrupts if necessary. Bit [1], denoted by PF, is used to indicate the status of the FPCP during concurrent instruction execution. If the PF bit is zero, the FPCP is executing an instruction; otherwise it is idle. Bit [0], denoted by TF, is used to communicate the true or false result of a conditional evaluation. If TF equals one, the condition is true; otherwise it is false.

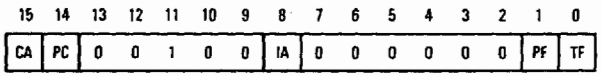


Figure 7-8. Null Primitive Format

As indicated by the format of this primitive, there are 32 possible null primitive encodings of which the FPCP uses only seven. Table 7-3 lists the FPCP null primitive encodings and the circumstances in which they are used.

Table 7-3. Null Primitive Encodings

CA	PC	IA	PF	TF	Usage
0	0	0	0	x	Returned by the FPCP in response to the write of a conditional predicate to the condition CIR. The TF bit indicates the result of the conditional evaluation; TF = 1 if the condition is true; TF = 0 if the condition is false.
0	0	0	1	0	Returned when the FPCP is in the idle state. The PF bit indicates that no instruction is being executed; thus, there is no expected response to this primitive.
0	0	1	0	0	Returned when the FPCP enters the middle or end phase of an instruction to allow concurrent execution by the main processor.The CA bit indicates that no further service is required of the main procesor, but the PF bit indicates that the FPCP has not completed execution of the instruction. The IA bit indicates that if the main processor is in the trace mode, it may process interrupts while waiting for the FPCP to complete execution of the instruction. Since this primitive does not request any specific service, there is no expected response from the main processor.
0	1	1	0	0	The same as the preceding response, except that the main processor is requested to pass the current program counter before proceeding with the next instruction. This response is returned only as the first response of a dialog.
1	0	1	0	0	Returned when the FPCP is executing an instruction and requires further service from the main processor before the next instruction can be executed. This response is also used when a new FPCP instruction is initiated while a previous one is still being executed. The expected response is for the main processor to re-read the response CIR (after servicing pending interrupts).
1	1	1	0	0	The same as the preceding response, except that the main processor is requested to pass the current program counter before processing any pending interrupts and re-reading the response CIR. This response is returned only as the first response of a dialog.

The meanings of the CA and PC bits are as previously described. If IA equals one, the main processor can process pending interrupts as part of the service for the null primitive; otherwise, interrupts should be ignored. The IA bit is set to a one by the FPCP for most null responses thus allowing the main processor to process pending interrupts anytime that it is "waiting" on the FPCP.

The PF bit indicates the processing state of the FPCP during concurrent instruction execution. In normal operation, the PF bit is of no concern to the main processor. However, if the main processor is in the trace mode, it should wait until the FPCP has completed execution of an instruction before taking the trace exception. By monitoring the PF bit in the null response primitive, the main processor can synchronize with the FPCP in this case. If PF equals zero, the FPCP is executing an instruction; otherwise, it is idle.

The TF bit applies only to the conditional instructions. When the main processor writes a conditional predicate to the condition CIR, the FPCP uses the null primitive to return the true or false result of the conditional evaluation. If TF equals one, the condition is true; otherwise, it is false. For all reads of the response CIR for other instruction types, TF is a don't care bit.

7.4.2.2 EVALUATE EFFECTIVE ADDRESS AND TRANSFER DATA PRIMITIVE. This primitive is used by the FPCP to request the transfer of a data item between the floating-point data and control registers and an external location (either memory or a main processor register). The format of this primitive is shown in Figure 7-9. The main processor services this request by evaluating the effective address indicated by the F-line word of the instruction and transferring the number of bytes indicated by the length field of the primitive to or from the operand CIR.

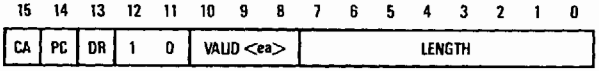


Figure 7-9. Evaluate Effective Address and Transfer Data Primitive Format

Note that the FPCP returns this primitive only once during an instruction dialog. When this primitive is read from the response CIR, it is discarded by the FPCP, and the response encoding is changed to the null primitive. By doing this, the FPCP avoids spurious service request primitives in systems where the MPU is not the main processor.

The meanings of the CA and PC bits are as previously described. The DR bit indicates the direction of data transfer between the effective address location and the operand CIR of the coprocessor. If DR equals zero, the operand is transferred from the effective address location to the coprocessor. If DR equals one, the operand is transferred from the coprocessor to the effective address location.

The effective address that is to be evaluated is specified in the F-line operation word, and any required extension words are fetched by the main processor as needed. If the predecrement or postincrement addressing mode is used, the address register is decremented or incremented before or after the transfer by the size of the operand, as indicated in the length field.

The valid EA field specifies various classes of addressing modes with the encodings shown in Table 7-4. If the effective address in the operation word is not of the specified class, the main processor should write an abort to the control CIR and take an F-line emulator trap. The addressing categories in Table 7-4 are as defined for all M68000 Family processors.

The number of bytes transferred to or from an effective address location is indicated in the length field. If the effective address is a main processor register (register direct), then only lengths of one, two, or four bytes are used. If the effective addressing mode is immediate, the length is always one or even, and the transfer is effective address to coprocessor. If the effective address is a memory location, any length is legal (including odd). If the effective address mode is predecrement or postincrement, with A7 as the specified register and a length of one, the transfer causes the stack pointer to be decremented or incremented by two in order to keep the stack aligned on a word boundary.

Table 7-5 lists the encodings of the evaluate effective address and transfer data primitive that are used by the FPCP and the cases for which they are used.

Table 7-4. Coprocessor Valid Effective Address Codes

000	Control Alterable
001	Data Alterable
010	Memory Alterable
011	Alterable
100	Control
101	Data
110	Memory
111	Any Effective Address

Table 7-5. Evaluate Effective Address and Transfer Data Primitive Encoding

Usage	CA	PC	DR	Valid <ea>	Length
F<op><ea>,FPn (OPCLASS 010)					
Issued as the first primitive of an instruction dialog to request the transfer of an operand from memory or a main processor data register to the FPCP. The length field indicates the size of the operand: byte, word, long or single, double, extended, or packed BCD.	1	*	0	101	1
	1	*	0	101	2
	**	*	0	101	4
	**	*	0	110	8
	**	*	0	110	12
FMOVE FPm,<ea> (OPCLASS 011)					
Issued after the conversion from the internal extended precision format to the destination format is completed to request the transfer of an operand from the FPCP to memory or a main processor data register. The length field indicates the size of the operand: byte, word, long or single, double, extended, or packed BCD.	1	0	1	001	1
	1	0	1	001	2
	**	0	1	001	4
	**	0	1	010	8
	**	0	1	010	12
FMOVE <ea>,FPcr and FMOVEM <ea>,FPcr_list (OPCLASS 100)					
Issued as the first primitive of an instruction dialog to request the transfer of one or more control registers from memory or a main processor register to the FPCP. The length field indicates the total size of all control registers to be moved, 4 bytes per register.	1	0	0	111	4
	1	0	0	101	4
	1	0	0	110	8
	1	0	0	110	12
FMOVE FPcr,<ea> and FMOVEM FPcr_list,<ea> (OPCLASS 101)					
Issued as the first primitive of an instruction dialog to request the transfer of one or more control registers from the FPCP to memory or a main processor register. The length field indicates the total size of all control registers to be moved, 4 bytes per register.	1	0	1	011	4
	1	0	1	001	4
	1	0	1	010	8
	1	0	1	010	12

*PC = 1 if any arithmetic exceptions are enabled; otherwise PC = 0.
**CA = 0 for some MC68882 instructions with S,D,X instruction operand formats; otherwise, CA = 1.

7.4.2.3 TRANSFER SINGLE MAIN PROCESSOR REGISTER PRIMITIVE. This primitive is used by the FPCP to request the transfer of one main processor register. The format of this primitive is shown in Figure 7-10. The main processor services this request by writing a long word to the operand CIR.

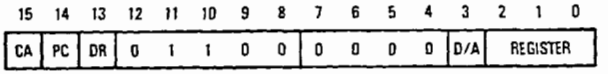


Figure 7-10. Transfer Single Main Processor Register Primitive Format

This primitive is only utilized for the move multiple floating-point data register instruction when the register list is specified as dynamic. Therefore, when this primitive is issued by the FPCP (to fetch the register list), CA is always set; DR, PC, and D/A are always clear (D/A identifies the selected register as a data or address register; zero indicates it is a data register). The least significant three bits identify the main processor data register that contains the register list.

Note that the FPCP returns this primitive only once during an instruction dialog. When this primitive is read from the response CIR, it is discarded by the FPCP, and the response encoding is changed to the null primitive until the request has been serviced. By doing this, the FPCP avoids spurious service requests in systems where the MPU is not the main processor.

7.4.2.4 TRANSFER MULTIPLE COPROCESSOR REGISTERS PRIMITIVE. This primitive is used by the FPCP to request the transfer of a list of floating-point data registers to or from memory. The format of this primitive is shown in Figure 7-11. The main processor services this request by performing an implied effective address evaluation, reading a register list from the register select CIR, and transferring the selected registers (where each register is the size indicated by the length field of the primitive) between the operand CIR and memory. The MPU uses long-word transfers whenever possible while servicing this primitive.

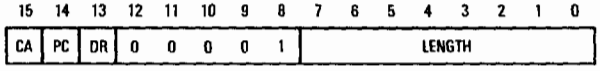


Figure 7-11. Transfer Multiple Coprocessor Registers Primitive Format

Note that the FPCP returns this primitive only once during an instruction dialog. When this primitive is read from the response CIR, it is discarded by the FPCP, and the response encoding is changed to the null primitive until the request has been serviced. By doing this, the FPCP avoids spurious service requests in systems where the MPU is not the main processor.

The meanings of the CA and PC bits are as previously described. For this primitive, CA is always set, and PC is always clear (since the FMOVEM instruction cannot cause an exception). The DR bit indicates the direction of the transfer between the effective address location and the operand CIR. If DR equals zero, the listed registers are transferred from the effective address location to the FPCP. If DR equals one, the listed registers are transferred from the FPCP to the effective address location.

The length field indicates the size, in bytes, of each register to be transferred; for the FPCP, the length is always 12 bytes. The register list that is read from the register select CIR is used by the main processor to determine the number of registers to be transferred (but not the order of the transfer). For each bit that is set in the 16-bit register list, one operand of the size indicated by the length field in transferred. Thus, the total number of bytes transferred in response to this primitive is the product of the length and the number of ones in the register list. Since the FPCP has only eight floating-point data registers, the register list always has zeros in the least significant byte.

The effective address that is evaluated by the main processor is specified in the F-line operation word, and any required extension words are fetched by the main processor, as

needed. If the predecrement or postincrement addressing mode is used, the address register is decremented or incremented (before or after the transfer) by the size of the operand as indicated by the length field. The effective addressing modes that are valid for this primitive are determined by the DR bit, and the mode is validated by the main processor. If DR equals zero, the control and postincrement addressing modes are allowed. If DR equals one, the control alterable and predecrement addressing modes are allowed. If the effective address field of the operation word is not valid for the selected multiple register transfer, the main processor writes an abort to the control CIR (before reading the register select CIR) and takes an F-line trap.

For the control and postincrement addressing modes, the registers are transferred using ascending addresses. For the postincrement addressing mode, the address register is incremented by the length value after each register is transferred. Thus, the final value of the address register is the initial value plus the total number of bytes transferred during the primitive execution.

For the predecrement addressing mode, the operands are written to memory with descending addresses, but the bytes within each operand are written to memory with ascending addresses. For example, Figure 7-12 illustrates the transfer of two floating-point data registers to a stack, using the −(An) addressing mode. The designated stack pointer is decremented by 12 bytes before the transfer of each register. Then the bytes within each register are written to memory with ascending addresses. Thus, the address register is decremented by the total number of bytes transferred by the end of the primitive execution.

7.4.2.5 TAKE PRE-INSTRUCTION EXCEPTION PRIMITIVE. Take exception primitives are used by the FPCP to instruct the main processor to abort the current operation and initiate exception processing. The main processor services these requests by writing an exception acknowledge to the control CIR (which clears the pending exception in the FPCP), by creating the appropriate stack frame on the currently active supervisor stack, and by beginning execution of an exception handler. The exception handler is located by using the vector number that is supplied as part of the take exception primitive. Table 7-6 lists the vector numbers used by the FPCP.

Note that the MC68881 returns one of these primitives only once during the instruction dialog. When an exception acknowledge is written to the control CIR, the take exception primitive is discarded by the MC68881, and the response encoding is changed to the null primitive. By doing this, the MC68881 assures that the take exception request is received

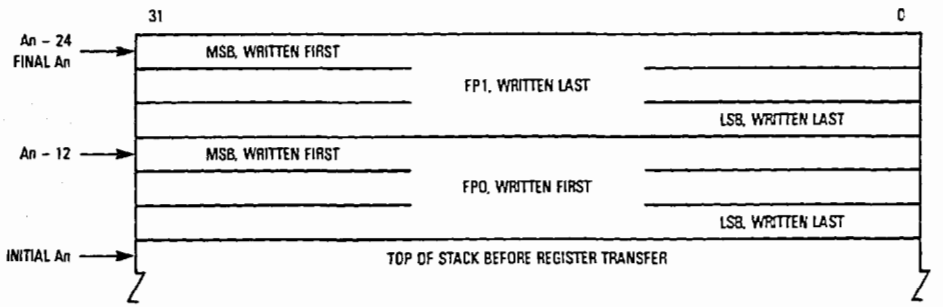


Figure 7-12. Transfer Multiple Floating-Point Data Register to Stack Example

Table 7-6. FPCP Vector Numbers

Vector Decimal	Number Hexadecimal	Vector Offset (Hexidecimal)	Assignment
11	$0B	$02C	F-Line Emulator
13	$0D	$034	Coprocessor Protocol Violation
48	$30	$0C0	Branch or Set on Unordered Condition
49	$31	$0C4	Inexact Result
50	$32	$0C8	Floating-Point Divide by Zero
51	$33	$0CC	Underflow
52	$34	$0D0	Operand Error
53	$35	$0D4	Overflow
54	$36	$0D8	Signaling NAN

by the main processor, but avoids spurious service request primitives in systems where the MPU is not the main processor.

The MC68882, however, does not discard the primitive unless a read of the save CIR is detected. This is to ensure that the FSAVE instruction is executed and the state of the conversion unit (CU) is saved. The state of the CU must be saved because a second instruction, in the CU, may be partially executed.

While the M68000 coprocessor interface defines three take-exception primitives, the FPCP utilizes only two of them. The other take exception primitive is described in the next section.

The take pre-instruction exception primitive is used by the FPCP when an arithmetic (OP-CLASS 000, 010, and 011) or conditional instruction is initiated and an exception is pending from a previously executed, concurrent instruction. This primitive is also returned if an illegal command word is written to the command CIR or if a protocol violation occurs. Finally, this primitive is issued when a conditional instruction is executed that utilizes one of the IEEE nonaware conditional predicates, and the NAN bit in the FPSR condition code byte is set. The format of this primitive is shown in Figure 7-13.

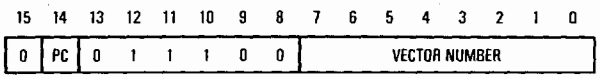


Figure 7-13. Take Pre-Instruction Exception Primitive Format

The CA bit is always zero for this primitive, since there is an implied protocol pre-emption in this service request. The PC bit is zero if the exception is pre-empting the execution of a new FPCP instruction. The PC bit is one if the exception is due to an illegal command word, or if it is reported during the execution of a conditional instruction in lieu of the true/false result of the conditional evaluation. The vector number identifies the type of the exception and is used by the main processor to locate the exception handler routine.

In response to this primitive, the MPU creates a four-word stack frame on top of the currently active supervisor stack. The format of this stack frame is shown in Figure 7-14. The value of the program counter in the stack frame is the address of the F-line operation word of

the FPCP instruction that was preempted by the exception (i.e., the arithmetic or conditional instruction attempted in the case of an exception pending from a previous instruction or an F-line exception, or the conditional instruction in the case of a BSUN exception). Thus, if no modifications are made to the stack frame within the exception handler, an RTE instruction causes the MPU to return and reinitiate the instruction that was being attempted when the primitive was received. Refer to the appropriate user's manual for further details on exception handling by the MPU.

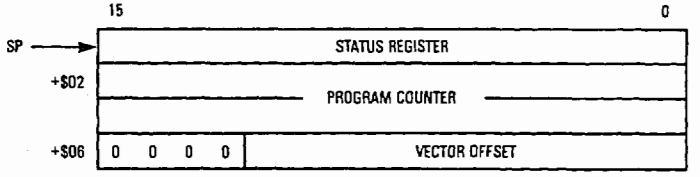


Figure 7-14. Pre-Instruction Exception Stack Frame

7.4.2.6 TAKE MID-INSTRUCTION EXCEPTION PRIMITIVE. This primitive is used by the FPCP when an exception occurs during the execution of an FMOVE FPm,<ea> instruction. In the MC68882, an exception caused by a previous instruction is reported by the current instruction using this primitive. *See* **7.4.2.5 TAKE PRE-INSTRUCTION EXCEPTION PRIMITIVE** for information common to both take exception primitives. The format of this primitive is shown in Figure 7-15.

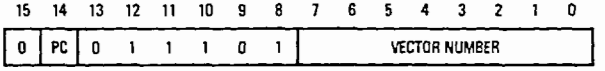


Figure 7-15. Take Mid-Instruction Exception Primitive Format

The CA bit is always zero for this primitive, because there is an implied protocol preemption in this service request. The PC bit is always zero, since a null primitive earlier in the dialog for the move-out instruction is used to request the program counter transfer. The vector number identifies the type of the exception, and is used by the main processor to locate the exception handler routine.

In response to this primitive, the MPU creates a ten-word stack frame on top of the currently active supervisor stack. The format of this stack frame is shown in Figure 7-16. If the exception is due to an FMOVE FPn,<ea> instruction, the ScanPC value is the address of the instruction immediately following the FMOVE instruction. The value of the program counter in the stack frame is the address of the F-line operation word of the FPCP instruction that caused the exception. The operation word image contains the F-line word of the FMOVE instruction. The effective address value is the memory address of the destination operand. Note that the take mid-instruction exception primitive is used in this case solely for the purpose of placing the evaluated effective address in the stack frame, to avoid requiring an exception handler to recalculate it.

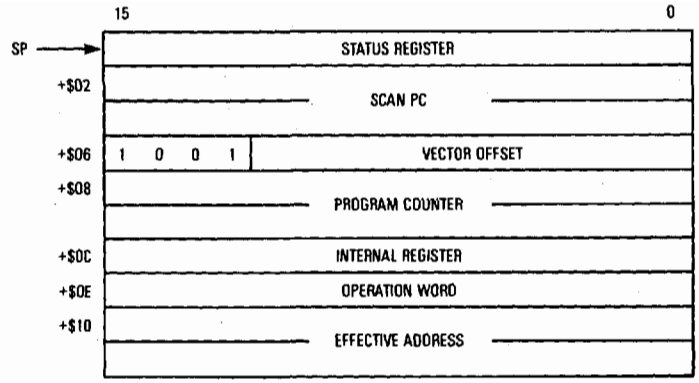


Figure 7-16. Mid-Instruction Stack Frame

If the exception is caused by a previous instruction, the ScanPC value is the address of the instruction immediately following the instruction that reported the exception. The value of the PC in the stack frame is the address of the F-line operation word of the instruction that reported the exception. The operation word image contains the F-line word of the instruction that reported the exception. The effective address only contains the valid effective address when an evaluate <ea> primitive was issued before the exception was reported.

If no modifications are made to the stack frame within the exception handler, an RTE instruction causes the MPU to return to read the response CIR. Thus, the main processor continues the execution of the instruction upon return.

7.4.2.7 RESPONSE PRIMITIVE SUMMARY. Table 7-7 lists in numeric order a summary of all primitive responses utilized by the FPCP.

7.5 INSTRUCTION DIALOGS

The following paragraphs describe in detail the coprocessor communications dialogs that are executed by the FPCP and MPU during each floating-point instruction. In this discussion, a dialog refers to the sequence of command and data transfers to the FPCP, and the service request primitives that are returned to control that sequence. Although the following discussion assumes that the main processor is an MC68020 or MC68030, information is also presented that may be used by designers of systems that utilize a different main processor.

The diagrams presented in the following paragraphs represent the activity of the MPU and the FPCP during the execution of a floating-point instruction. In these diagrams, boxes are used to identify periods of time during which a device is actively participating in the execution of an instruction; the absence of a box during a period indicates that a device is waiting on the other one to complete an operation, or that concurrent execution of unrelated instructions may take place.

Each box in the following diagrams is labeled to indicate the activity depicted by that box. The labels above the boxes identify the actions taken by the main processor, and the labels below the boxes identify the encoding of the response CIR at any time during a dialog.

Table 7-7. MC68881/MC68882 Primitive Responses

Primitive Value	Primitive Type	Comments
$0800 $0801 $0802 $0900	Null	CA=0, PC=0, IA=0, PF=0, TF=0 CA=0, PC=0, IA=0, PF=0, TF=0 CA=0, PC=0, IA=0, PF=1, TF=0 CA=0, PC=0, IA=1, PF=0, TF=0
$1504 $1608 $160C	Evaluate <ea> and Transfer Data CA=0, PC=0, DR=0 (External to MC68882)	Single Double Extended
$1C0B $1C31 $1C32 $1C33 $1C34 $1C35 $1C36	Take Pre-Instruction Exception PC=0	F-Line Emulator Inexact Result Floating-Point Divide by Zero Underflow Operand Error Overflow Signaling NAN
$1D0D $1D31 $1D32 $1D33 $1D34 $1D35 $1D36	Take Mid-Instruction Exception PC=0	Coprocessor Protocol Violation Inexact Result Floating-Point Divide by Zero Underflow Oeprand Error Overflow Signaling NAN
$3104 $3208 $320C	Evaluate <ea> and Transfer Data CA=0, PC=1, DR=1 (MC68882 to External)	Single Double Extended
$4900	Null	CA=0, PC=1, IA=1, PF=0, TF=0
$5504 $5608 $560C	Evaluate <ea> and Transfer Data CA=0, PC=1, DR=0 (External to MC68882)	Single Double Extended
$5C30	Take Pre-Instruction Exception PC=0	Branch or Set On Unordered
$810C	Transfer Multiple Coprocessor Registers CA=1, PC=0, DR=0 (Memory to FPCP)	
$8900	Null	CA=1, PC=0, IA=1, PF=0, TF=0
$8C00 $8C01 $8C02 $8C03 $8C04 $8C05 $8C06 $8C07	Transfer Single Main Processor Register CA=1, PC=0, DR=0 (Main Processor to FPCP)	D0 D1 D2 D3 D4 D5 D6 D7
$9501 $9502 $9504 $9608 $960C $9704	Evaluate <ea> and Transfer Data CA=1, PC=0, DR=0 (External to FPCP)	Byte Word Long, Single, FPCR, or FPSR Double or Two FPCR's (Memory Only) Extended, Packed, or Three FPCR's (Memory Only) FPIAR
$A10C	Transfer Multiple Coprocessor Registers CA=1, PC=0, DR=1 (FPCP to Memory)	
$B101 $B102 $B104 $B208 $B20C $B304	Evaluate <ea> and Transfer Data CA=1, PC=0, DR=1 (FPCP to External)	Byte Word Long, Single, FPCR, or FPSR Double or Two FPCR's (Memory Only) Extended, Packed or Three FPCR's (Memory Only) FPIAR

Table 7-7. MC68881/MC68882 Primitive Responses (Continued)

Primitive Value	Primitive Type	Comments
$C900	Null	CA=1, PC=1, IA=1, PF=0, TF=0
$CC00 $CC01 $CC02 $CC03 $CC04 $CC05 $CC06 $CC07	Transfer Single Main Processor Register CA=1, PC=1, DR=0 (Main Processor to FPCP)	D0 D1 D2 D3 D4 D5 D6 D7
$D501 $D502 $D504 $D608 $D60C	Evaluate <ea> and Transfer Data CA=1, PC=1, DR=0 (External to FPCP)	Byte Word Long or Single Double (Memory Only) Extended or Packed (Memory Only)

When a response CIR encoding is indicated, that encoding is received by the main processor any time that the response CIR is read until the next primitive encoding is indicated.

In all of the succeeding paragraphs, the following assumptions are made:

1. Before the start of an instruction dialog, except for the FSAVE and FRESTORE instructions, the FPCP is in the idle state.
2. The MPU and the FPCP communicate via a 32-bit data bus.
3. The memory width is 32 bits, and all memory operands are long-word aligned.

Also, for periods during which the MPU is required to wait for the FPCP (i.e., during move-to-memory operation, or if the MPU is in the trace mode), only one of the response CIR reads is explicitly indicated. In actual operation, numerous reads of the response CIR may occur in these cases. Similarly, if the FPCP is not idle before the initiation of a new instruction, multiple reads of the null (CA=1, IA=1; $8900) primitive may occur after the command or condition CIR write and before the read of the first primitive shown in a diagram.

7.5.1 General Instructions

This group of instructions includes all of the arithmetic instructions, the move system control register instructions, the move instructions, and the move multiple floating-point register instructions. The factor common to these instructions is the format of the F-line operation word, which uses the cpGEN format of the M68000 Family coprocessor instruction set. Thus, the initial phase of the communications dialog for these instructions is identical, with the MPU writing the command word to the FPCP and then relying on the FPCP to control the remainder of the dialog through the use of the coprocessor interface response primitive set. The following paragraphs discuss the five different protocols that are used by the FPCP for this group of instructions.

For each of the general instruction dialogs, with the exception of the register-to-register dialog, there is an important consideration for systems that use the FPCP with a main processor other than an MPU. This consideration is that the come-again request in any evaluate effective address and transfer data primitive or transfer multiple coprocessor

registers primitive should not be ignored. The FPCP sets the CA bit in these primitives to assure correct operation regardless of the frequency relationship between the FPCP clock and the main processor clock. By requiring the main processor to perform a final read of the response CIR (which is a cycle that is synchronous with the FPCP CLK signal) after the last operand CIR access and before continuing with the next instruction, the FPCP assures that the operand transfer is completed internally before the main processor can initiate the next instruction by writing the command or condition CIRs. This sequence assures that spurious protocol violations (detected by the FPCP) do not occur in systems where the main processor clock frequency is much faster than the MC68881 clock frequency.

During the instruction dialogs for external-to-register (opclass 010 and register-to-external (opclass 011)) instructions, the MC68882 in some cases issues the evaluate effective address and transfer data primitive with CA = 0 instead of CA = 1. In these cases, the main processor need not read the response CIR after the coprocessor has transferred the operands for that instruction. This provides more potential instruction overlap with the next coprocessor instruction. Normally, a second coprocessor instruction causes the main processor to write to the command or condition CIR and then to read from the response CIR. If the read from the response CIR of the second instruction occurs earlier than three clocks after the completion of the last operand transfer of the previous instruction, spurious protocol violations may occur. In a worst-case situation, the main processor uses these three clocks in writing to the command CIR. However, if the main processor has a higher clock frequency than the MC68882, it is possible that the write to the command CIR can take less than three MC68882 clocks. The design of the MC68882 allows the MPU clock frequency to be as much as 1.5 times the MC68882 clock frequency. For main processors other than the MC68020 or MC68030, the system designer must ensure that the main processor does not read the response CIR during the initiation of an instruction less than three MC68882 clocks after the last operand transfer of the previous instruction.

7.5.1.1 REGISTER-TO-REGISTER (OPCLASS 000). This dialog is utilized for all of the arithmetic and move instructions that use floating-point data registers for both the source and destination operands and for the FMOVECR instruction. Since the FPCP contains both operands when such an instruction is initiated, no external data references are required before the calculation can be performed. Thus, after the MPU has written the command word to the FPCP, it is released to execute the next instruction. If any arithmetic exceptions are enabled, the FPCP requests the transfer of the program counter. This request can be ignored by a main processor using an MC68881, but the MC68882 issues a protocol violation if the main processor ignores this request. The program counter write cycle does not affect instruction execution time (since it occurs concurrently with the FPCP instruction execution).

The FPCP dialog for this instruction type is shown in Figure 7-17. Also shown in this figure is the key for all of the dialog figures presented in subsequent paragraphs.

7.5.1.2 EXTERNAL-TO-REGISTER (OPCLASS 010). This dialog is utilized for all of the arithmetic and move instructions that reference memory or a main processor register for the source operand. Since the FPCP does not contain both operands when such an instruction is initiated, external data references are required before the calculation can be performed. The FPCP requests the fetch of the required external operand with the first primitive of the dialog. The second primitive of the dialog is then used to release the main processor to execute the next instruction (once the operand transfer is completed). Note that the read of the first primitive causes the response CIR encoding to be changed to the null primitive,

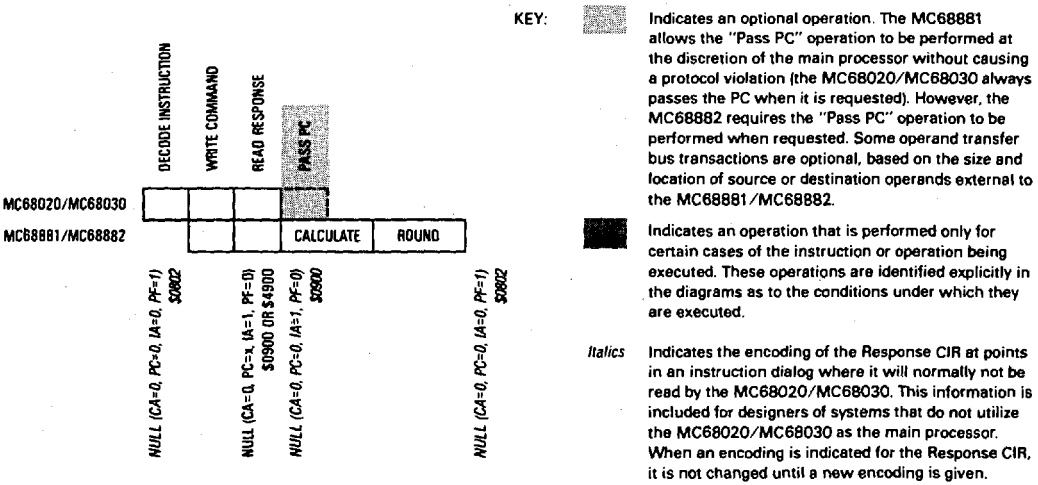


Figure 7-17. MC68881 Register-to-Register Instruction Dialog

thus avoiding spurious request primitives in non-MPU based systems. The MC68881 dialog for this instruction type, which also applies to the MC68882 with operand data formats other than single, double, or extended, is shown in Figure 7-18.

When the operand data format is single, double, or extended, the MC68882 issues the evaluate <ea> and transfer data primitive with CA = 0 instead of CA = 1. Figure 7-19 shows the dialog for the MC68882 external-to-register instructions with single, double, or extended data format operands.

If any arithmetic exceptions are enabled, the FPCP requests the transfer of the program counter with the first primitive. However, the main processor using an MC68881 can ignore this request; the MC68882 issues a protocol violation if the main processor ignores the request.

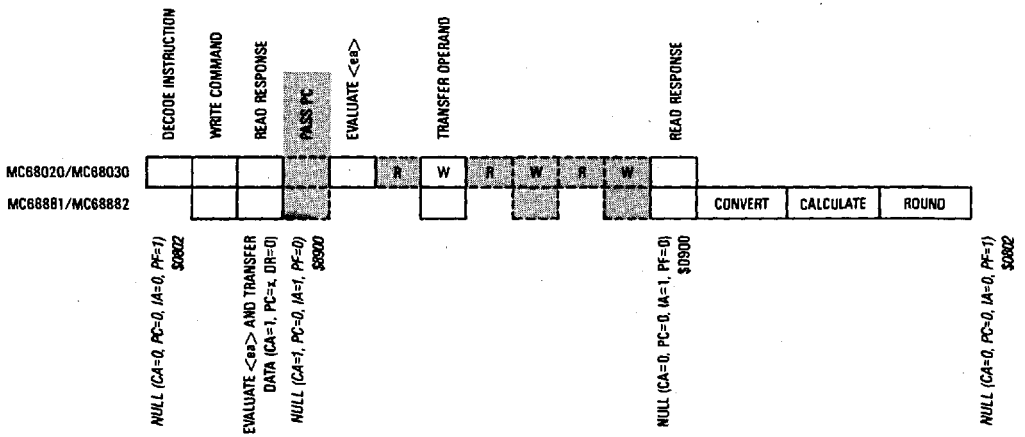


Figure 7-18. MC68881/MC68882 External-to-Register Instruction Dialog

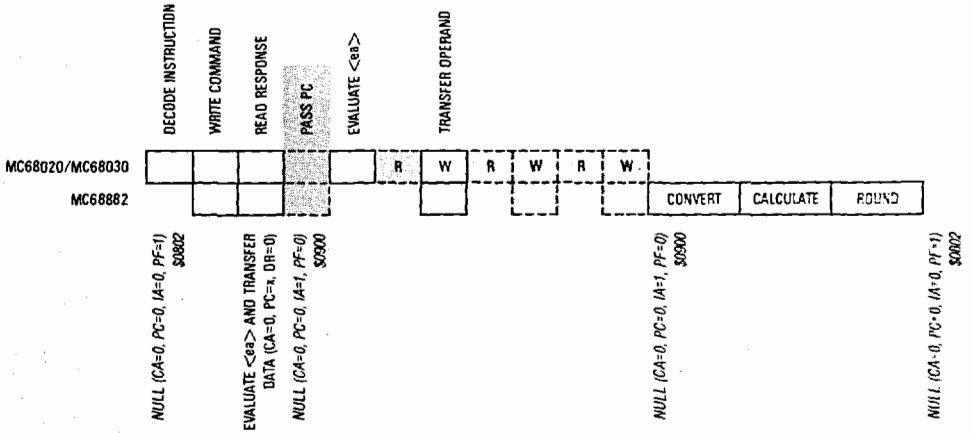


Figure 7-19. MC68882 External-to-Register Instruction Dialog

7

The operation boxes that are marked "R" and "W" indicate an operand read or write cycle, respectively, by the MPU. Those operand transfer boxes that are shaded are optionally executed, depending on the size and location of the source operand. For example, none of the shaded boxes are executed for source operands that reside in the MPU registers. Also, note that the FMOVECR instruction, while it is an opclass 010 instruction, uses the register-to-register protocol described in **7.5.1.1 REGISTER-TO-REGISTER (OPCLASS 000)**.

7.5.1.3 REGISTER-TO-EXTERNAL (OPCLASS 011). This dialog is utilized only for the move from floating-point data register instruction. The MC68881 dialog for this instruction type, which also applies to the MC68882 except when the data format is single, double, or extended, is shown in Figure 7-20. The first primitive returned depends on the destination

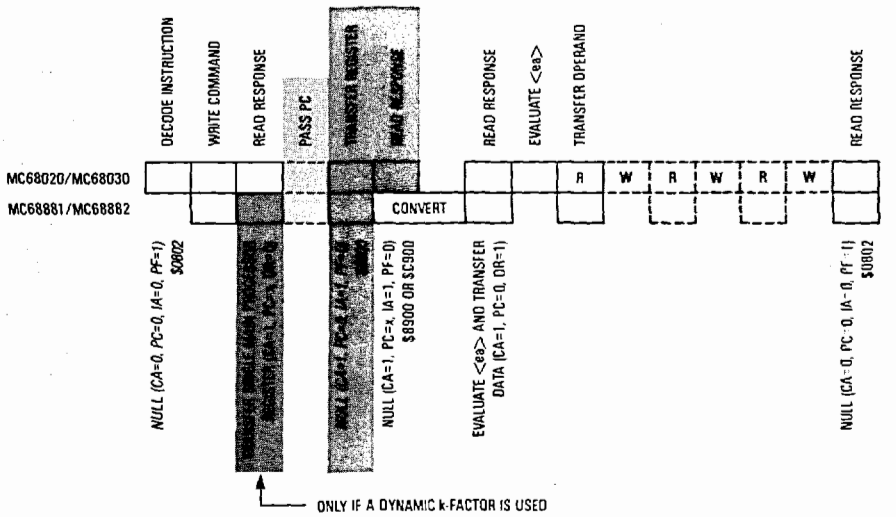


Figure 7-20. MC68881/MC68882 Register-to-External Instruction Dialog

data format, since additional format information is required to generate a packed decimal destination operand when a dynamic k factor is specified. If the destination data type is packed decimal and a dynamic k factor is used, the first response is the transfer single main processor register primitive (to transfer the contents of the register containing the k factor). For all other destination data formats, the first request is the null (CA=1) primitive, since a conversion from the internal data format to the desired destination format must be performed before any further action is required of the main processor. For the dynamic k-factor case, the conversion starts after the main processor register transfer is completed. The conversion starts immediately after the first read of the response CIR for all other destination operand formats. The main processor is allowed to service pending interrupts while it is waiting for the conversion to complete.

If any arithmetic exceptions are enabled, the first primitive requests the transfer of the program counter. However, this request can be ignored when the main processor uses an MC68881; the MC68882 issues a protocol violation when the main processor ignores this request. The program counter write cycle may not affect instruction execution time (since it can occur concurrently with the operand conversion if the destination format is not packed decimal with a dynamic k factor). Only the first primitive requests the transfer of the program counter; thus, if the transfer single main processor register primitive is issued first, the PC bit is not set in any subsequent null primitive. If the first primitive is the null primitive and the program counter transfer is required, the PC bit is set.

The pass program counter operation is requested in one of two of the primitive encodings (shown in Figure 7-20 with the notation "PC=x"). For the packed decimal with a dynamic k-factor case, the dark shaded operations are always performed with the PC bit set if necessary. The MPU services the transfer single main processor primitive with the PC bit set by first transferring the program counter and then transferring the requested register. For all other destination data formats, the dark shaded operations are not performed, and the box labeled "Convert" is "folded under" the first box labeled "Read Response". For these cases, the null primitive may request the transfer of the PC, and that transfer occurs concurrently with the operand conversion by the FPCP.

When the operand conversion is completed, the next read of the response CIR returns the evaluate effective address and transfer data primitive. The main processor then reads the conversion result from the operand CIR and writes it to the appropriate destination location. Note that the read of the evaluate effective address and transfer data primitive causes the response CIR encoding to be changed to the null primitive, thus avoiding spurious request primitives in non-MC68020 or non-MC68030 based systems.

The operation boxes that are marked "R" or "W" indicate an operand read or write cycle, respectively, by the MPU. Those operand transfer boxes that are lightly shaded are optionally executed, depending on the size and location of the source operand. For example, none of the shaded boxes are executed for destination operands that reside in the MPU registers.

The MC68882 dialog differs from the dialog shown in Figure 7-20 when the operand data format is single, double, or extended, except for the following conditions:

- The data in the floating-point register is data type NAN, unnormalized, or denormalized.
- A rounding overflow or underflow has occurred, and the operand data format is single or double.
- The INEX2 bit of the FPSR exception enable byte is set, and the operand data format is single or double.

When any of these conditions occurs, the dialog shown in Figure 7-20 applies. Otherwise, when the operand data format is single, double, or extended, the MC68882 issues the evaluate <ea> and transfer data primitive with CA = 0 instead of CA = 1. Figure 7-21 shows the MC68882 dialog.

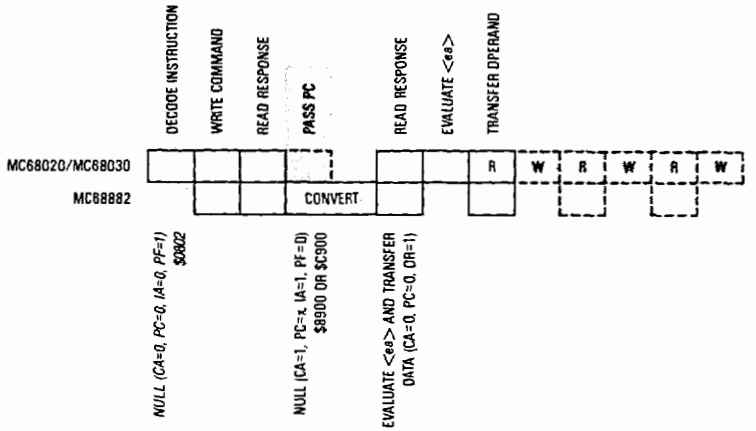


Figure 7-21. MC68882 Register-to-External Instruction Dialog (S, D, and X Formats)

7.5.1.4 MOVE CONTROL REGISTERS (OPCLASS 100 and 101). This dialog is utilized for the move single or multiple floating-point system control registers instructions. The dialog for this instruction type is shown in Figure 7-22. The first primitive of the dialog requests that the main processor evaluate the effective address and transfer the appropriate number of bytes to or from the operand CIR. The read of the first primitive causes the response CIR encoding to be changed to the null primitive, thus avoiding spurious request primitives in non-MC68020 or non-MC68030 based systems. When the transfer data primitive service is complete, the main processor is released to begin execution of the next instruction. Note

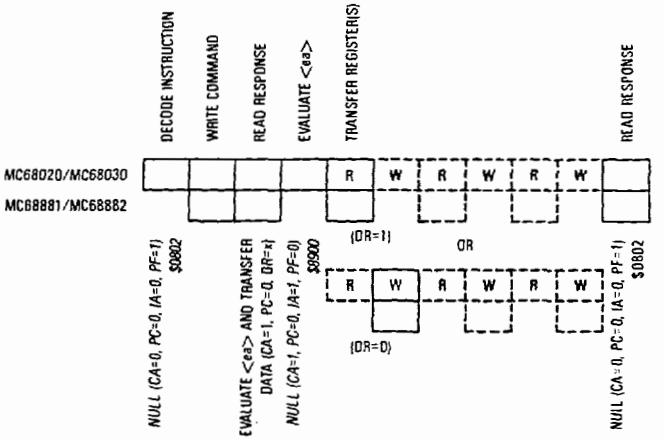


Figure 7-22. Move Control Register Instruction Dialog

that since this instruction type cannot cause an exception, the PC bit is not set in any primitive; thus, these instructions can be used to read or write the control registers without overwriting the FPIAR contents.

The operation boxes that are marked "R" or "W" indicate an operand read or write cycle, respectively, by the MPU. Those operand transfer boxes that are shaded are optionally executed, depending on the size and location of the source operand. For example, none of the shaded boxes are executed for source operands that reside in the MPU registers.

7.5.1.5 MOVE MULTIPLE FPn (OPCLASS 110 and 111). This dialog is utilized for the move multiple floating-point data registers instruction. The dialog for this instruction type is shown in Figure 7-23. The first primitive of the dialog depends on the type of register list specified by the instruction. If the static register list form of the instruction is used, the first service request issued is the transfer multiple coprocessor registers primitive. For the dynamic register list form, the first primitive requests the transfer of the main processor data register that contains the register mask, and then the transfer multiple coprocessor registers primitive is issued. In response to the transfer multiple coprocessor registers primitive, the main processor reads the register list from the register select CIR and transfers one register for each bit that is set in the list. (Note that the register list can be equal to zero; in which case, no register transfer occurs.)

The read of the transfer single main processor register and transfer multiple coprocessor registers primitives causes the response CIR encoding to be changed to the null primitive, thus avoiding spurious request primitives in non-MC68020 or non-MC68030 based systems. If the transfer single main processor register primitive is issued, the transfer multiple coprocessor register primitive is not issued until the first service request is completed. When the transfer multiple coprocessor register primitive service is complete, the main processor is released to begin execution of the next instruction. Note that since this instruction type cannot cause an exception, the PC bit is not set in any primitive; thus, these

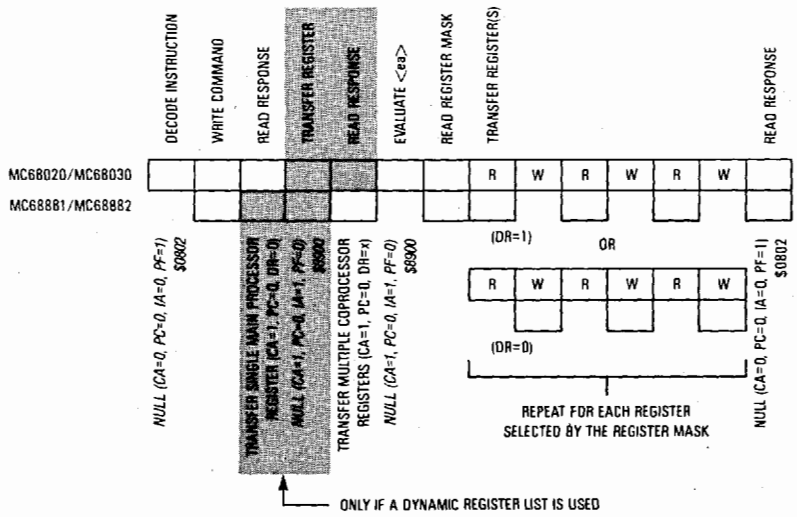


Figure 7-23. Move Multiple Floating-Point Data Registers Instruction Dialog

instructions can be used to read the floating-point data registers without overwriting the FPIAR contents.

The operation boxes that are marked "R" and "W" indicate an operand read or write cycle, respectively, by the MPU.

7.5.2 Conditional Instructions

This group of instructions includes the FBcc, FDBcc, FNOP, FScc, and FTRAPcc instructions. These instructions have two factors in common. First, the execution of each instruction is inherent in the M68000 Family coprocessor instruction set definition, and the dialog used for all of them is the same. Second, in each of these instructions, the coprocessor completes all previous floating-point instructions before it begins to evaluate the result of the received conditional predicate. This guarantees a sequential execution model. The dialog begins when the main processor writes the conditional predicate to the FPCP and then reads the response CIR. If the APU and the CU (in the case of an MC68882) are not idle, a null primitive (CA = 1, IA = 1, PC = 0, TF = 0) is returned, and the main processor reads the response CIR again later. This process of rereading the response CIR continues until the coprocessor is idle. Note that it is possible for a pre-instruction exception to be reported at any time during this process. If no exception is reported, and when the coprocessor is finally idle, the coprocessor evaluates the condition and responds with a null primitive (CA = 0, TF = x, where x is the true or false result). The main processor then proceeds with the appropriate conditional action. The dialog (assuming that the coprocessor is idle when the conditional predicate is written) is shown in Figure 7-24.

7.5.3 Context Switch Instructions

This group of instructions includes the FSAVE and FRESTORE instructions. The factor common to these instructions is that the execution of each one is inherent in the M68000 Family coprocessor instruction set definition, and the coprocessor does not control the

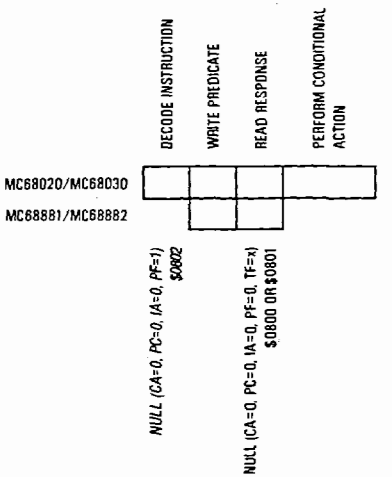


Figure 7-24. Conditional Instruction Dialog

dialog in the flexible manner available with the general and conditional instruction types. The dialog consists of the save or restore command, followed by the transfer of the appropriate state frame. The only control that the FPCP has over this dialog is for the FSAVE instruction; in which case, it may request that the main processor delay the save operation until the FPCP is ready to perform it. These dialogs are discussed in the following paragraphs.

7.5.3.1 FSAVE. This dialog is utilized for the context save instruction. The dialog for this instruction is shown in Figure 7-25. There are no primitive responses during this dialog; instead, the FPCP controls the frame transfer to a limited extent through the use of the format word encoding.

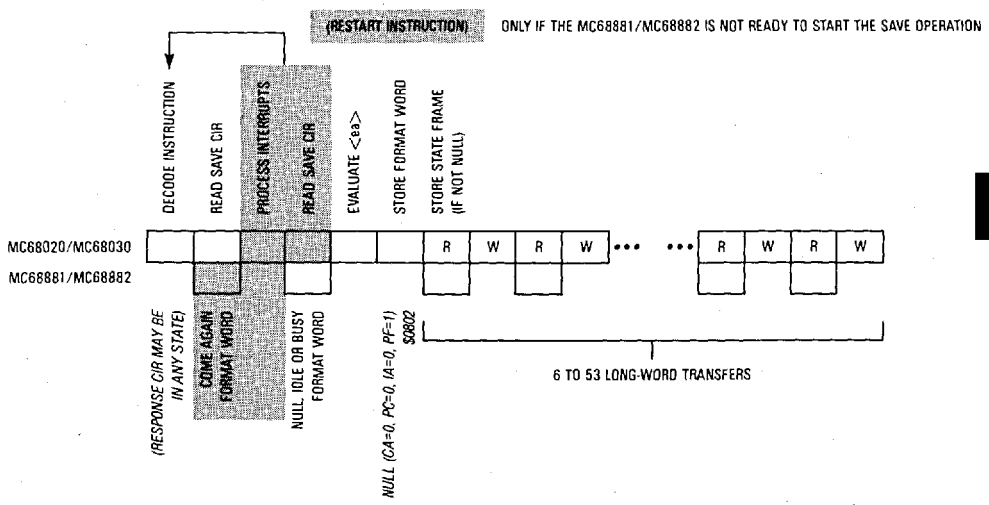


Figure 7-25. FSAVE Instruction Dialog

The main processor initiates this dialog by reading from the save CIR. During this read cycle, the FPCP returns a format word that indicates the current state of the machine. For most cases of this dialog with the MC68881, the first read of the save CIR returns the idle format word, and the main processor then proceeds to transfer six long words from the operand CIR to memory. In this dialog with the MC68882, the idle format word is followed by 14 long words. Optionally, the first primitive may be a null format word, in which case no state frame is transferred. Alternatively, the first primitive may be a busy format word, in which case 45 (53 for the MC68882) long words are transferred. Finally, the save CIR read may return the not-ready, come-again format word. In this case, the main processor may process pending interrupts and restart the instruction, or reread the save CIR until a different format word is received. The invalid format word may also be returned, as discussed in **7.5.4.6 FORMAT EXCEPTION, FSAVE INSTRUCTION.**

After the MPU receives a valid format word, it then evaluates the effective address and writes the format word to that address. The appropriate state frame is then transferred to the effective address, and the main processor is free to proceed with the execution of the

next instruction. Note that by using this sequence, the MPU can take a pre-instruction exception in response to a pending interrupt (if a not-ready, come-again format word is received) and then return to restart the instruction rather than taking a mid-instruction exception.

Note that after the state save operation is complete, the FPCP is in the idle state with no pending exceptions.

7.5.3.2 FRESTORE. This dialog is utilized for the context restore instruction. The dialog for this instruction is shown in Figure 7-26. There are no primitive responses during this dialog; instead, the FPCP controls the frame transfer to a limited extent through the use of the format word encoding.

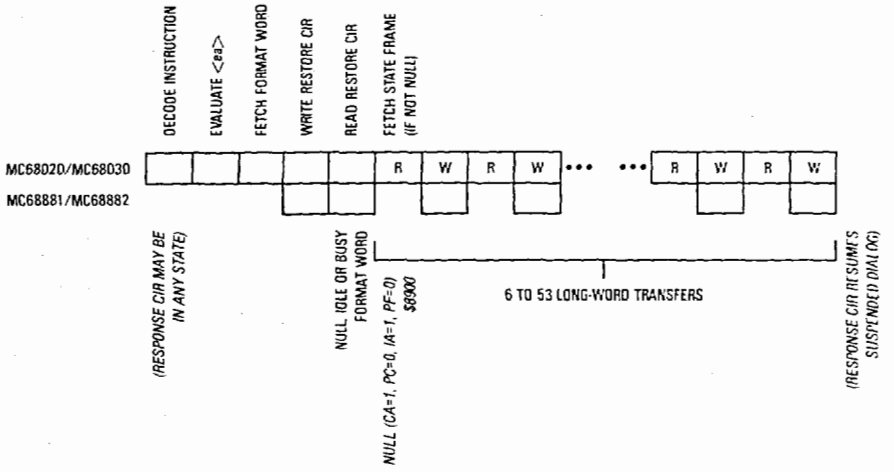


Figure 7-26. FRESTORE Instruction Dialog

The main processor initiates this dialog by evaluating the effective address, fetching a format word from that address, and writing the format word to the restore CIR. The main processor then reads the restore CIR to verify that the format word is valid. During this read cycle, the FPCP returns a format word that indicates if the format word that was written is valid for the current revision of the device. If the format word is valid, the same value that was written is read back from the restore CIR, and the main processor proceeds to transfer the state frame appropriate for the format word. The state frame size is 0, 6, or 45 long words for the current implementation of the MC68881. For the MC68882, the corresponding state frame sizes are 0, 14, and 53 long words. The invalid format word may also be returned as discussed in **7.5.4.7 FORMAT EXCEPTION, FRESTORE INSTRUCTION**.

Note that after the state restore operation is complete, the FPCP is in the state of the instruction that was previously suspended with an FSAVE instruction.

7.5.4 Exception Processing

This group of dialogs is actually a set of special cases of the dialogs described previously; they are grouped here for quick reference, and to simplify the preceding discussions. For each of the exception processing dialogs, only the differences from the normal instruction dialogs shown previously are discussed here. Also, it should be noted that these dialogs do not include all exception processing sequences that involve the FPCP; they only include those exceptions that are directly related to the coprocessor interface operation. For example, main processor detected F-line exceptions are not included since no coprocessor interface dialog occurs as part of the exception processing for this type of an exception.

The dialog for the coprocessor protocol violation exception is also omitted from the following diagrams. This is because these exceptions are not expected to occur during the normal operation of a fully debugged system, and the dialog is not readily predictable (either before or after the protocol violation occurs). By definition, the cause of the exception for main processor detected protocol violations is a hardware failure (since the FPCP cannot return an illegal response primitive).

For FPCP-detected protocol violations, the cause is most likely a software failure that causes a new instruction to be initiated before the previous instruction dialog is completed. In this case, the new instruction dialog is aborted immediately, but the previous instruction dialog may not terminate for some time. (The previous dialog may be completed incorrectly, since the protocol violation is never reported to the previous instruction.)

7.5.4.1 TAKE PRE-INSTRUCTION EXCEPTION. This dialog is utilized by the MC68881 when an arithmetic (OPCLASS 000, 010, or 011) or conditional instruction is initiated and an arithmetic exception is pending from a previous instruction, or when the main processor writes an undefined, reserved command word to the command CIR. In either case, this dialog consists of two write cycles and one read cycle, as shown in Figure 7-27. First, the main processor attempts to initiate a new instruction by writing to the command CIR; it then reads the response CIR to determine the next required action. The MC68881 returns the take pre-instruction exception in this case, indicating the appropriate vector number.

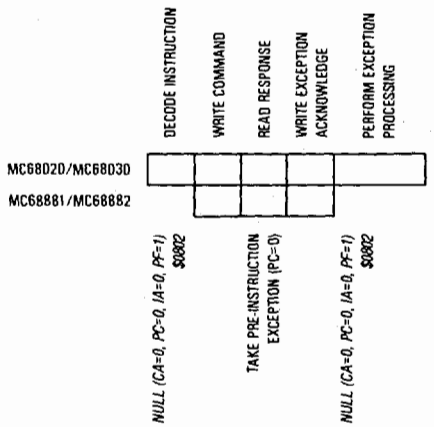


Figure 7-27. Take Pre-Instruction Exception Dialog — MC68881

The main processor services this primitive by writing an exception acknowledge to the control CIR and initiating exception processing.

Note that the write of the exception acknowledge causes the response CIR encoding to be changed to the null primitive, thus assuring that the take exception primitive is received by the main processor while avoiding spurious request primitives in non-MPU based systems.

The MC68882 uses a similar dialog for pre-instruction exceptions. However, when the main processor writes an exception acknowledge to the control CIR, the MC68882 does not enter the idle state. Instead, the MC68882 retains the take pre-instruction exception primitive in its response register. Any floating-point instruction other than an FSAVE (or an FRESTORE of the null state) reports the same exception again. An FSAVE (or an FRESTORE of the null state) restores the MC68882 to an idle state, allowing subsequent floating-point instructions to execute. Figure 7-28 shows the dialog which includes the minimum instructions recommended for an exception handler. (Refer to **5.2.2 Exception Handler Code**.)

Figure 7-29 shows the dialog that usually occurs when the handler for a pre-instruction exception does not begin with an FSAVE instruction. A protocol violation could occur; in which case, the dialog would not apply. Otherwise, as the dialog shows, at the completion of the exception handler routine, the main processor attempts to execute the same instruction again, and that instruction takes the same exception. If the exception handler included a floating-point instruction but no preceding FSAVE instruction, the floating-point instruction would take the original exception again, nesting the repetitions of the exception handler.

Figure 7-30 shows the dialog that applies when an exception handler does not set the exception pending bit, even though it begins with an FSAVE instruction and closes with an FRESTORE. A protocol violation cannot occur in this case, but because the exception pending bit is not set, the instruction again takes the exception when it is reinitiated after returning from the exception handler.

7.5.4.2 TAKE MID-INSTRUCTION EXCEPTION. The MC68881 uses this dialog only if an exception occurs during the execution of the FMOVE FPm,<ea> instruction. In this case, the protocol for the normal execution of the instruction is followed, and then the mid-instruction exception is reported with the last primitive (in lieu of the null primitive normally used to terminate the dialog). The main processor services this primitive by writing an exception processing acknowledge to the control CIR and initiating exception processing.

The dialog for this operation is shown in Figure 7-31. (For simplicity, this diagram assumes that the destination data format is not packed decimal with a dynamic k factor.) Note that a write of the exception acknowledge causes the response CIR encoding to be changed to the null primitive, thus assuring that the take-exception primitive is received by the main processor while avoiding a spurious request primitive in non-MC68020 or non-MC68030 based systems.

The MC68882 uses the mid-instruction exception if an exception occurs in the FMOVE FPm,<ea> instruction, as the MC68881 does. However, since the MC68882 allows concurrent execution of floating-point instructions, an instruction that has begun execution can report an exception caused by a previous instruction. When the previous instruction

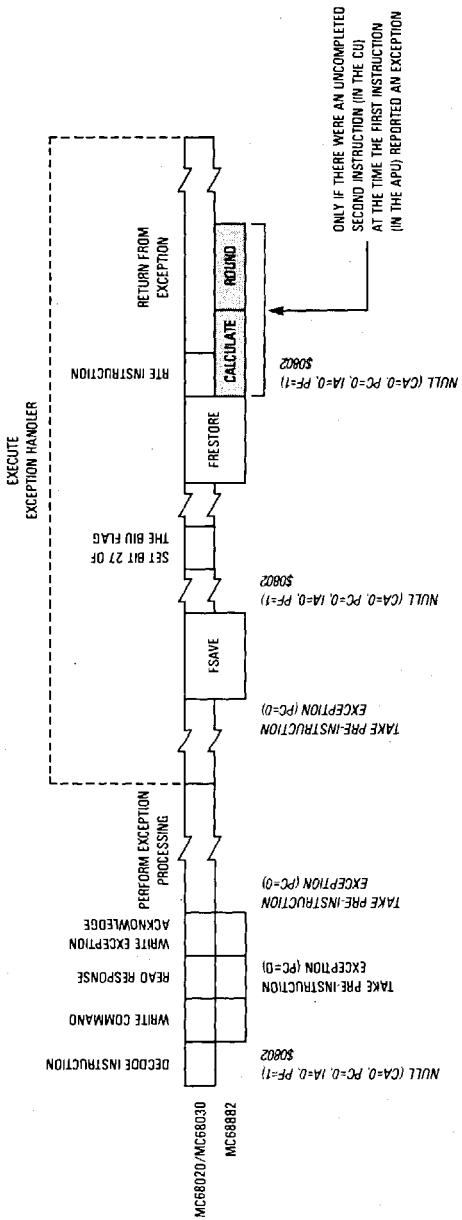


Figure 7-28. Take Pre-Instruction Exception Dialog — MC68882

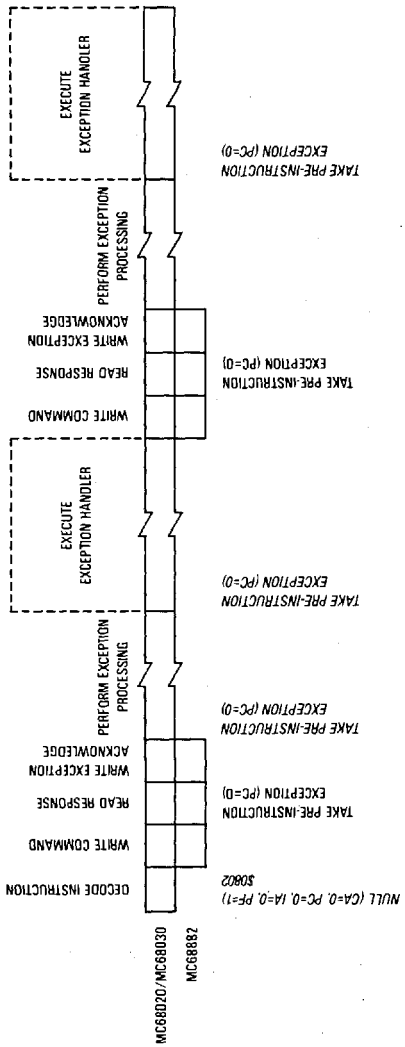


Figure 7-29. Take Pre-Instruction Exception Dialog — MC68882 with No FSAVE Instruction in the Handler

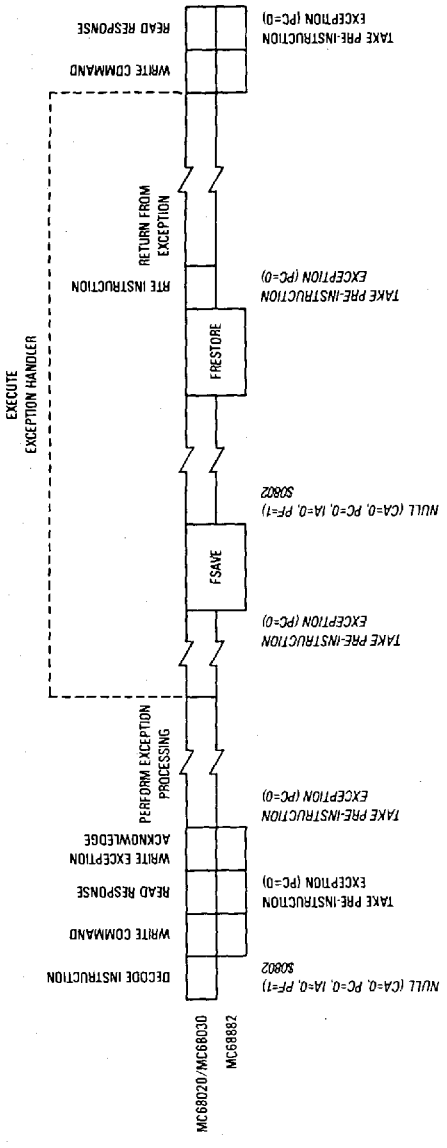


Figure 7-30. Take Pre-Instruction Exception Dialog — MC68882 with No BSET Instruction in the Exception Handler

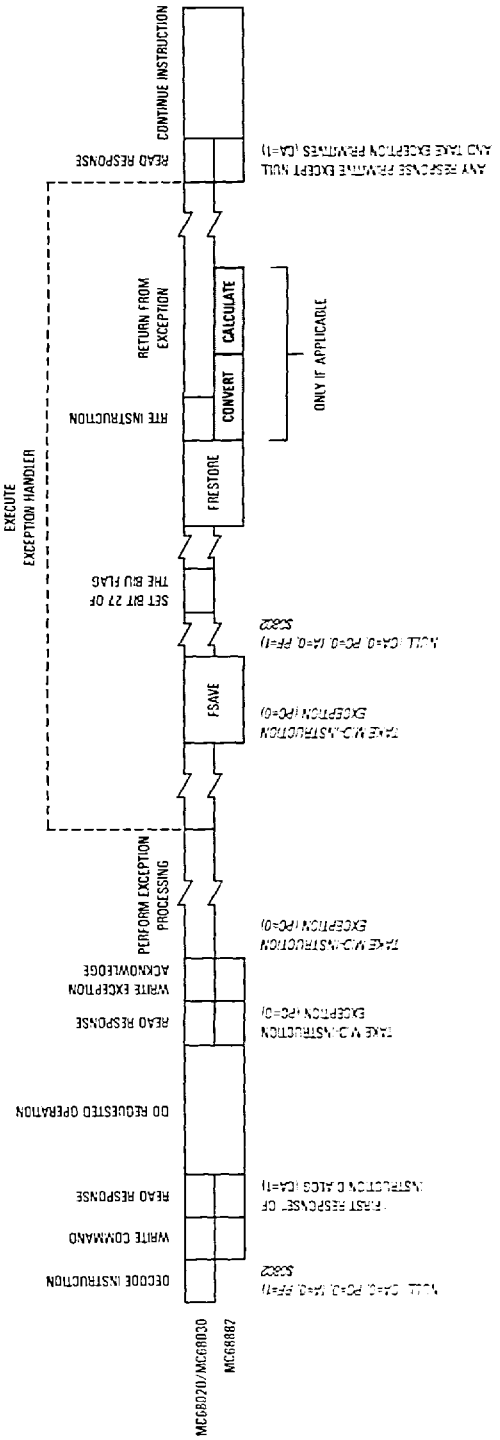


Figure 7-31. Take Mid-Instruction Exception Dialog — MC68881

makes an exception pending, the exception is reported on the next read operation of the response CIR. Therefore, all reads of the response CIR must be ready to take an exception. If the read of the response CIR occurs in the middle of an instruction, a take mid-instruction exception is taken. The dialog shown in Figure 7-32 is a generic case that applies to all instructions and to every read of the response CIR after the instruction has issued its first response. The first response is the response issued by the conversion unit (CU) or the arithmetic processing unit (APU) to the bus interface unit (BIU), not the null primitive (CA = 1, IA = 1) at the beginning of an instruction when (in an MC68881) the APU is busy or (in the MC68882) the CU is busy.

The MC68882 dialog is similar, except that the exception handler must begin with an FSAVE instruction. The significant difference is that the write exception acknowledge operation does not cause the MC68882 to return a null primitive. The FSAVE instruction of the exception handler changes the primitive to a null primitive.

In the MC68882, an instruction that is executing in the arithmetic processing unit (APU) concurrently with another instruction in the conversion unit (CU) may cause an exception. The instruction in the CU reports the exception as a mid-instruction exception when it completes. Figure 7-32 shows the dialog for this case using a general instruction dialog. When the previous instruction causes an exception, the read response CIR operation for the current instruction reads a take mid-instruction primitive, and the main processor performs exception processing. As in the nonconcurrent case, the write exception acknowledge operation does not cause the MC68882 to return a null primitive. The FSAVE instruction of the exception handler changes the primitive to a null primitive. Figure 7-33 shows the same case but with a specific instruction (FMOVE FPm,<ea>) in the CU.

When the exception handler does not contain an FSAVE instruction, the take mid-instruction exception primitive is not replaced by a null primitive, and the next floating-point instruction takes the exception again. Figure 7-34 shows the dialog for a mid-instruction exception that uses an incomplete handler.

Figure 7-35 shows the concurrent case using an exception handler that does not include the BSET instruction. Following the return from the exception handler, the main processor reads the take mid-instruction exception primitive from the response CIR and performs exception processing again for the same exception.

7.5.4.3 MID-INSTRUCTION INTERRUPT. This dialog is utilized by the FPCP only if an interrupt is pending during the calculation phase of the FMOVE FPm,<ea> instruction. In this case, the protocol for the normal execution of the instruction is followed except that the main processor performs exception processing for the interrupt, executes the interrupt handler, and returns to the point where the dialog was suspended in the middle of the execution of the instruction by the FPCP. From the perspective of the FPCP, the dialog appears to follow the normal sequence, with a long delay between successive reads of the response CIR by the main processor.

The dialog for this operation is shown in Figure 7-36. (For simplicity, this diagram assumes that the destination data format is not packed decimal with a dynamic k factor.) Note that it is possible (even probable) that the conversion of the output operand is completed before the main processor returns from the interrupt. Thus, the response CIR is prepared to return the evaluate effective address and transfer data primitive as soon as the main processor returns. Since the read of this primitive causes the FPCP to discard it and change the

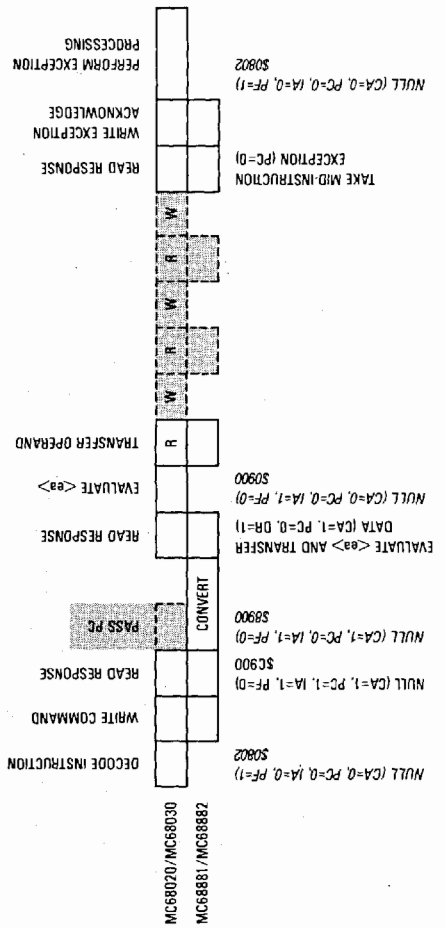


Figure 7-32. Take Mid-Instruction Exception Dialog — MC68882 General Concurrent Case

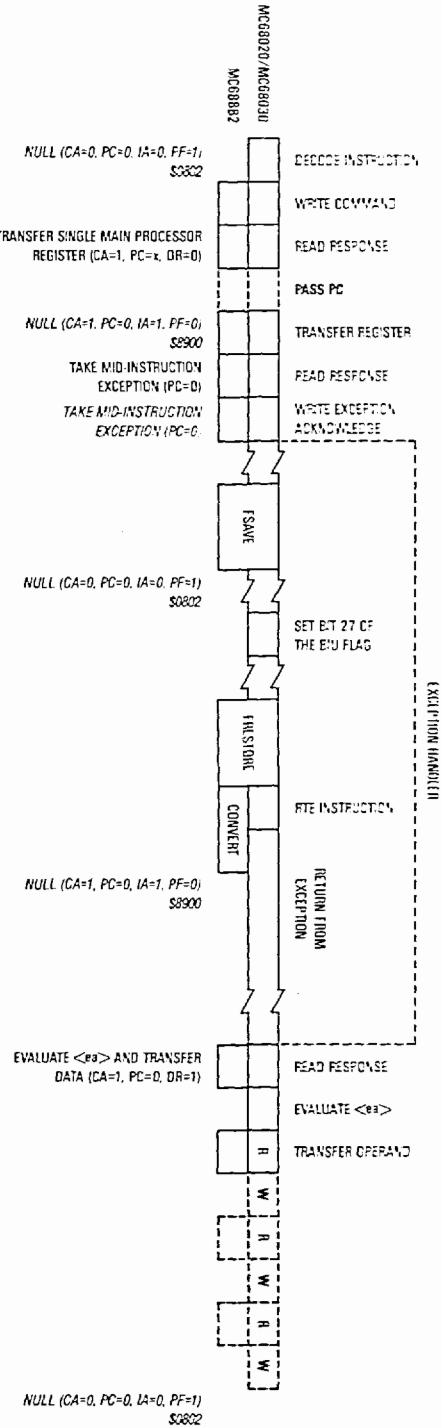


Figure 7-33. Take Mid-Instruction Exception Dialog — MC68882 FMOVE Concurrent Case

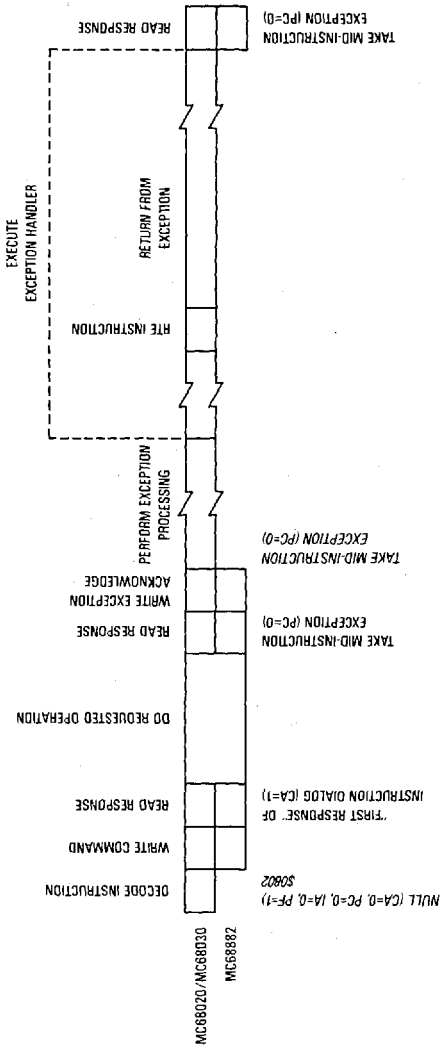


Figure 7-34. Take Mid-Instruction Exception Dialog — MC68882 with No FSAVE Instruction in the Handler

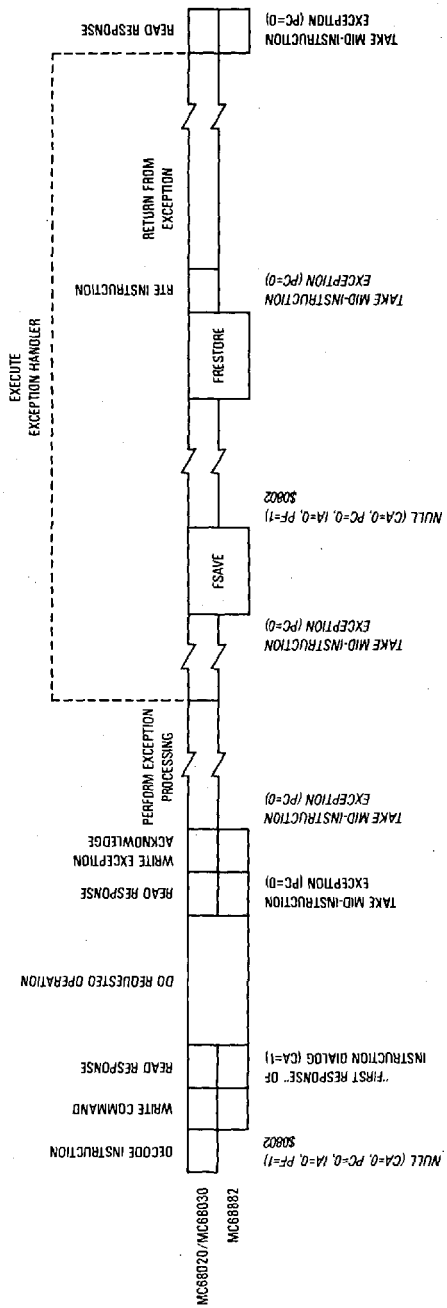


Figure 7-35. Take Mid-Instruction Exception Dialog — MC68882 with No BSET Instruction in the Exception Handler

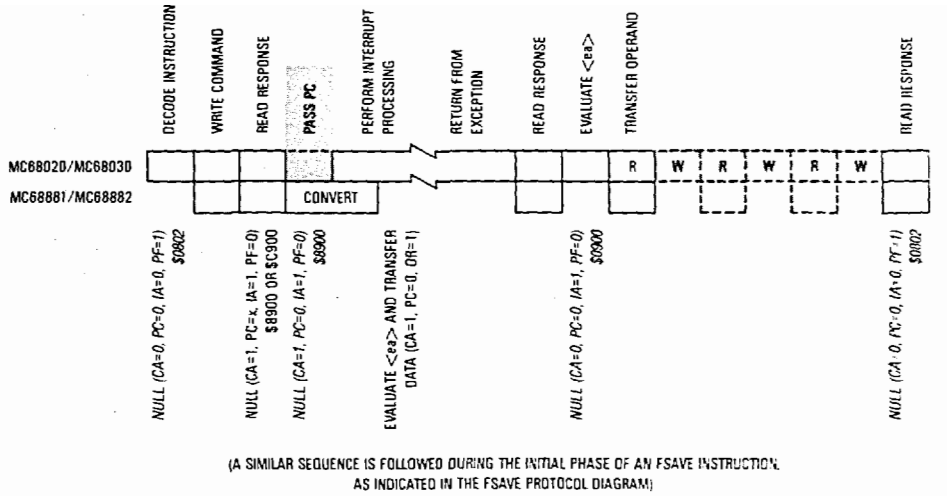


Figure 7-36. Mid-Instruction Interrupt Dialog

response CIR encoding to the null primitive, the interrupt handler (or any other routine) must not casually read the response CIR to determine the status of the FPCP, or the suspended protocol is disrupted. Rather, the only valid method for checking the status of the FPCP is to execute the FSAVE instruction and examine the state frame that is generated; followed by an FRESTORE instruction to reinstate the previous context of the FPCP.

7.5.4.4 TAKE BSUN EXCEPTION. This dialog is utilized by the FPCP when a conditional instruction is initiated by writing one of the IEEE nonaware conditional predicates to the condition CIR with the SNAN enable bit and the NAN condition code bit set. The dialog is shown in Figure 7-37.

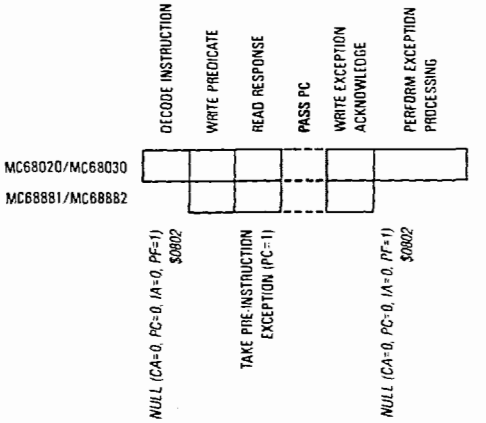


Figure 7-37. Take BSUN Exception Dialog

When the main processor reads the response CIR to receive the true/false result of the conditional evaluation, the FPCP returns the take pre-instruction exception primitive instead of the null primitive. In order to update the FPIAR, the PC bit of this primitive is also set. The main processor services this primitive by transferring the program counter, writing an exception acknowledge to the control CIR, and then initiating exception processing. Although the MC68020 or MC68030 always performs the program counter transfer when it is requested, other main processors may choose to ignore this request from an MC68881 without incurring a protocol violation. The MC68882 returns a protocol violation whenever the main processor ignores a request for transfer of the program counter.

Note that the write of the exception acknowledge causes the response CIR encoding to be changed to the null primitive, thus assuring that the take exception primitive is received by the main processor while avoiding spurious request primitives in non-MC68020 or non-MC68030 based systems.

7.5.4.5 F-LINE EMULATOR EXCEPTION. This dialog is utilized by the FPCP when a general instruction is initiated, and the value written to the command CIR is not a legal FPCP command word encoding. In this case, the dialog consists of two write cycles and one read cycle, as shown in Figure 7-38. First, the main processor attempts to initiate a new instruction by writing to the command CIR; it then reads the response CIR to determine the appropriate action to be taken. In this case, the first read of the response CIR returns a take exception primitive with the F-line emulator vector number. The main processor services this primitive by writing an exception acknowledge to the control CIR and initiating exception processing.

Note that the write of the exception acknowledge causes the response CIR encoding to be changed to the null primitive while avoiding spurious request primitives in non-MC68020 or non-MC68030 based systems.

7.5.4.6 FORMAT EXCEPTION, FSAVE INSTRUCTION. This dialog is utilized by the FPCP when an FSAVE or FRESTORE instruction dialog is interrupted by an attempt to initiate a

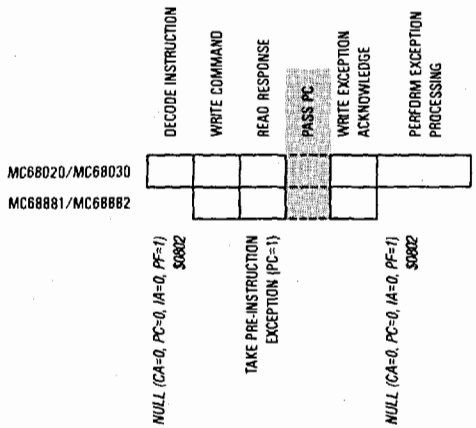


Figure 7-38. Take F-Line Emulator Exception Dialog

new FSAVE instruction (by reading from the save CIR). In this case, the FPCP returns the invalid format word to signal the illegal nesting of the FSAVE instruction. The main processor services this format word by writing an abort to the control CIR and initiating exception processing. The dialog for this operation is shown in Figure 7-39.

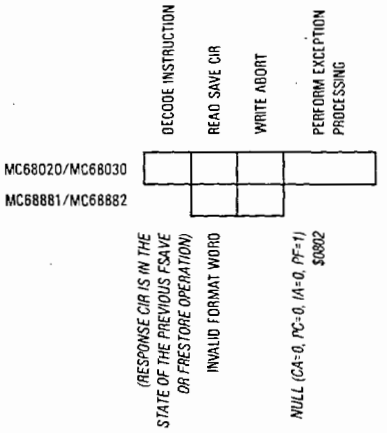


Figure 7-39. FSAVE Format Exception Dialog

Since the MPU writes an abort to the FPCP in response to the illegal format word, the FSAVE or FRESTORE that was interrupted by the nested FSAVE is destructively aborted with no indication to the suspended instruction of this occurrence. Thus, a suspended save operation continues to read the "frame" from the operand CIR if it is resumed, even though the data in the operand CIR is not valid. Likewise, a suspended restore operation writes the remainder of the frame to the operand CIR if it is resumed, even though the data written is ignored and the restore operation is not performed. Due to the destructive behavior of a nested FSAVE instruction, programmers must be certain that the FPCP is not executing an FSAVE or FRESTORE instruction prior to an attempt to execute a new FSAVE instruction. If there is a possibility that a nested FSAVE might occur, the MPU MOVES instruction might be used to read the save CIR before the FSAVE is executed. If the value returned from the save CIR is the illegal format word, then the new FSAVE should be postponed. Reading the save CIR in this manner is not destructive.

7.5.4.7 FORMAT EXCEPTION, FRESTORE INSTRUCTION. This dialog is utilized by the FPCP when an FRESTORE instruction is initiated by writing an invalid format word value to the restore CIR. (In this context, the term invalid format value refers to any value that is not a null, idle, or busy format word value recognized by the FPCP.) In this case, the FPCP returns the explicit invalid format word ($02xx) to signal the unrecognized format word value. The main processor services this format word by writing an abort to the control CIR and initiating exception processing. The dialog for this operation is shown in Figure 7-40. Note that this is a destructive exception since any instruction that was executing is aborted when the FRESTORE instruction is initiated. However, this should not be detrimental since a successful restore operation also aborts any previously executing instruction.

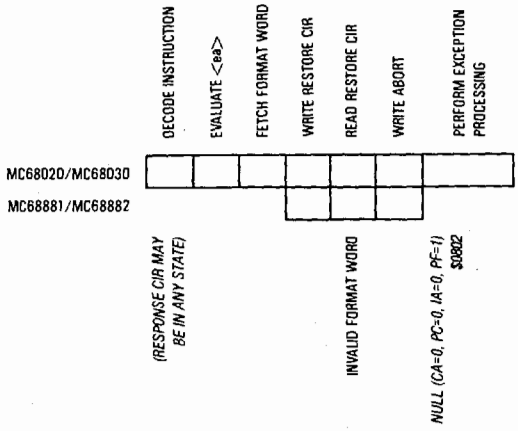


Figure 7-40. FRESTORE Format Exception Dialog

SECTION 8
INSTRUCTION EXECUTION TIMING

This section gives the instruction execution times for the MC68881/MC68882 (FPCP) in terms of external clock cycles. This section provides the user with some reasonably accurate execution timing guidelines, but not exact timings for every possible circumstance. This approach is used since the exact execution time for an instruction is highly dependent on such things as external data formats, input operand values, operand type combinations, and timing relationships with respect to the main processor. The timing numbers presented in the following tables allow the assembly language programmer or compiler writer to predict worst-case timings needed to evaluate the performance of the FPCP or to optimize code for concurrent execution. Also, the effect that various data formats and operand values have on execution times can be observed to allow optimizing data structures for the highest performance for a given application. Finally, the timings for exception processing, context switching, and interrupt processing are included so that designers of multitasking or real-time systems can predict such things as task switch overhead and maximum interrupt latency due to floating-point operations.

When binary real data formats are used and no register conflicts occur, the MC68882 performs the operand transfer and data conversion for most general type instructions concurrently with calculations for preceding instructions. This section includes timing information that shows the amount of instruction overlap this concurrency provides.

8.1. FACTORS AFFECTING EXECUTION TIMES

When investigating instruction execution timing for the FPCP, it is assumed that the following information is required in order to make informed engineering trade-offs:

- Best case instruction execution timings, for determining whether or not an FPCP-based system can meet certain data processing performance criteria.
- Worst-case instruction timings and how they affect execution concurrency, to allow programs and compilers to be optimized to take maximum advantage of overlap under any timing circumstances.
- Guidelines to indicate how various programming practices can be utilized to improve upon the worst-case execution times, and thus allow performance to approach, as closely as possible, the best case execution times for a given task.
- The effects that an FPCP might have on system-related timings such as context switch overhead time in multitasking systems, or interrupt latency time in a real-time system.

First of all, when defining the performance of any machine that can operate in an asynchronous manner, or where data dependencies affect execution times, a set of assumptions must be made in order to provide a measurable environment. In this manual, instruction execution times are shown in clock cycles to remove clock frequency dependencies, and the following assumptions apply to define the context of the times shown.

- The main processor is an MC68020, acting as the host to the FPCP, and the two devies use the same clock input.

- When the main processor initiates a command to the FPCP, any previous floating-point instruction has been completed and the FPCP is in the idle state.
- All operands in memory, as well as the system stack, are long word aligned.
- A 32-bit data bus is used for communications between the MC68020 and both the FPCP and the system memory.
- All memory accesses occur with no wait states (i.e., three clock cycle reads and writes).
- All coprocessor accesses, except those to the response and save CIRs, occur with no wait states. Accesses to the response and save CIRs require two wait cycles (five clock reads).

Note that the clock signal relationship between the MC68020/MC68030 (MPU) and the FPCP assumed for these discussions is not a system requirement, but merely a simplification that allows easy measurement of instruction times. However, the ratio of MPU clock frequency to FPCP clock frequency can be any reasonable value. In general, the clock frequency of the FPCP affects absolute instruction timing more than that of the MPU, since floating-point operations are usually computation intensive. However, the clock frequency relationship of the MPU and FPCP can affect the execution time of an instruction due to the time needed to transfer operands of various sizes and due to actual activity of the two devices. The magnitude of the dependency of execution times on the clock frequency of the MPU varies with instruction types, since some instructions spend a relatively small amount of their overall execution time in communication with the main processor; whereas, other instructions spend almost all of their execution time in communication with the main processor.

With this set of assumptions as a starting point, several factors must be defined that contribute to the overall execution time for a given instruction. Some of these factors are common to all instructions, while others are only applicable to certain instructions or data types. Particularly, the execution times for the conditional and system control instructions are not widely variable, but the execution time for an arithmetic or data movement instruction is heavily affected by data values and exception checking. In order to better understand how these factors are combined to calculate the execution time for an arithmetic or move-to-floating-point register instruction, it is helpful to divide coprocessor instruction execution into the following steps:

1. Receive the command word from the host processor, decode it, and return the first service request primitive.
2. Receive the main processor program counter, if required.
3. Receive an external operand, if required.
4. Convert the operand to the internal extended format.
5. Perform the algorithm specified by the command word on the operand(s).
6. Round the result to the correct precision, check the result of the computation for conditions such as overflow, then store the result into a floating-point data register.

The first three of these steps require approximately the same amount of time for any instruction, but the last three steps can require widely varying amounts of time even when comparing the execution time for a given instruction with different data inputs. For purposes of this discussion, the first three steps are referred to as the instruction start-up phase, the fourth step as the conversion phase, the fifth step as the calculation phase, and

the sixth step as the round/store phase. The following paragraphs discuss the factors that affect the execution time of an arithmetic instruction during each of these phases.

8.1.1 Instruction Start-Up Phase

The factor that affects execution time most heavily during this phase of an instruction is the location and format of an external oprand. The three possible locations for an input operand are:

1. In a floating-point data register,
2. In a main processor data register, and
3. In external memory.

If an operand resides in a floating-point data register before an instruction starts, no data movement operation is required to prepare it for the calculation phase, and thus, the start-up phase is very short. If an operand resides in a main processor data register, the FPCP uses the evaluate effective address and transfer data response primitive to request it from the MPU. In this case, the MPU does not generate any operand memory cycles, and the operand is transferred to the FPCP with a single bus cycle. The FPCP then converts the signed integer or single precision floating-point number to extended precision and proceeds to the calculate phase.

For the third operand location case, execution time can vary widely due to two separate mechanisms, the addressing mode and alignment of the operand in memory, and the data format and value of the operand. The addressing mode used to locate an operand affects execution time in a straightforward manner due to the fixed nature of effective address calculations by the MPU. For example, if the addressing mode used is address register indirect, (An), then no instruction prefetch words and one long-word indirect address fetch may be required to calculate the final address of the operand. Then, once the operand is located, up to three long word fetches may be required to transfer the operand to the FPCP. The execution times for these operations are quite predictable (i.e., there are no data dependencies involved), although they are affected by instruction stream alignment, MPU cache hits, memory access times, memory width, and operand alignment. As mentioned earlier, certain assumptions are made with regard to these factors (for the purposes of this discussion) so that the tables in this section may be simplified. In order to include the effects of these factors, refer to the MC68020 user's manual or the MC68030 user's manual for more information regarding bus operation.

The second mechanism that can affect execution times for operands in memory is the data format. For the integer and binary floating-point formats, the execution times for conversions required to prepare the operand for the calculation are relatively free from data dependencies. However, for the packed decimal floating-point format, execution times can vary significantly due to the value of the input operand.

8.1.2 Calculation Phase

This is the most volatile portion of an instruction with respect to execution times. The main factor that affects the calculation time is the operation to be performed (e.g., a sine operation requires far more time than an add operation), but for a given operation, the execution time is data dependent. For the monadic operations, the data dependency is limited to the type and value of the input operand; for the dyadic operations, the combination of the

types and values of the two operands can also affect execution time (in this context, data type refers to the FPCP-extended precision representation of one of the five IEEE data types: normalized, denormalized, zero, infinity, and not-a-numer). Because execution times vary due to data values and type combinations, Tables 8-14 and 8-15 indicate the execution time for each arithmetic operation with typical arguments, along with timing values for special case operand types such as zero, infinity, etc.

8.1.3 Round/Store Result Phase

The execution time for this phase of an instruction is dependent on the mode of operation that is programmed into the floating-point control register, as well as the value of the result. For example, if the rounding precision is programmed to be extended, execution is faster than if it is single. Also, if the result of a calculation overflows or underflows the destination precision, then more time is required to handle that exception. In the following paragraphs, the overall execution times for the arithmetic operations assume the best case round/store phase time. Table 8-16 lists the values used to calculate execution times for various rounding precisions and exception handling operations.

8.2 CONCURRENT INSTRUCTION EXECUTION

An important factor that should be considered when optimizing MPU and FPCP programs is the amount of concurrent execution time that an instruction allows. It is also an important consideration in calculating overall execution time.

Concurrency between MPU and the FPCP applies when the main processor executes MPU instructions while the coprocessor completes execution of a floating-point instruction. The MC68882 can execute two floating-point instructions concurrently, providing additional concurrency not available in the MC68881.

Overlap time between instructions determines the degree of concurrency that is possible. Overlap time is derived from the combination of the tail of an instruction with the head of the next instruction, where tail and head are portions of the total execution time of an instruction. The tail is the portion of the total execution time during which another instruction can be executed. The head is the portion of the total execution time that can be performed while another instruction is completing. The overlap time of two consecutive instructions is either the tail of the first instruction or the head of the second, whichever is less.

The tail of a floating-point instruction is the time during which the MPU can execute a subsequent instruction. It consists of the time during which the processor releases the MPU to allow a subsequent instruction to begin. During this period, the coprocessor is still performing the calculations necessary to complete the current instruction.

In the case of the MC68881, overlap occurs only when the subsequent instruction is an MPU instruction. Table 8-25 is used to calculate the portion of the MC68881 instruction that can overlap with an MPU instruction. If the subsequent instruction is a floating-point instruction, the MPU is requested to wait to execute it until the coprocessor finishes the current instruction. That is, the head portion of the execution time for a floating-point instruction executing in an MC68881 is zero.

The MC68882, however, can obtain the operand of a subsequent floating-point instruction and convert the operand to internal format during the tail of the previous instruction. This portion of the instruction is defined as the head of the MC68882 instruction. It is the portion of the instruction that begins when the instruction is initiated by the MPU, and ends when the present coprocessor instruction can no longer operate under the tail of a previous instruction. The actual values of head and tail that apply to the MC68882 floating-point instructions are shown in Table 8-3. The tails of MC68882 floating-point instructions can execute concurrently with MPU instructions as well as with other MC68882 instructions. The head times for the MC68882 instructions indicate the degree of concurrency that is allowed.

Each floating-point instruction of the MC68882 is either fully concurrent, partially concurrent, or nonconcurrent. Instructions for which the head time equals the total execution time are fully concurrent. Those for which head and tail values are shown are partially concurrent. The instructions that have zero head values are noncurrent.

Concurrent execution of floating-point instructions in the MC68882 can significantly improve coprocessor performance. Refer to **5.1.2 Optimization of Code for the MC68882** for more information on the effects of concurrency on the performance of programs.

8.3 INTERRUPT LATENCY TIMES

In real-time systems, a very important factor pertaining to overall system performance is the response time required for a processor to handle an interrupt. In the M68000 Family of processors, interrupts are allowed to be asserted to the processor asynchronously, and they are handled on the next instruction boundary. While the average interrupt latency for the MPU is quite short, the maximum latency is often of critical importance since real-time interrupts cannot require servicing in less than the maximum interrupt latency. The maximum interrupt latency for the MPU is approximately 250 clock cycles (for the MOVEM.L ([d32,An],Xn,d32), D0–D7/A0–A7 instruction where the last data fetch is aborted with a bus error; refer to the MC68020 user's manual or the MC68030 user's manaul for more detailed information), but some FPCP instructions may take two or three times that long to execute with typical operand types, combinations and values.

It may be unacceptable in a real-time system to have a worst-case interrupt latency time as large as 600 or more clock cycles (the length of some long floating-point instructions). Therefore, the FPCP allows interrupts to be processed in the middle of the execution of a floating-point instruction, whenever possible, to reduce the latency time. The FPCP does this in four ways:

1. By returning the null (CA=0, IA=1, PF=0) primitive when it enters the calculated phase of an instruction that allows concurrency. If the MPU is not in the trace mode, it is then free to fetch the next instruction and process any pending interrupts at the instruction boundary. If the MPU is in the trace mode, it waits for the FPCP to complete execution and return the null (CA=0, PF=1) primitive before continuing with the next instruction, but it services pending interrupts while it is waiting.
2. By returning the null (CA=1, IA=1) primitive when the main processor attempts to initiate a floating-point instruction while the FPCP is unable to begin another operation, thus allowing the MPU to service interrupts while waiting for the coprocessor to start execution of the new instruction.

3. By returning the null (CA=1, IA=1) primitive during internal conversions for nonconcurrent instruction execution (e.g, FMOVE.<fmt>FPn,<ea> in the MC68881 or FMOVE.W FPn,<ea> in the MC68882) before returning service request primitives to complete the operation.

4. By returning the not-ready, come-again format code during internal operations required by the FSAVE instruction. The FPCP returns this format code in some cases (as described in **6.4 CONTEXT SWITCHING**) to enable it to store a smaller state frame, and the MPU can process interrupts while waiting for the save operation to begin.

For the first two cases, the MPU is allowed to process interrupts during the tail period defined in **8.2 CONCURRENT INSTRUCTION EXECUTION**. For the third case, the period during which the MPU can process interrupts is illustrated in Figure 8-1. The timing for the fourth case is similar to the third case, except that the periods labeled "Convert," "Round," and "Transfer" for the FPCP are not used for those purposes but instead for saving of the internal state.

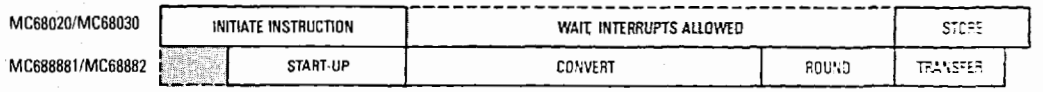


Figure 8-1. Nonconcurrent Instruction Execution, Interrupts Allowed

Basically, the maximum interrupt latency time for any FPCP instruction is equal to the worst-case execution time minus the interrupts allowed time, where both of these values are calculated using the tables in this section. For concurrent instructions, the execution time and allowed concurrency times are shown, and the interrupt latency is the difference between these two values. For nonconcurrent instructions, the amount of time during which interrupts are allowed is shown in the tables as the number of allowed overlap clock cycles, and the interrupt latency is approximately equal to the total execution time minus the allowed overlap time. However, as shown in Figure 8-1, there may be two separate time periods during which the MPU is not allowed to process interrupts. For some instructions, such as the FMOVE.P FPn,<ea> instruction, these two periods are approximately equal and make up a small fraction of the overall execution time for the operation. On the other hand, for the FRESTORE and FSAVE instructions, the time required to transfer a busy state frame is roughly equal to the overall execution time. In fact, the worst-case interrupt latency due to an FPCP instruction is for the FRESTORE instruction with a busy state frame.

8.4 COPROCESSOR INTERFACE OVERHEAD

For all of the instruction timings shown in the following tables, all coprocessor interface bus cycle timing and associated processing are included in the overall execution times. However, it is assumed that when the main processor beings execution of a floating-point instruction, the FPCP has completed execution of any previous instruction and is ready to begin a new instruction. (Note that the criteria for determining the readiness of the MC68881 is different from that of the MC68882. The MC68881 is ready to begin an instruction if the previous instruction is completed. The MC68882, however, is ready to begin an instruction if the instruction has completed executing in the CU and the CU has handed off the instruction to the APU.) Thus, the MPU is never required to wait while the FPCP completes

an instruction. Also, it is assumed that when the MPU is waiting for the FPCP during a nonconcurrent instruction, the main processor reads the response register at exactly the moment when the FPCP is prepared to return a service request primitive to complete or continue the instruction. If these conditions are not met, the actual instruction execution time can be shorter or longer than the values shown in the tables, due to synchronization of the two devices.

First, it must be noted that the FPCP does not begin execution of an instruction until the start of the read cycle from the response CIR in which the first primitive of the instruction dialog is returned to the main processor. If the MPU attempts to initiate a floating-point instruction before the previous one has completed and the coprocessor is ready, the FPCP queues the command word or conditional predicate and then instructs the MPU to wait (by encoding the null (CA=1, IA=1) primitive in the response CIR) until the previous instruction is completed and the FPCP is ready to begin the next instruction. When the previous instruction has completed execution, the FPCP does not begin execution of the queued instruction until the next read of the response CIR. The sequence of events for this situation is:

1. The FPCP allows concurrent instruction execution by returning the null (CA=0, IA=1, PF=0) primitive to the MPU.
2. The MPU encounters the next FPCP instruction and attempts to initiate execution by writing to the command or condition CIR. The MPU then starts a read from the response CIR to determine what further action should be taken.
3. The FPCP queues the instruction initiation request and changes the encoding of the response CIR to null (CA=1, IA=1), causing the MPU to wait.
4. The MPU continues to read the response CIR repeatedly until a new primitive is encoded or an interrupt becomes pending (if an interrupt occurs, the MPU resumes polling of the response CIR after the interrupt handler executes an RTE instruction).
5. The MC68881 APU becomes idle (by completing the previous instruction) and waits for the next read of the response CIR. In the MC68882, the CU hands off the instruction to the APU after the APU completes the previous instruction and waits for the next read of the response CIR.
6. The MPU reads the response CIR, which either results in the return of a take exception primitive (due to an exception during the previous instruction) or causes the FPCP to begin execution of the new instruction by returning the first primitive required for that operation.

The timing relationship of the main processor and the FPCP during this sequence can affect the overall execution time of the new instruction, due to synchronization between the two devices. Specifically, if the MPU begins a read of the response CIR exactly one clock cycle before the FPCP completes the execution of the previous instruction in the APU, the FPCP immediately begins execution of the new instruction by returning the first primitive of the new instruction dialog during that read cycle. This case is shown in Figure 8-2, which illustrates the best-case timing for coprocessor interface overhead: two clock cycles.

Figure 8-2 also illustrates the typical coprocessor interface overhead timing, which occurs when the MPU initiates a new instruction and the FPCP is in the idle state. For this case, there is no overlap with a previous instruction at the beginning of the instruction dialog, and the coprocessor interface overhead for the new instruction is 11 clock cycles. Also,

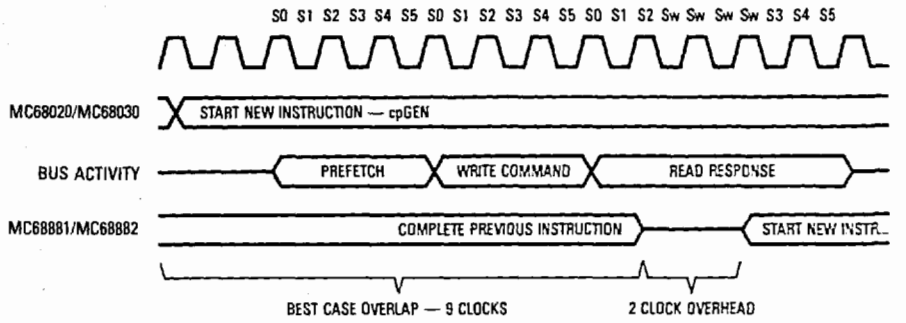


Figure 8-2. Best-Case Coprocessor Interface Overhead Timing

note that this example assumes that the instruction prefetch requested by the cpGEN start-up operation was not satisfied by the previous prefetch bus cycle, and it does not hit in the MPU on-chip instruction cache. Refer to **8.5.2 MC68881 Detail Timing Tables** for further discussion of the effects of instruction prefetching by the MPU.

If the read cycle to the response CIR occurs before the FPCP has completed execution of the previous instruction, the MPU processes the null (CA = 1, IA = 1) primitive that is returned (by checking for pending interrupts and re-reading the response CIR if there are none). This operation requires 10 MPU clock cycles, and thus there is a 10 clock cycle worst-case synchronization period that is part of the effective execution time for the new instruction. (The worst case occurs if the response CIR read cycle starts two clock cycles before the previous instruction in the APU is completed.) The worst-case timing is shown in Figure 8-3. The exact amount of synchronization time required is dependent on the system environment, such as the clock signal relationship between the MPU and the FPCP, and the context of an instruction sequence. In all of the following tables, the typical case of no overlapped execution is assumed; thus, a coprocessor interface overhead value of 11 clock cycles is included in the timing numbers. If an attempt is made to optimize an instruction sequence for overlapped execution, the coprocessor interface overhead may be reduced by as much as nine clock cycles. However, incorrect "optimization" may result in an 11 clock cycle overhead, which is no worse than the no overlap case previously described.

A similar overhead effect occurs for nonconcurrent instructions that allow interrupt processing by returning the not-ready format code (FSAVE instruction), or the null (CA = 1, IA = 1) primitive in the middle of instruction execution (i.e., the FSAVE and FMOVE FPn,<ea> instructions). In these cases, the FPCP completes as much of the instruction as possible while allowing interrupts, and then prepares to encode a valid format code or service request primitive in the save or response CIR during the next read by the MPU. The timing relationship between the start of the read cycle and the completion of internal operations by the FPCP is identical to the timing previously described for the instruction start-up phase. Thus, the same 10 clock cycle overhead factor might be added to the execution time for these instructions.

In the following tables, the assumptions stated earlier apply (i.e., the main processor is an MPU running on the same clock as the FPCP), and the coprocessor interface overhead for all operations other than instruction initiation is included based on those assumptions. If an instruction is executed under conditions other than those described, the execution time may be increased in increments of 10 clock cycles if necessary.

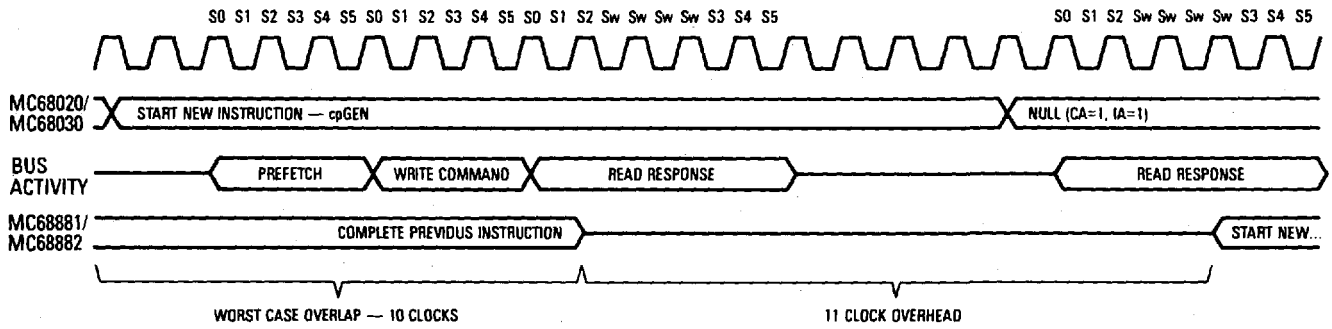


Figure 8-3. Worst-Case FPCP Interface Overhead Timing

8.5 EXECUTION TIMING TABLES

In the following paragraphs, timing tables are presented that allow the calculation of best-case, typical, and worst-case execution times for any FPCP instruction. These tables are based on the assumptions previously stated and include the total execution time for each instruction. In other words, the numbers that are calculated using these tables indicate the time from the beginning of execution of the coprocessor instruction by the MPU (i.e., when the instruction has been prefetched and loaded into the instruction decode register) to completion of the instruction by the FPCP and/or MPU (i.e., when a read of the response CIR indicates a null (CA=0, PF=1), when conditional processing is completed, or when the last operand transfer to or from the FPCP has been completed).

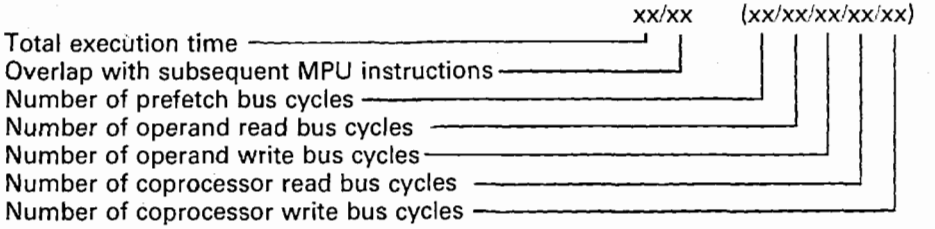
Bus cycle activity is also indicated by the tables and includes all bus cycles generated by a particular operation. Note that instruction prefetch and operand write cycles requested by the execution of a given instruction may not actually be executed during the excution of the instruction, but are queued by the MPU bus interface unit for completion as soon as the bus is available. (Refer to the MC68020 and MC68030 user's manual for more information on bus cycle overlap.) When a floating-point operation is completed, a prefetch request has been generated by the MPU to replace each word of the instruction stream used by the instruction or to refill the instruction pipe in the case of a conditional branch, a trap instruction, or an exception.

The timing information shown in the following tables for some operations includes three numbers that depend on the context of the instruction (i.e., the alignment of the instruction stream, whether the MPU instruction cache is enabled, and whether the cache contains the instruction.)

8

1. The best-case value, where prefetches hit in the MPU on-chip cache and the instruction benefits from the maximum overlap, in the MPU pipeline, with other instructions. Due to the highly volatile nature of the instruction pipeline, this case is not easy to achieve intentionally but occurs occasionally.
2. The cache-only case, where prefetches hit in the MPU on-chip cache, but the instruction does not overlap with preceding or following instructions.
3. The worst-case, where prefetches do not hit in the MPU on-chip cache or the cache is disabled, and there is no instruction overlap. It is further assumed that the instruction is aligned so that a prefetch is executed before the MPU writes to the FPCP command CIR.

The execution time entries in most of the following tables contain seven numbers. The left-most number is the total execution time for the operation in clock cycles, followed by the number of clock cycles of the total execution time that is allowed to overlap with execution of other operations by the main processor. Then, in parenthesis, the bus cycle activity is included, which indicates the number of instruction prefetch, operand read, operand write, coprocessor read, and coprocessor write bus cycles that are generated by the execution of the instruction. An example of the format of an entry from the timing table is:



The total number of clocks required for the bus activity in each entry can be derived by multiplying the total number of bus cycles by three. (This does not account for the fact that reads from the response and save CIRs require five clocks rather than three, but the two clock-cycle discrepancy is usually negligible compared with the overall execution time for an instruction.) For some instructions, the number of coprocessor read cycles indicated by the tables may not reflect the actual number of read cycles that are executed during the dialog for the instruction. This is because only the first occurrence of a series of null (CA = 1, IA = 1) or null (CA = 0, PF = 0, IA = 1) primitives is included in the tables. For example, the FPCP forces the main processor to wait during the conversion phase of the FMOVE FPn,<ea> instruction by using the null response primitive. The MPU may perform numerous response CIR read cycles during the time that it waits for the FPCP, but only the first of this series of read cycles is included in the timing table entry. Although this simplification may fail to indicate the true number of coprocessor read cycles executed by the MPU, it allows the tables to accurately indicate the minimum number of different response primitive and operand reads that must be executed by a main processor, regardless of its type or clock and bus speed.

The timing tables in the following paragraphs are divided into two major groups. First, several tables are presented that allow quick determination of the typical execution time for all instructions. These tables are comprehensive but assume typical operand inputs and operating conditions for simplicity. No more than two tables are used to determine the typical execution time for a given instruction. One table is used to determine the basic execution time for the selected instruction, and a second table (one of five listing the instruction groups) is used to determine the additional time required for the calculation of the effective address by the MPU, for those instructions that require an effective address calculation.

The second group of tables is used to calculate a more precise execution timing value for a specific instruction, addressing mode, and operand type combination than is available in the first group of tables. This group of tables is also useful for the calculation of execution times where the main processor is not a MPU, since the timing for each phase of instruction execution is included in a separate table. This alows timings that are only dependent on the FPCP to be calculated and added to the timing characteristic of the main processor.

8.5.1 Timing Tables for Typical Execution

This set of tables allows a quick determination of the typical execution time for any FPCP instruction when the MPU is used as the main processor. The first table presented is for effective address calculations performed by the MPU. Entries from this table are added to the entries in the other tables in this subsection, if necessary, to obtain the overall execution time for an operation. The assumptions that apply to the following tables are:

- The main processor is an MC68020 and operates on the same clock as the FPCP. Instruction prefetches do not hit in the MC68020 cache (or it is disabled), and the instruction is aligned so that a prefetch occurs before the command CIR is written by the MC68020.

NOTE

The timing numbers are derived assuming that the main processor is an MC68020. The MC68030 has a more optimized coprocessor interface and can benefit from the data cache hits. These improvements of the coprocessor interface are not used in determining typical operation. Actual operation when using the MC68030 always yields better values than the calculations derived from these tables.

- A 32-bit memory interface is used, and memory accesses occur with zero wait states. All memory operands, as well as the stack pointers, are long word aligned.
- Accesses to the FPCP require three clock cycles, with the exception of read accesses to the response and save CIRs, which require five clock cycles.
- No instruction overlap is utilized so the coprocessor interface overhead is 11 clocks. This can be reduced to two clock cycles if optimized code sequences are used or may be 11 clock cycles if overlap is attempted and a synchronization delay is required.
- Typical operand conversion and calculation times are used (i.e., input operands are assumed to be normalized numbers in the legal range for a given function).
- No exception is enabled, no exception occurs, and the default rounding mode and precision of round-to-nearest, extended precision is used.

8.5.1.1 EFFECTIVE ADDRESS CALCULATIONS. For any instruction that requires an operand external to the FPCP, an evaluate effective address and transfer data response primitive is issued by the FPCP during the dialog for that instruction. The amount of time that is required for the MPU to calculate the effective address while processing this primitive for each addressing mode, excluding the transfer of the data to the FPCP, is shown in Table 8-1. The times shown in this table include all bus cycles required to perform the address calculation (such an instruction prefetches and memory indirect fetches).

For the FMOVEM instruction, Table 8-1 is also used to determine the time required for the MPU to perform an address calculation (implied by the transfer multiple coprocessor registers primitive). For the FScc, FRESTORE, and FSAVE instructions, the request to evaluate an effective address is implied by the F-line instruction word; therefore, no response primitive is issued by the FPCP to request the evaluation. The following table is used for these three instructions to adjust the basic instruction execution time to reflect the addressing mode that is used.

Note that Table 8-1 applies only to the MPU effective address calculation time for coprocessor instructions. The execution times included in this table are not the same as the calculate effective address times given in the MPU user's manuals for normal instruction execution.

8.5.1.2 ARITHMETIC OPERATIONS. Three tables provide the typical instruction execution time for each arithmetic instruction. This group of instructions includes the majority of the FPCP operations such as FADD, FSUB, etc. In addition to the instructions that perform arithmetic calculations as part of their function, the FCMP, FMOVE, and FTST instructions are also included since an implicit conversion is performed by those operations. For memory operands, the timing for the appropriate effective addressing mode must be added to the numbers in these tables to determine the overall instruction execution times. In order to simplify these tables, the overall execution times for the MC68881 are listed in Table 8-2, the overall execution times for the MC68882 are listed in Table 8-3, and the bus cycle activity numbers are listed in Table 8-4. In addition to the total execution times for the MC68882, Table 8-3 lists the head and tail values required for calculating concurrency.

Table 8-1. Effective Address Calculations

Addressing Modes	Best Case	Cache Case	Worst Case
Dn or An	0/0 (0/0/0/0/0)	0/0 (0/0/0/0/0)	0/0 (0/0/0/0/0)
(An)	0/0 (0/0/0/0/0)	2/0 (0/0/0/0/0)	2/0 (0/0/0/0/0)
(An)+	3/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)
−(An)	3/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)
(d_{16},An) or (d_{16},PC)	0/0 (0/0/0/0/0)	2/0 (0/0/0/0/0)	3/0 (1/0/0/0/0)
(xxx).W	0/0 (0/0/0/0/0)	2/0 (0/0/0/0/0)	3/0 (1/0/0/0/0)
(xxx).L	1/0 (0/0/0/0/0)	4/0 (0/0/0/0/0)	5/0 (1/0/0/0/0)
#<data>	0/0 (0/0/0/0/0)	0/0 (0/0/0/0/0)	0/0 (0/0/0/0/0)
(d_8,An,Xn) or (d_8,PC,Xn)	1/0 (0/0/0/0/0)	4/0 (0/0/0/0/0)	5/0 (1/0/0/0/0)
(d_{16},An,Xn) or (d_{16},PC,Xn)	3/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	7/0 (1/0/0/0/0)
(B)	3/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	7/0 (1/0/0/0/0)
(d_{16},B)	5/0 (0/0/0/0/0)	8/0 (0/0/0/0/0)	9/0 (1/0/0/0/0)
(d_{32},B)	11/0 (0/0/0/0/0)	14/0 (0/0/0/0/0)	16/0 (2/0/0/0/0)
([B],I)	8/0 (0/1/0/0/0)	11/0 (0/1/0/0/0)	12/0 (1/1/0/0/0)
([B],I,d_{16})	8/0 (0/1/0/0/0)	11/0 (0/1/0/0/0)	12/0 (1/1/0/0/0)
([B],I,d_{32})	10/0 (0/1/0/0/0)	13/0 (0/1/0/0/0)	15/0 (2/1/0/0/0)
([d_{16},B],I)	10/0 (0/1/0/0/0)	13/0 (0/1/0/0/0)	14/0 (1/1/0/0/0)
([d_{16},B],I,d_{16})	10/0 (0/1/0/0/0)	13/0 (0/1/0/0/0)	15/0 (2/1/0/0/0)
([d_{16},B],I,d_{32})	12/0 (0/1/0/0/0)	15/0 (0/1/0/0/0)	17/0 (2/1/0/0/0)
([d_{32},B],I)	16/0 (0/1/0/0/0)	19/0 (0/1/0/0/0)	21/0 (2/1/0/0/0)
([d_{32},B],I,d_{16})	16/0 (0/1/0/0/0)	19/0 (0/1/0/0/0)	21/0 (2/1/0/0/0)
([d_{32},B],I,d_{32})	18/0 (0/1/0/0/0)	21/0 (0/1/0/0/0)	24/0 (3/1/0/0/0)

B = Base address; 0, An, PC, Xn, An+Xn, PC+Xn. Form does not affect timing.
I = Index; 0 or Xn.
NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn does not affect timing.

8.5.1.3 MC68882 CONCURRENT OPERATIONS. The MC68882 overall instruction timing table, Table 8-3, contains the H and T numbers, which are helpful in estimating the instruction execution overlap resulting from concurrent execution of floating-point instructions.

- H — Head. The effective address calculation should be added to the head to obtain the true head time. (This does not apply for FMOVE to memory if a register conflict occurs.)
- T — Tail. The period during which the MC68882 can begin another floating-point instruction.

The total execution time for a set of instructions is the sum of the overall execution times of the individual instructions in the set minus the total overlap time. This formula applies to both the MC68881 and the MC68882; for the MC68881, the overlap between floating-point instructions is zero.

Table 8-5 lists an example of the use of the timing tables to calculate the execution time for a sequence of instuctions. The table compares the execution times, in clock cycles, of the individual instructions and the total execution time for the sequence using the MC68881

Table 8-2. MC68881 Overall Execution Times

Instruction	FPn to FPm	Memory Source or Destination Operand Format				
		Integer	Single	Double	Extended	Packed
FABS	35	62	54	60	58	872
FACOS	625	652	644	650	648	1462
FADD	51	80	72	78	76	888
FASIN	581	608	600	606	604	1418
FATAN	403	430	422	428	426	1240
FATANH	693	720	712	718	716	1530
FCMP	33	62	54	60	58	870
FCOS	391	418	410	416	414	1228
FCOSH	607	634	626	632	630	1444
FDIV	103	132	124	130	128	940
FETOX	497	524	516	522	520	1334
FETOXM1	545	572	564	570	568	1382
FGETEXP	45	72	64	70	68	882
FGETMAN	31	58	50	56	54	868
FINT	55	82	74	80	78	892
FINTRZ	55	82	74	80	78	892
FLOGN	525	552	544	550	548	1362
FLOGNP1	571	598	590	596	594	1408
FLOG10	581	608	600	606	604	1418
FLOG2	581	608	600	606	604	1418
FMOD	70	99	91	97	95	907
FMOVE to FPn	33	60	52	58	56	870
FMOVE to memory	—	100	80	86	72	2002
FMOVECR	29	—	—	—	—	—
FMUL	71	100	92	98	96	908
FNEG	35	62	54	60	58	872
FREM	100	129	121	127	125	937
FSCALE	41	70	62	68	66	878
FSGLDIV	69	98	90	96	94	906
FSGLMUL	59	88	80	86	84	896
FSIN	391	418	410	416	414	1228
FSINCOS	451	478	470	476	474	1288
FSINH	687	714	706	712	710	1524
FSQRT	107	134	126	132	130	944
FSUB	51	80	72	78	76	888
FTAN	473	500	492	498	496	1310
FTANH	661	688	680	686	684	1498
FTENTOX	567	594	586	592	590	1404
FTST	33	60	52	58	56	870
FTWOTOX	567	594	586	592	590	1404

*Add the appropriate effective address calculation time.
**If the source or destination is an MPU data register, subtract five or two clock cycles, respectively.
***Assumes a static k factor is used if the destination data format is packed decimal. Add 14 clock cycles if a dynamic k factor is used.
****The source operand is from the constant ROM rather than a floating-point data register.

Table 8-3. MC68882 Overall Execution Times

Monadic	Memory Source or Destination Operand Format***																	
	FPn to FPm			Integer****			Single****			Double			Extended			Packed		
	H	T	Total	H	T	Total	H	T	Total	H	T	Total	H	T	Total	H	T	Total
FABS	17	17	38	21	28	68	30	20	51	36	20	57	42	20	63	13	811	893
FACOS	17	607	628	21	618	658	30	610	641	36	610	647	42	610	653	13	1401	1483
FADD	17	35	56	21	54	94	30	38	69	36	38	75	42	38	81	13	827	909
FASIN	17	563	584	21	574	614	30	566	597	36	566	603	42	566	609	13	1357	1439
FATAN	17	385	406	21	396	436	30	388	419	36	388	425	42	388	431	13	1179	1261
FATANH	17	675	696	21	686	726	30	678	709	36	678	715	42	678	721	13	1469	1551
FCMP	17	17	38	21	36	76	30	20	51	36	20	57	42	20	63	13	809	891
FCOS	17	373	394	21	384	424	30	376	407	36	376	413	42	376	419	13	1167	1249
FCOSH	17	589	610	21	600	640	30	592	623	36	592	629	42	592	635	13	1383	1465
FDIV	17	87	108	21	106	146	30	90	121	36	90	127	42	90	133	13	879	961
FETOX	17	479	500	21	490	530	30	482	513	36	482	519	42	482	525	13	1273	1355
FETOXM1	17	527	548	21	538	578	30	530	561	36	530	567	42	530	573	13	1321	1403
FGETEXP	17	27	48	21	38	78	30	30	61	36	30	67	42	30	73	13	821	903
FGETMAN	17	13	34	21	24	64	30	16	47	36	16	53	42	16	59	13	807	889
FINT	17	37	58	21	48	88	30	40	71	36	40	77	42	40	83	13	831	913
FINTRZ	17	37	58	21	48	88	30	40	71	36	40	77	42	40	83	13	831	913
FLOGN	17	507	528	21	518	558	30	510	541	36	510	547	42	510	553	13	1301	1383
FLOGNP1	17	553	574	21	564	604	30	556	587	36	556	593	42	556	599	13	1347	1429
FLOG10	17	563	584	21	574	614	30	566	597	36	566	603	42	566	609	13	1357	1439
FLOG2	17	563	584	21	574	614	30	566	597	36	566	603	42	566	609	13	1357	1439
FMOD	17	54	75	21	73	113	30	57	88	36	57	94	42	57	100	13	846	928
FMOVE to FPn	21	*	21	21	8	48	34	*	34	40	*	40	46	*	46	13	809	891
FMOVE to FPn**	10	0	21	21	8	48	28	6	34	34	6	40	40	6	46	13	809	891
FMOVE to memory*****	—	—	—	0	0	110	38	*	38	44	*	44	50	*	50	0	0	2006
FMOVE to memory**	—	—	—	0	0	110	0	0	38	0	0	44	0	0	50	0	0	2006
FMOVECR******	10	0	32	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
FMUL	17	55	76	21	74	114	30	58	89	36	58	95	42	58	101	13	847	929
FNEG	17	17	38	21	28	68	30	20	51	36	20	57	42	20	63	13	811	893
FREM	17	84	105	21	103	143	30	87	118	36	87	124	42	87	130	13	876	958
FSCALE	17	25	46	21	44	84	30	28	59	36	28	65	42	28	71	13	817	899
FSGLDIV	17	53	74	21	72	112	30	56	87	36	56	93	42	56	99	13	845	927
FSGLMUL	17	43	64	21	62	102	30	46	77	36	46	83	42	46	89	13	835	917
FSIN	17	373	394	21	384	424	30	376	407	36	376	413	42	376	419	13	1167	1249
FSINCOS	17	433	454	21	444	484	30	436	467	36	436	473	42	436	479	13	1227	1309
FSINH	17	669	690	21	680	720	30	672	703	36	672	709	42	672	715	13	1463	1545
FSQRT	17	89	110	21	100	140	30	92	123	36	92	129	42	92	135	13	883	965
FSUB	17	35	56	21	54	94	30	38	69	36	38	75	42	38	81	13	827	909
FTAN	17	455	476	21	466	506	30	458	489	36	458	495	42	458	501	13	1249	1331
FTANH	17	643	664	21	654	694	30	646	677	36	646	683	42	646	689	13	1437	1519
FTENTOX	17	549	570	21	560	600	30	552	583	36	552	589	42	552	595	13	1343	1425
FTST	17	15	36	21	26	66	30	18	49	36	18	55	42	18	61	13	809	891
FTWOTOX	17	549	570	21	560	600	30	552	583	36	552	589	42	552	595	13	1343	1425

*These instruction do not have a tail time. The next instruction's head can be added to determine the effective head time.

**When register conflict occurs, concurrency is decreased.

***Add the effective address time to obtain overall execution time. Add the effective address time to obtain effective head time. (This does not apply to the FMOVE to memory instruction.)

****If the source or destination is an MPU data register, subtract five or two cycles, respectively.

*****Assumes a static k factor is used if the destination data format is packed decimal. Add 14 clock cycles if a dynamic k factor is used.

******The source operand is from the constant ROM rather than a floating-point data register.

Table 8-4. Bus Cycle Activity — Arithmetic Operations

Operation Type	FPm Source	Memory Source or Destination Operand Format*				
		Integer**	Single**	Double	Extended	Packed
FPn Destination	(1/0/0/1/1)	(1/1/0/2/2)	(1/1/0/2/2)	(1/2/0/2/3)	(1/3/0/2/4)	(1/3/0/2/4)
Move to Memory***	—	(1/0/1/4/1)	(1/0/1/4/1)	(1/0/2/5/1)	(1/0/3/6/1)	(1/0/3/6/1)

*Add the appropriate effective address calculation bus cycle activity.

**If the source or destination is an MC68020 data register, subtract (0/1/0/0/0) or (0/0/1/0/0), respectively.

***Includes the read of one null (CA=1, IA=1) primitive when the conversion starts, the evaluate effective address and transfer data primitive when the conversion is complete, and a null (CA=0) primitive after the transfer is complete. The MPU reads additional null (CA=1, IA=1) primitives while waiting for the transfer to start. For an MC68881, if no interrupts occur, the number of additional response CIR read cycles is 5 for integer, 3 for single or double, 1 for extended and ~194 for packed. For an MC68882, the number of additional response read cycles is 5 for integer and ~194 for packed.

Table 8-5. Timing Calculation Example

Instruction	<ea> Time	MC68881 Times	MC68882			
			Times	Adjusted	Effective Head Effective Tail	Actual Overlap
FMUL.D <ea>,FP1	6	98	95			
			H=36	36+6=42		
			T=58	58	58	58
FMOVE.D FP2,<ea>	6	86	44			
			H=44	44+6=50	40+42=92	
			T=*	*		
FADD.D<ea>,FP1	6	78	75			
			H=36	36+6=42		
			T=38	38	38	38
FMOVE.X FP0,PF2	0	33	21			
			H=21	21	21+42=63	
			T=*	*		
FMUL.D <ea>,FP2	6	98	95			
			H=36	36+6=42		
			T=58	58	59	58
FMOVE.D FP1,<ea>	6	86	44			
			H=44	44+6=50	51+42=93	
			T=*	*		
FADD.D <ea>,FP2	6	78	75			
			H=36	36+6=42		
			T=38	38	38	
FMOVE.X FP0,FP1	0	33	21			
			H=21	21	21	21
			T=*	*		
Total	36	557	470			175

overall 881 time: 557+36= 593
overall 882 time: 40+36−175= 331
ratio: 593/331 = 1.80

with the corresponding times using the MC68882. The first column lists the instructions in the set. The second column lists the time required to obtain the operand at the effective address. These numbers are taken from Table 8-1. This time applies to both coprocessors; it is added to the execution time for the instruction. The third column lists the total execution times for each instruction when executing in the MC68881.

The four columns to the right list values that apply to the MC68882. From left to right, the columns contain the following values:

1. The overall execution time for each instruction when executing in the MC68882, and the H and T numbers for each of the instructions. The T values for the fully-concurrent FMOVE instructions are shown as T = *.
2. The adjusted head time, which is the sum of the effective address calculation time and the head time. For the FMOVE to memory instructions (opclass 011), the effective address calculation is not added to the head time. The tail time is not altered.
3. The effective head and effective tail time for each instruction. Where T is shown as T = *, the effective head time is the sum of the FMOVE H time plus the H time of the subsequent instruction. The tail time is not altered.
4. The actual overlap time, which is the lesser of the effective tail and the effective head of the column to the left.

At the bottom of the table, the totals show the overall times. For the MC68881, this time is the sum of the total execution time plus the effective address time. For the MC68882, it is the sum of the total execution time plus the effective address time, less the actual overlap time. The conclusion is that for the instruction sequence shown here, the MC68881 required 1.80 times longer to execute compared to the MC68882.

8.5.1.4 MOVE CONTROL REGISTER AND FMOVEM OPERATIONS. Table 8-6 shows the execution times for the FMOVE FPcr and FMOVEM instructions. The timing for the appropriate effective addressing mode must be added to the numbers in this table to determine the overall instruction execution times.

Table 8-6. Move Control Register and MOVEM Execution Times

Operation*		Best Case	Cache Case	Worst Case
FMOVE	FPcr,Rn	29/6 (0/0/0/3/1)	31/6 (0/0/0/3/1)	34/9 (1/0/0/3/1)
	FPcr,<ea>	31/6 (0/0/1/3/1)	33/6 (0/0/1/3/1)	36/9 (1/0/1/3/1)
	Rn,FPcr	26/6 (0/0/0/2/2)	28/6 (0/0/0/2/2)	31/9 (1/0/0/2/2)
	<ea>,FPcr	31/6 (0/1/0/2/2)	33/6 (0/1/0/2/2)	36/9 (1/1/0/2/2)
	#<data>,FPcr	30/6 (0/0/0/2/2)	30/6 (0/0/0/2/2)	31/9 (2/0/0/2/2)
FMOVEM	FPcr_list,<ea>	25+6n/6 (0/0/n/2+n/1)	27+6n/6 (0/0/n/2+n/1)	30+6n/9 (1/0/n/2+n/1)
	<ea>,FPcr_list	25+6n/6 (0/n/0/2/1+n)	27+6n/6 (0/n/0/2/1+n)	30+6n/9 (1/n/0/2/1+n)
	#<data>, FPcr_list	24+6n/6 (0/0/0/2/1+n)	25+6n/6 (0/0/0/2/1+n)	29+6n/9 (1+n/0/0/2/ 1+n)
FMOVEM	FPdr_list,<ea>	35+25n/6 (0/0/3n/3+3n/1)	37+25n/6 (0/0/3n/3+3n/1)	40+25n/9 (1/0/3n/3+3n/1)
	<ea>,FPdr_list	33+31n/6 (0/3n/0/3/1+3n)	35+31n/6 (0/3n/0/3/1+3n)	38+31n/9 (1/3n/0/3/1+3n)
	Dn,<ea>	49+25n/6 (0/0/3n/4+3n/2)	51+25n/6 (0/0/3n/4+3n/2)	54+25n/9 (1/0/3n/4+3n/2)
	<ea>,Dn	47+31n/6 (0/3n/0/4/2+3n)	49+31n/6 (0/3n/0/4/2+3n)	52+31n/9 (1/3n/0/4/2+3n)

*Add the appropriate effective address calculation time. n is the number of registers transferred. Add two clocks if the coprocessor is an MC68882.

NOTE: FPcr or FPdr indicates any one of the floating-point control or data registers, respectively. FPcr_list or FPdr_list indicates a list of any combination of the floating-point control or data registers, respectively.

8.5.1.5 CONDITIONAL INSTRUCTIONS. Table 8-7 lists the execution times for the FPCP conditional instructions. Each entry in this table, except those for the FScc instruction, is complete and does not required the addition of values from any other table. For the FScc instruction, the only additional factor that must be included is the calculate effective address time for the operand to be modified.

Table 8-7. Conditional Instruction Execution Times

Operation	Comments	Best Case	Cache Case	Worst Case
FBcc.W	Branch Taken Branch Not Taken	18/6 (0 0 0/1/1) 16/6 (0/0 0/1/1)	20 6 (0 0 0 1/1) 18 6 (0 0 0 1/1)	23 6 (2 0 0 1 1) 19 6 (1 0 0 1 1)
FBcc.L	Branch Taken Branch Not Taken	18/6 (0 0 0/1/1) 16/6 (0/0 0/1/1)	20 6 (0 0 0 1/1) 18 6 (0 0 0 1/1)	23 6 (2 0 0 1 1) 21 6 (2 0 0 1 1)
FDBcc	True, Not Taken False, Not Taken False, Taken	18/6 (0 0 0 1/1) 22/6 (0 0 0 1/1) 18 6 (0 0 0 1/1)	20 6 (0 0 0 1 1) 24 6 (0 0 0 1 1) 20 6 (0 0 0 1 1)	24 9 (2 0 0 1 1) 32 9 (4 0 0 1 1) 26 9 (3 0 0 1 1)
FNOP	No Operation	16 6 (0 0 0 1/1)	18 6 (0 0 0 1 1)	19 6 (1 0 0 1 1)
FScc	Dn (An)+ or −(An)* Memory**	16/6 (0 0 0/1/1) 18/6 (0/0/1/1/1) 16/6 (0/0/1/1/1)	18 6 (0 0 0 1/1) 22 6 (0 0 1 1 1) 20 6 (0 0 1/1/1)	21 9 (2 0 0 1 1) 25 9 (2 0 1 1 1) 23 9 (2 0 1 1 1)
FTRAPcc	Trap Taken Trap Not Taken	36/6 (0/1/4/1/1) 16/6 (0/0 0/1/1)	39 6 (0 1/4 1/1) 18 6 (0 0 0 1/1)	47 9 (3 1 4 1 1) 22 9 (2 0 0 1 1)
FTRAPcc.W	Trap Taken Trap Not Taken	38/6 (0/1/4/1/1) 18/6 (0 0 0/1/1)	41 6 (0 1/4 1/1) 20 6 (0 0 0 1/1)	45 9 (3 1 4 1 1) 23 9 (2 0 0 1 1)
FTRAPcc.L	Trap Taken Trap Not Taken	40 6 (0 1/4 1/1) 20 6 (0 0 0 1 1)	43 6 (0 1 4 1 1) 22 6 (0 0 0 1 1)	52 9 (4 1 4 1 1) 27 9 (3 0 0 1 1)

*For condition true; subtract one clock for condition false.
**Add the appropriate effective address calculation time.

Since the conditional instructions are intrinsic to the M68000 Family coprocessor interface (i.e., they are not defined by the FPCP through the use of response primitives), the MPU performs most of the processing associated with these instructions. The only part of the instruction that the FPCP performs is the evaluation of the condition predicate written to the condition CIR. Thus, the execution times shown in Table 8-7 are heavily dependent on the environment in which the main processor executes.

The overlap allowed times listed for these instructions indicate the time at the beginning of the instruction that can overlap with the execution of the previous instruction by the FPCP. No overlap is allowed at the end of the instruction since the FPCP is always idle while the MPU is completing the operation.

8.5.1.6 FSAVE AND FRESTORE INSTRUCTIONS. The time required for a context save or restore operation is shown in Table 8-8. The appropriate calculate effective address times must be added to the values in this table to obtain the total execution time for these operations. For the FSAVE instruction, the FPCP may use the not-ready format code to force the MPU to wait while internal operations are completed in order to reduce the size of the saved state frame or reach a point where a save operation can be performed. The idle time occurs if the FPCP is in the idle phase when the save CIR is written (refer to **6.4.3 FSAVE Protocol** and **6.4.4 FRESTORE Protocol** for definitions of instruction phases). The busy time occurs if the FPCP is in the initial phase or at a save boundary in the middle

phase when the save CIR is written. Times for the MC68882, which stores eight additional long words in the idle and busy state frames, are shown in separate table entries.

Table 8-8. FSAVE and FRESTORE Instruction Execution Times

Coprocessor	Operation	State Frame	Best Case	Cache Case	Worst Case
MC68881	FRESTORE	Null Idle Busy	19/4* (0/1/0/1/1) 55/4* (0/7/0/1/7) 289/4* (0/46/0/1/46)	21/4* (0/1/0/1/1) 57/4* (0/7/0/1/7) 291/4* (0/46/0/1/46)	22/4* (1/1/0/1/1) 58/4* (1/7/0/1/7) 292/4* (1/46/0/1/46)
MC68881	FSAVE	Null Idle Busy	14/1 (0/0/1/1/0) 50/1 (0/0/7/7/0) 284/1 (0/0/46/46/0)	16/1 (0/0/1/1/0) 52/1 (0/0/7/7/0) 286/1 (0/0/46/46/0)	18/1 (1/0/1/1/0) 54/1 (1/0/7/7/0) 288/1 (1/0/46/46/0)
MC68882	FRESTORE	Null Idle Busy	19/4* (0/1/0/1/1) 103/4* (0/15/0/1/15) 337/4* (0/54/0/1/54)	21/4* (0/1/0/1/1) 105/4* (0/15/0/1/15) 339/4* (0/54/0/1/54)	22/4* (1/1/0/1/1) 106/4* (1/15/0/1/15) 340/4* (1/54/0/1/54)
MC68882	FSAVE	Null Idle Busy	14/1 (0/0/1/1/0) 98/1 (0/0/15/15/0) 332/1 (0/0/54/54/0)	16/1 (0/0/1/1/0) 100/1 (0/0/15/15/0) 334/1 (0/0/54/54/0)	18/1 (1/0/1/1/0) 102/1 (1/0/15/15/0) 336/1 (1/0/54/54/0)

*Add the appropriate effective address calculation time. Note that the overlap time available for the FRESTORE instruction is of little use, since this operation destroys the previous context of the FPCP.

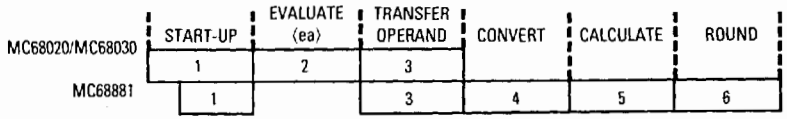
**The second overlap allowed number represents the period during which the FPCP is preparing to perform the save operation and the MPU can process interrupts.

8.5.2 MC68881 Detail Timing Tables

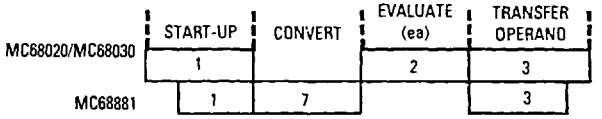
This set of tables provides the information needed to calculate a more precise execution time for an instruction executing in the MC68881, based on the input operand format and type, than can be obtained with the typical timing tables shown previously. Also, these tables contain the information necessary to determine instruction execution timing for a system that does not utilize the MPU as the main processor. The assumptions stated previously are used for these tables, with further restrictions described separately for each table. Note that the timing numbers in the typical timing tables are derived, in most cases, by using the following set of tables.

In order to better understand the relationship of each table in this group, the following diagrams are included. These diagrams break each basic instruction type into separate execution components. For each component, the appropriate tables that are used to calculate the execution time are identified. These diagrams can also be used to clarify the distribution of responsibility for instruction execution between the MPU and the MC68881 and to more clearly illustrate the periods of time during which overlapped execution may occur. In these diagrams, the numbers inside each box indicate the table that is used to determine the timing for that phase of the instruction; the identification key for these tables

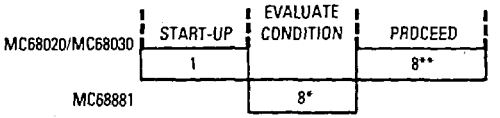
Memory-to-Register, Register-to-Register Operations



Move Register-to-Memory



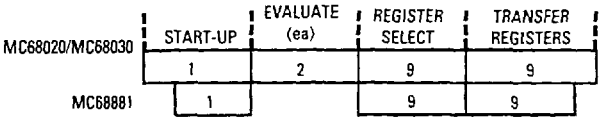
Conditional Operations



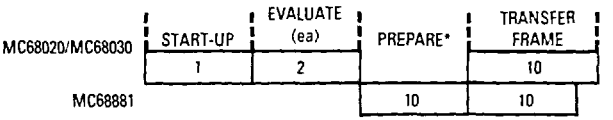
*The timing for the evaluation of the conditional predicate is shown separately in **8.5.2.7 CONDITIONAL TERMINATION**.

**The action taken by the MC68020/MC68030 after the conditional predicate is evaluated by the – xCCw on the instruction (FPcc, FDBcc, FScc, or FTRAPcc).

Move Control or Multiple Registers

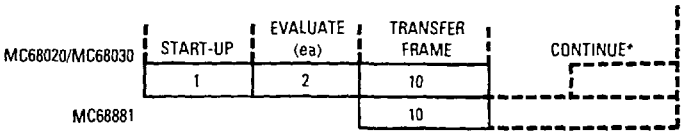


Context Save Operation



*During this period, the MC68881 may force the MC68020/MC68030 to wait while an internal operation is completed, or reaches a point where a save operation can be performed.

Context Restore Operation



*When the context restore operation is completed, the MC68881 continues with any operation that was suspended by a previous context save. The MC68020/MC68030 does not re-establish communications with the MC68881 during the FRESTORE instruction, but the execution of a subsequent RTE instruction restores the MC68020/MC68030 context to the state of the previously suspended operation if necessary.

Table Identification:

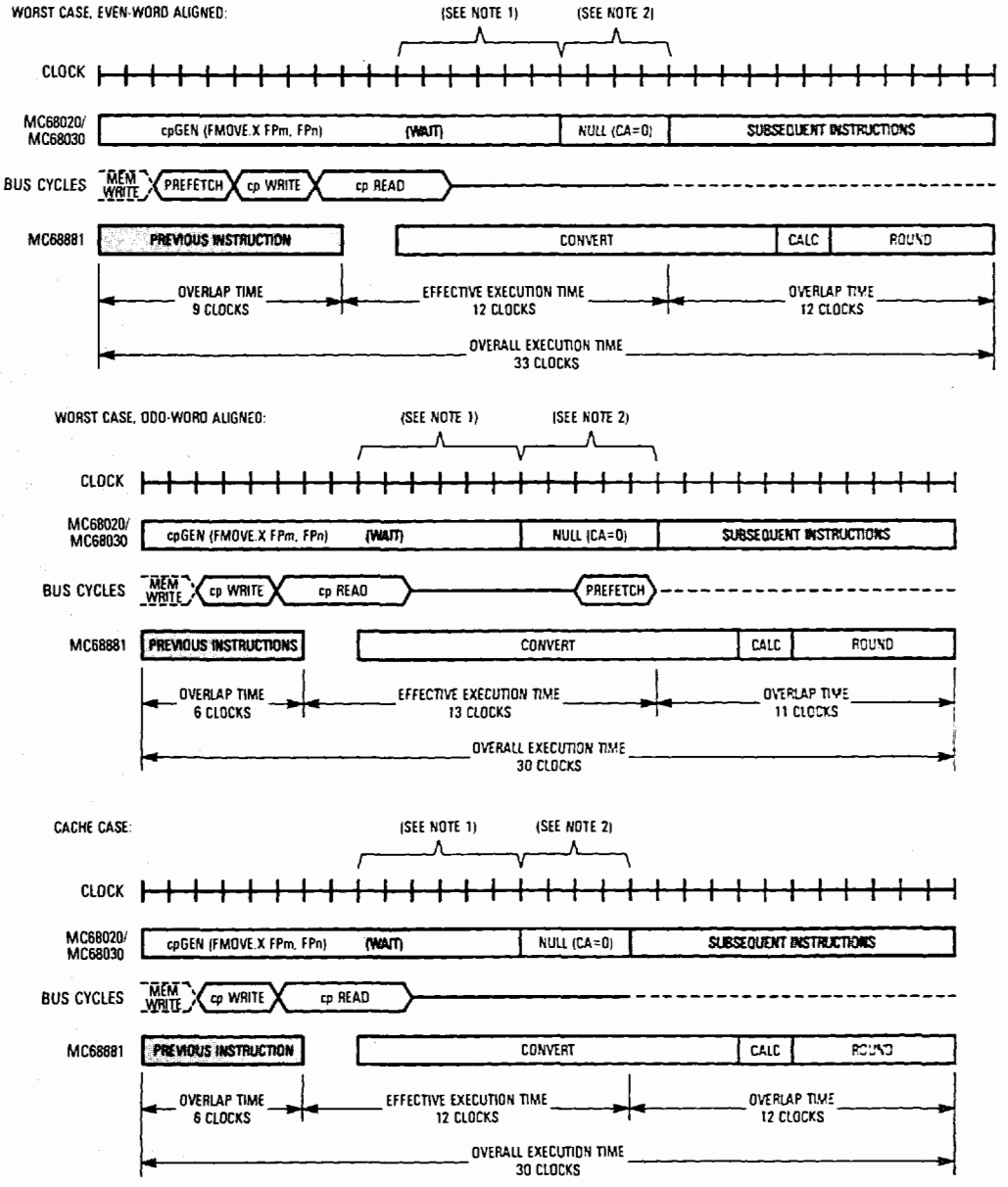
1 — Instruction Start-Up
2 — Effective Address Calculations
3 — Operand Transfers
4 — Input Operand Conversions
5 — Arithmetic Calculations
6 — Rounding and Exception Handling
7 — Output Operand Conversions
8 — Conditional Instructions
9 — Multiple Register Transfers
10 — State Frame Transfers
11 — Exception Processing

As an example of the use of the information in the following paragraphs, consider the FADD.P (A0)+,FP0 instruction. First, the instruction start-up table is used to determine the time required by the MPU to initiate the instruction (by writing the command word and reading the first response primitive). In this case, the first response is evaluate effective address and transfer data (with the PC bit set if any exceptions are enabled). The operand transfer table is then used to determine the time required to transfer the packed decimal string from memory to the MC68881, and this table requires the addition of the effective address calculation time. Thus, the calculate effective address table is used to determine the time required by the MPU to calculate the effective address, (A0)+, and those numbers are added to the start-up and transfer timing numbers. Note that these first three values are almost entirely dependent on the MPU and do not apply if the main processor is not an MC68020 or MC68030.

To complete the timing calculation, a fourth table is used to determine the decimal-to-binary conversion time, based on the input operand data type and value. Finally, the fifth and sixth tables used determine the time required for the addition and rounding operations. The second set of three operations are totally independent of the main processor, and timing numbers derived for them can be utilized by non-MPU based system designers.

As a further aid to understanding the interaction of the MPU with the MC68881 during the execution of an instruction, four diagrams are presented in Figures 8-4 and 8-5. The bus cycle activity and overlapped execution that is allowed during the communications dialog is shown in the diagrams, in addition to illustrations of the effect of instruction alignment, enabled exceptions, and device synchronization. These diagrams represent the clock-cycle-by-clock-cycle activity of the two devices for four cases of the FMOVE instruction. The first three diagrams describe the FMOVE.X FPm,FPn instruction for worst-case and cache-case operation, and the fourth diagram describes the FMOVE.X (An),FPn instruction.

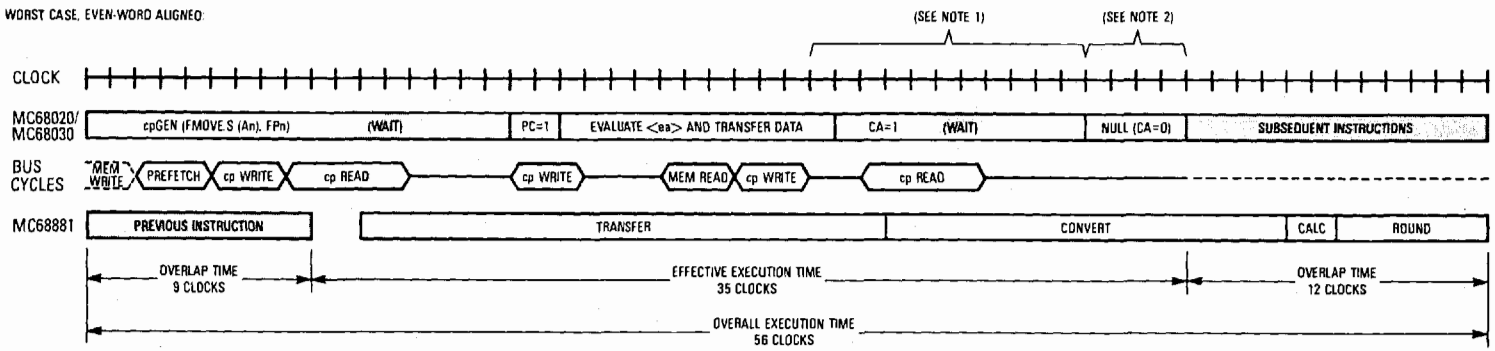
The three diagrams in Figure 8-4 show three cases of the FMOVE.X FPm,FPn instruction. The first and second cases show worst-case operation (where the instruction prefetches required to replace the FMOVE instruction do not hit in the MPU on-chip cache) for the two possible alignments of the instruction. If the first word of the instruction is at an even word address, the prefetch request generated by the cpGEN start-up operation (to replace the F-line operation word) causes an external bus cycle to be executed. This prefetch acquires two words, one of which fills the cpGEN request, and one that is held in a temporary register. The time required to execute this prefetch cycle adds directly to the overall execution time for the instruction, as well as the front-end overlap allowed time. When the null (CA=0) primitive is processed by the MPU, a second prefetch request is generated (to replace the command word which is filled with the word from the temporary register). Thus, the null operation prefetch request does not generate an external bus cycle.



NOTES:

1. These six clocks do not add to the overall execution time for the instruction, only the effective execution time for the MC68020.
2. This operation does not add to the overall execution time for the instruction, only the effective execution time for the MC68020.

Figure 8-4. Instruction Overlap Examples — FMOVE.X FPm,FPn



NOTES:

1. These eleven clocks do not add to the overall execution time for the instruction, only the effective execution time for the MC68020.
2. This operation does not add to the overall execution time for the instruction, only the effective execution time for the MC68020.

Figure 8-5. Instruction Overlap Example — FMOVE.S (An),FPn

When the MPU polls the response CIR, the MC68881 begins execution of the instruction in the fourth clock cycle of the read cycle. As the MC68881 proceeds with the conversion operation, the MPU then completes the cpGEN start-up operation and processes the null (CA=0) primitive. The 10 clock cycles required to perform these operations overlap with the execution of the instruction by the MC68881 and, thus, are not included in the overall execution time calculation (although they are included in the effective execution time calculation). The same consideration applies to the second and third diagrams in Figure 8-4.

As shown in the second diagram of Figure 8-4, if the first word of the instruction is at an odd word address, the prefetch requested by the cpGEN start-up is filled from the temporary register (which was loaded by a prefetch requested during the previous MPU instruction) and an external bus cycle is not required. When the null (CA=0) primitive is processed, a second prefetch request is generated which must be filled by the execution of an external bus cycle. Thus, the start-up operation for this case is a minimum of three clock cycles shorter than the first case (although the overlap allowed time is also shorter) while the time required to process the null primitive is at least one clock cycle longer. Since the null processing overlaps with the execution of the operand conversion by the MC68881, the overall execution time for the instruction is shorter, although the overlap allowed time at the end of the instruction is reduced.

For the third case, both of the instruction prefetch requests generated during the instruction execution are satisfied by either the temporary register or the on-chip instruction cache. Thus, the overall execution time achieves the absolute best case while allowing the maximum possible overlap between the two devices.

The diagram in Figure 8-5 illustrates the execution of the FMOVE.S (An),FPn instruction where the instruction is even-word aligned, the MPU cache is disabled, and at least one of the arithmetic exceptions is enabled. Under these conditions, the cpGEN start-up operation is identical to the first diagram in Figure 8-4, except that the primitive returned by the MC68881 is evaluate effective address and transfer data with the PC and CA bits set. Thus, the first operation performed by the MPU while processing this primitive is to pass the program counter, which adds two clock cycles to both the effective and overall execution times. (Note that the third clock cycle of the coprocessor write cycle overlaps with the effective address calculation.) The MPU then evaluates the effective address, (An), which requires two clock cycles, and transfers the 32-bit single precision operand from memory. The come-again operation is then performed, which requires 10 clock cycles, followed by a four clock period during which the null (CA=0) primitive is processed.

The MC68881 does not start the input conversion operation until the single precision operand is internally passed to the execution unit. The MC68881 bus interface unit requires three clocks from the end of the operand write cycle to transfer the operand to the execution unit; thus, the conversion does not begin until three clock cycles after the end of the write cycle. This three-clock-cycle transfer operation and part of the conversion operation occur simultaneously with the completion of the CA=1 and null processing by the MPU. Thus, 15 clock cycles of the MPU effective execution time do not contribute to the overall execution time for the instruction.

The previous four examples are intended to clarify the meaning of the detailed execution timing tables that follow. The only difference between the FMOVE instruction examples presented and any of the monadic or dyadic instructions is that the convert and calculate times are different. (The round time is also different if an exception occurs.) Also, the

effective address calculation and operand transfer times are different. Notice that the timing prior to the start of the conversion operation is almost entirely dependent on the execution characteristics of the main processor, while the timing for the rest of the instruction is dependent solely on the FPCP. This distinction is useful when the execution timing for a main processor other than the MC68020 or MC68030 is to be determined.

NOTE

The term "not normalized" is used frequently in the following tables. This term is used where conditions allow the input of a denormalized or unnormalized number, and the term "denormalized" is used where only a denormalized input is possible. Refer to **3.2.2 Denormalized Numbers** for a description of the denormalized and unnormalized data types.

8.5.2.1 INSTRUCTION START-UP. When the MPU encounters an FPCP instruction, it decodes the type of the coprocessor instruction and then initiates communications with the FPCP using the appropriate coprocessor interface bus cycle. Table 8-9 lists the execution timing of the MPU for the start-up phase of each of the coprocessor instruction types. For the general instruction type, the start-up time includes the command CIR write and response CIR read cycles that initiate the instruction dialog between the MPU and the FPCP. For the conditional instruction types, the start-up time includes the condition CIR write and response CIR read cycles.

Table 8-9. Instruction Start-Up Times

Instruction Type	Best Case	Cache Case	Worst Case
General*	12/6 (0/0/0/1/1)	14/6 (0/0/0/1/1)	17/9 (1/0/0/1/1)
FBcc	12/6 (0/0/0/1/1)	14/6 (0/0/0/1/1)	14/6 (0/0/0/1/1)
FDBcc, FScc and FTRAPcc	12/6 (0/0/0/1/1)	14/6 (0/0/0/1/1)	17/9 (1/0/0/1/1)
FSAVE**	13/1 (0/0/1/1/0)	15/1 (0/0/1/1/0)	15/1 (0/0/1/1/0)
FRESTORE**	16/4** (0/1/0/1/1)	18/4** (0/1/0/1/1)	18/4** (0/1/0/1/1)

*These execution time numbers represent the overall execution time for this operation with respect to the MPU, and therefore, are used to calculate the effective execution time of the instruction. However, six clock cycles always overlap with the execution of a register-to-register instruction (OPCLASS 000) by the FPCP, and therefore, should not be included in the calculation to generate the overall execution time.

**Add the appropriate effective address calculation time. Note that the overlap time available for the FRESTORE instruction is of little use, since this operation destroys the previous context of the FPCP.

For the FSAVE instruction, the start-up time includes the read of the save CIR and the write of the format word to memory. For the FRESTORE instruction, the start-up time includes the read of the format word from memory, the write of the restore CIR, and the read of the restore CIR to validate the format word. The effective address calculation time is not included for the FSAVE and FRESTORE instructions; the appropriate values must be obtained from the calculate effective address table and added to the start-up values for these instructions.

If an enabled pre-instruction exception is pending when the MPU attempts to initiate an FPCP instruction, the instruction start-up operation is performed for the general or conditional instruction types, and then the MPU proceeds to perform exception processing (at

the request of the FPCP). In this case, the start-up timing numbers are added to the values from the exception processing tables to determine the time required to begin execution of the exception handler.

The MPU terminates all instructions except FSAVE and FRESTORE by processing a null (CA=0) primitive (unless a mid-instruction exception occurs). Therefore, the timing values in Table 8-10 should be included in the calculation of the effective execution time for the MPU, where appropriate.

Table 8-10. Null Primitive Time Values

Primitive Type	Best Case	Cache Case	Worst Case
Null (CA=0) with no tracing	4/4* (0/0/0/0/0)	4/4* (0/0/0/0/0)	5/5* (1/0/0/0/0)

*Overlap is allowed for register-to-register and external-to-register instructions only (OPCLASS 000 and 010).

8.5.2.2 TRANSFER OPERAND. Tables 8-11 and 8-12 show the timing for the transfer of an operand to or from the FPCP by the MPU. Table 8-11 shows the values for external source or destination operands that reside in an MPU register or in memory, and Table 8-12 shows the values for immediate source operands. For input transfers, the timing numbers shown include the time required by the MPU to process the evaluate effective address and transfer data (with CA=1) primitive, and for the FPCP to perform the internal transfer of the operand to the execution unit. For the MPU, the last clock cycle of the transfer operation and the processing for CA=1 always overlaps with the input operand transfer and conversion operations by the FPCP, and therefore, is not added to the overall execution time for the instruction (although these operations are included in the calculation of the effective execution time for the MPU).

8

Table 8-11. Operand Transfer Time — External Operand

Transfer Type	Operand Format				
	Byte	Word	Long, Single	Double	Ext., Packed
From MC68020 Dn	14/0 (0/0/0/1/1)	14/0 (0/0/0/1/1)	14/0 (0/0/0/1/1)	—	—
From Memory*	19/0 (0/1/0/1/1)	19/0 (0/1/0/1/1)	19/0 (0/1/0/1/1)	25/0 (0/2/0/1/2)	31/0 (0/0/3/1/0)
To MC68020 Dn	17/0 (0/0/0/3/0)	17/0 (0/0/0/3/0)	17/0 (0/0/0/3/0)	—	—
To Memory**	19/0 (0/0/1/3/0)	19/0 (0/0/1/3/0)	19/0 (0/0/1/3/0)	25/0 (0/0/2/4/0)	31/0 (0/0/3/5/0)

*Add the appropriate effective address calculation time. Eleven clocks of the MPU processing overlap with execution by the MC68881, which requires five or three clock cycles after the last coprocessor write cycle to complete the internal transfer for double or any other format, respectively. Thus, reduce the numbers above by six clocks for double or eight clocks for any other format for calculation of the overall execution time.

**Add the appropriate effective address calculation time. In the event the destination is packed decimal and a dynamic k factor is used, add 14/0 (0/0/0/1/1).

Table 8-12. Operand Transfer Time — Immediate Operand

Immediate Operand Format	Best Case	Cache Case	Worst Case
Byte, Word	14/0 (0/0/0/1/1)	14/0 (0/0/0/1/1)	17/0 (1/0/0/1/1)
Long, Single	18/0 (0/0/0/1/1)	18/0 (0/0/0/1/1)	19/0 (1/0/0/1/1)
Double	22/0 (0/0/0/1/2)	22/0 (0/0/0/1/2)	24/0 (2/0/0/1/2)
Extended, Packed	26/0 (0/0/0/1/3)	26/0 (0/0/0/1/3)	30/0 (3/0/0/1/3)

For output operand transfers, the timing numbers include the processing for the evaluate effective address and transfer data primitive (with CA = 1). Since no overlap occurs during an output transfer, the values in the table are used directly in the overall execution time calculation. Note that the bus cycle activity numbers include the read of the evaluate effective address and transfer data primitive at the end of the conversion (even though the execution time for the conversion is not included). The read time is included because null (CA = 1, IA = 1) primitives are read during the instruction start-up operation and while waiting for the conversion to complete, and the evaluate effective address and transfer data primitive is read during the processing of one of those primitives.

In order to calculate the effective execution time for the MPU for either input or output transfers, the processing time for the null (CA = 0) primitive that terminates the dialog must be included. For output conversions that cause an enabled exception, the take mid-instruction exception primitive is returned after the operand transfer is complete. In this case, the appropriate exception processing execution time values must be included in lieu of the null (CA = 0) processing time in the calculation of the overall execution time.

8.5.2.3 INPUT OPERAND CONVERSION. All FPCP instructions that require an input operand execute an implied conversion to the 80-bit extended precision format that is used internally. The amount of time required to perform this conversion depends on the format, value, and type of the input operand. Table 8-11 shows the amount of time required to convert an input operand to the internal data format, starting from the end of the internal operand transfer after the last write cycle to the operand CIR.

For dyadic operations, one portion of Table 8-13 for conversions from each combination of source data format and type versus destination data type is included. For monadic operations, one portion includes the conversion timing for any data format and type. Only one number is listed in each entry, since the total number of clock cycles required is equal to the number of overlap allowed clock cycles, and no bus cycles are generated during this stage of an instruction (since the FPCP does not require any further services of the MPU after this stage of an instruction starts).

8.5.2.4 ARITHMETIC CALCULATION. Tables 8-14 and 8-15 show the time required by the MC68881 to perform any of its general-purpose arithmetic operations. One portion of Table 8-14 shows the execution time values for each dyadic instruction with respect to the combination of input operand data types. Table 8-15 shows the execution time values for all of the monadic operations. Each entry in these tables includes the time from the end of the input operand conversion to completion of the calculation. Only one number is shown for each entry, since no bus cycles are generated during this stage of an instruction. Also, the total number of clock cycles required for the calculation is equal to the number of overlap allowed clock cycles, since the FPCP does not require any further services of the MPU after this stage of an instruction starts.

Some entries in these tables refer to a footnote that contains more detailed timing information for an operation (e.g., the table for addition contains an entry that references the ADD footnote, which contains three numbers, based on the input operands). Furthermore, in some cases, an entry refers to another table that contains the execution time required to handle certain input operands. For example, if an entry contains NAN1, refer to the entry of the same name in **8.5.2.10 EXCEPTION PROCESSING.**

Table 8-13. Input Operand Conversion

Dyadic Input Conversions — Source Operand is Byte, Word, or Long:

Destination \ Source	Normalized +	Normalized −	Not Normalized	Zero	Infinity	NAN
Normalized	24	26	—	22	—	—
Unnormalized	36	38	—	34	—	—
Zero	30	32	—	28	—	—
Infinity	28	30	—	26	—	—
NAN	30	32	—	28	—	—

Dyadic Input Conversions — Source Operand is Single Precision:

Destination \ Source	Normalized	Not Normalized	Zero	Infinity	NAN
Normalized	18	36	22	24	26
Unnormalized	30	48	34	36	38
Zero	24	42	28	30	32
Infinity	22	40	26	28	30
NAN	24	42	28	30	32

Dyadic Input Conversions — Source Operand is Double Precision:

Destination \ Source	Normalized	Not Normalized	Zero	Infinity	NAN
Normalized	16	34	20	22	24
Unnormalized	28	46	32	34	36
Zero	22	40	26	28	30
Infinity	20	38	24	26	28
NAN	22	40	26	28	30

Dyadic Input Conversions — Source Operand is Extended Precision:

Destination \ Source	Normalized	Not Normalized	Zero	Infinity	NAN
Normalized	10	26	12	12	14
Unnormalized	22	38	24	24	26
Zero	16	32	18	18	20
Infinity	14	30	16	16	18
NAN	16	32	18	18	20

Monadic or Dyadic Input Conversions — Source Operand is Packed Decimal:

Destination \ Source	Normalized	Not Normalized	Zero	Infinity	NAN
Normalized	~822	~822	22	22	24
Unnormalized	~848	~848	34	34	36
Zero	~842	~842	28	28	30
Infinity	~840	~840	26	26	28
NAN	~842	~842	28	28	30

~Indicates a typical conversion time. The minimum maximum conversion time is 954 clock cycles.

Monadic or Dyadic Input Conversions — Source Operand is FPm:

Destination \ Source	Normalized	Not Normalized	Zero	Infinity	NAN
Normalized	14	30	16	16	18
Unnormalized	26	42	28	28	30
Zero	20	36	22	22	24
Infinity	18	34	20	20	22
NAN	20	36	22	22	24

Monadic Input Conversions — Source Operand is in Memory:

Format \ Type	Normalized +	Normalized −	Not Normalized	Zero	Infinity	NAN
Byte, Word, Long	22	24	—	20	—	—
Single	16		30	20	24	24
Double	14		28	18	22	22
Extended	8		20	10	12	12

Table 8-14. Arithmetic Calculation Times — Dyadic Operations

FADD Calculation Time:

Destination \ Source		Normalized	Zero	Infinity	NAN
		+ –	+ –	+ –	+ –
Normalized	+	ADD	2+	6	NAN2
	–				
Zero	+	2+	6 26	6	NAN2
	–		26 6		
Infinity	+	6	6	6 20	NAN2
	–			20 6	
NAN	+	NAN1	NAN2	NAN2	NAN3
	–				

ADD:
- 24+ if the source and estination exponents are equal, and the source mantissa is less than the destination mantissa.
- 26+ if the source and destination exponents are equal, and the source mantissa is greater than or equal to the destination mantissa.
- 28+ if the source and destination exponents are not equal.

FCMP Calculation Time:

Destination \ Source		Normalized	Zero	Infinity	NAN
		+ –	+ –	+ –	+ –
Normalized	+	CMP 6	6	8 6	NAN4
	–	6 CMP		6 8	
Zero	+	8 6	6	8 6	NAN4
	–	6 8		6 8	
Infinity	+	6	6	6	NAN4
	–				
NAN	+	NAN1	NAN2	NAN2	NAN3
	–				

CMP:
- 8 if thesource exponent is greater than the desination exponent.
- 10 if the source exponent is less than or equal to the destination exponent.

FDIV Calculation Time:

Destination \ Source		Normalized	Zero	Infinity	NAN
		+ –	+ –	+ –	+ –
Normalized	+	DIV	20	6 8	NAN2
	–			8 6	
Zero	+	6 8	20	6 8	NAN2
	–				
Infinity	+	6 8	6 8	20	NAN2
	–				
NAN	+	NAN1	NAN2	NAN2	NAN3
	–				

DIV:
- 78+ if the intermediate result is normalized.
- 80+ if the intermediate result is denormalized.

FMOD Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	MOD		20		6+		NAN2	
Normalized −	MOD		20		6+		NAN2	
Zero +	6+		20		6+		NAN2	
Zero −	6+		20		6+		NAN2	
Infinity +	IOP		20		20		NAN2	
Infinity −	IOP		20		20		NAN2	
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −	NAN1		NAN2		NAN2		NAN3	

MOD: 18+ if the quotient is zero; else:
(40+70 (INT (1+destination exponent−source exponent))) +
((( )))

FMUL Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	MUL		6 8		6 8		NAN2	
Normalized −	MUL		6 8		6 8		NAN2	
Zero +	6	8	6	8	20		NAN2	
Zero −	6	8	6	8	20		NAN2	
Infinity +	6	8	20		6	8	NAN2	
Infinity −	6	8	20		6	8	NAN2	
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −	NAN1		NAN2		NAN2		NAN3	

MUL: 46+ if the intermediate result is normalized.
48+ if the intermediate result is not normalized.

FREM Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	REM		20		6+		NAN2	
Normalized −	REM		20		6+		NAN2	
Zero +	6+		20		6+		NAN2	
Zero −	6+		20		6+		NAN2	
Infinity +	IOP		20		20		NAN2	
Infinity −	IOP		20		20		NAN2	
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −	NAN1		NAN2		NAN2		NAN3	

REM: 18+ if the quotient is zero; else:
(40+70 (INT (1+destination exponent−source exponent))) +
(((64)))

FSCALE Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	SCALE		6+		20		NAN2	
Normalized −	SCALE		6+		20		NAN2	
Zero +	6		6		20		NAN2	
Zero −	6		6		20		NAN2	
Infinity +	6		6		20		NAN2	
Infinity −	6		6		20		NAN2	
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −	NAN1		NAN2		NAN2		NAN3	

SCALE:
- 12+ if the source exponent (unbiased) is less than zero.
- 16+ if the source exponent (unbiased) is in the range [0 . . . 15].
- 20+ if the source exponent (unbiased) is greater than 15.

FSGLDIV Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	SGLDIV		20		6	8	NAN2	
Normalized −	SGLDIV		20		8	6	NAN2	
Zero +	6	8	20		6	8	NAN2	
Zero −	6	8	20		6	8	NAN2	
Infinity +	6	8	6	8	20		NAN2	
Infinity −	6	8	6	8	20		NAN2	
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −	NAN1		NAN2		NAN2		NAN3	

SGLDIV:
- 44 if no extendedprecision underflow or overflow occurs.
- 62 if an extended precision overflow occurs.
- 90 if an extended precision underflow occurs.

FSGLMUL Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	SGLMUL		6		6		NAN2	
Normalized −	SGLMUL		8		8		NAN2	
Zero +	6	8	6	8	20		NAN2	
Zero −	6	8	6	8	20		NAN2	
Infinity +	6	8	20		6	8	NAN2	
Infinity −	6	8	20		6	8	NAN2	
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −	NAN1		NAN2		NAN2		NAN3	

SGLMUL:
- 34 if no extended precision underflow or overflow occurs.
- 52 if an extended precision overflow occurs.
- 80 if an extended precision underflow occurs.

FSUB Calculation Time:

Destination \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
Normalized +	SUB		2+		8		NAN2	
Normalized −								
Zero +	4+		26 8	8 26	8		NAN2	
Zero −								
Infinity +	6		6		20 8	8 20	NAN2	
Infinity −								
NAN +	NAN1		NAN2		NAN2		NAN3	
NAN −								

SUB:
- 24+ if the source and destination exponents are equal, and the source mantissa is less than the destination mantissa.
- 26+ if the source and destination exponents are equal, and the source mantissa is greater than or equal to the destination mantissa.
- 28+ if the source and destination exponents are not equal.

If an entry in these tables is appended with a plus sign (+), the appropriate timing numbers from the rounding and exception handling table (in **8.5.2.10 EXCEPTION PROCESSING**) must be used to calculate the overall execution time for an instruction.

Otherwise, the numbers from these tables include the time to handle exceptional operand cases and produce the final result.

8.5.2.5 OUTPUT OPERAND CONVERSION. The FMOVE.<fmt> FPn,<ea> instruction performs an implicit conversion from the 80-bit extended precision format used internally by the FPCP to an external data format. Table 8-16 lists the conversion times for most output operations. Since the execution timing for conversions from the internal extended precision format to either single or double precision is highly data dependent, the timing for these operations (for in-range, nonzero input values) is listed in a second table, Table 8-17.

The amount of time required to perform this conversion depends on the value and type of the input operand and the format of the desired output. The values given in the following tables, in FPCP clock cycles, include the time from the fourth clock cycle of the first response CIR read (which returns a null (CA=1, IA=1) primitive) to completion of the conversion (when a read of the response CIR returns an evaluate effective address and transfer operand primitive). Only one number is shown for each entry, since no bus cycles are generated during this stage of an instruction. Also, the total number of clock cycles required for operand conversion is equal to the number of overlap allowed clock cycles (during which time interrupts may be handled; normal program execution is not allowed), since the FPCP does not require any services of the MPU during this stage of an instruction.

8.5.2.6 ROUNDING AND EXCEPTION HANDLING. Tables 8-18 and 8-19 contain the execution times for rounding and for various exception handling operations. For the typical execution time tables shown previously, it is assumed that the MC68881 uses the default operating mode of round-to-extended precision, and no overflow or underflow exceptions occur. If this is not the case, the round/store phase of most arithmetic instructions takes longer to execute. The entries in the typical execution time tables include the processing time for no underflow, overflow, or round overflow as indicated in Table 8-18.

Table 8-15. Arithmetic Calculation Times — Monadic Operations

Operation \ Source	Normalized +	Normalized −	Zero +	Zero −	Infinity +	Infinity −	NAN +	NAN −
FABS	4+		4+		8		NAN2	
FACOS	594+		12[1] 20[2]		20		NAN2	
FASIN	550+		6		20		NAN2	
FATAN	372+		6		12[1] 20[2]	14[1] 22[2]	NAN2	
FATANH	662+		6		20		NAN2	
FCOS	360+3		8		20		NAN2	
FCOSH	576+		8		8		NAN2	
FETOX	466+		8		6		NAN2	
FETOXM1	514+		6		6	8	NAN2	
FGETEXP	exponent=0:16 exponent>0:20 exponent<0:22		6		20		NAN2	
FGETMAN	6		6		20		NAN2	
FINT, FINTRZ	fraction=0:8 fraction≠0:30 result=0:28		6		60		NAN2	
FLOGN	494+ IOP		22		6	20	NAN2	
FLOGNP1	540+[4]		6		6	20	NAN2	
FLOG10	550+ IOP		22		6	20	NAN2	
FLOG2	550+ IOP		22		6	20	NAN2	
FMOVE to FPn	2+		6		6		NAN2	
FMOVECR	18[1] 26[2]		—		—		—	
FNEG	4+		4+		8		NAN2	
FSIN	360+[3]		6		20		NAN2	
FSINCOS	420+[3]		20		26		NAN6	
FSINH	656+		6		6		NAN2	
FSQRT	76+ IOP		6		6	20	NAN2	
FTAN	442+[3]		6		20		NAN2	
FTANH	630+		6		8		NAN2	
FTENTOX	536+		8		6		NAN2	
FTST	8		8		8		NAN5	
FTWOTOX	536+		8		6		NAN2	

NOTES:
1. If the extended precision rounding mode is used.
2. If the single or double precision rounding mode is used.
3. This assumes that the source operand is in the range (−9... +9). If the source operand is outside of that range, the appropriate REM calculation time required to perform the argument reduction must be added to this value.
4. If the source operand is less than or equal to −1, use the IOP time.

Table 8-16. Output Operand Conversion

Dest. Format / Source Type	Normalized +	Normalized −	Not Normalized +	Not Normalized −	Zero	Infinity	NAN
Integer, No Overflow	50	52	60	62	18	—	24
Integer, Overflow	52	56	62	66	18	24 26	24
Single	(see below)		(see below)		16	18	NAN7
Double	(see below)		(see below)		16	18	NAN7
Extended	18		(see note 1)		16	16	NAN7
Packed	(see note 2)		(see note 2)		24	24	NAN2

NOTES:
1. 26 clocks if the source operand is an unnormalized number. 56 clocks if the source operand is a denormalized number.
2. 1942 clocks is the typical time required for the conversion, if no overflow occurs. The maximum time is 3674 clocks.

Table 8-17. Output Operand Conversion — Binary Real Formats

Conversion Result / Source Type	Normalized	Not Normalized
No Underflow, Overflow or Round Overflow	38	48
No Underflow or Overflow; Round Overflow	42	52
Overflow; RN or RZ Mode; No Round Overflow	44	54
Overflow; RN or RZ Mode; Round Overflow	48	58
Overflow; RM or RP Mode; No Round Overflow	46	56
Overflow; RM or RP Mode; Round Overflow	50	60
Underflow; No Round Overflow	66	76
Underflow; Round Overflow	70	80

Table 8-18. Rounding Operation Time Values

Rounding Precision	Result	Clock Cycles
Extended	No Underflow, Overflow, or Round Overflow	6
	No Underflow or Overflow; Round Overflow	6
	Underflow	34
	Overflow; RN or RZ Mode; No Round Overflow	14
	Overflow; RM or RP Mode; No Round Overflow	16
	Round Overflow (Not Caused By Rounding); RN or RZ Mode	16
	Round Overflow (Not Caused By Rounding); RM or RP Mode	18
	Round Overflow (Caused By Rounding); RN or RZ Mode	20
	Round Overflow (Caused By Rounding); RM or RP Mode	22
Single or Double	Result is Zero	6
	No Underflow, Overflow, or Round Overflow	24
	No Underflow or Overflow; Round Overflow	28
	Underflow; No Round Overflow	56
	Underflow; Round Overflow	60
	Overflow; RN or RZ Mode; No Round Overflow	30
	Overflow; RM or RP Mode; No Round Overflow	32
	Overflow; RN or RZ Mode; Round Overflow	34
	Overflow; RM or RP Mode; Round Overflow	36

Table 8-18 indicates the number of clock cycles that should be added in the calculation of the execution time for an arithmetic instruction (both the total and the overlap allowed numbers) to account for the various rounding precision and exception handling combinations. The entries in the table include the time from the end of the calculation phase to completion of the FPCP instruction execution (i.e., when the PF bit in the null (CA=0) primitive is clear if the response CIR is read).

When an FMOVE instruction that moves data between FP registers is executed in the MC68882 and the FPCR mode control byte specifies single or double precision rounding, no instruction execution concurrency is allowed. Similarly, an FMOVE instruction that moves data to a single or double precision memory location executes without overlap in the MC68882.

Table 8-19 includes the entries referenced previously in the arithmetic calculation and output operand conversion tables for exceptional operand inputs. The values in this table are used for the calculation or conversion timing in lieu of a value from the appropriate table. For example, if an output operand conversion table entry references NAN7, then the timing number from the NAN7 entry in Table 8-19 is used as the conversion time value.

Table 8-19. Exception Handling Time Values

Exception Identifier	Conditions	Clock Cycles
IOP	Source Operand is Not Denormalized Source Operand is Denormalized	20 32
NAN1	Destination is a QNAN, Source is not Denormalized Destination is a QNAN, Source is Denormalized Destination is an SNAN, Source is not Denormalized Destination is an SNAN, Source is Denormalized	28 52 30 54
NAN2	The NAN is a QNAN The NAN is an SNAN	28 30
NAN3	Both NANs are QNANs Source is a QNAN, Destination is an SNAN Source is an SNAN, Destination is a QNAN Both NANs are SNANs	28 30 32 30
NAN4	The NAN is a QNAN The NAN is an SNAN	30 32
NAN5	The NAN is a QNAN The NAN is an SNAN	8 10
NAN6	The NAN is a QNAN The NAN is an SNAN	38 40
NAN7	The NAN is a QNAN The NAN is an SNAN	22 24

8.5.2.7 CONDITIONAL TERMINATION. The effective execution time for the conditional and context switch instructions is not heavily dependent on the FPCP, since the execution of these operations is performed, for the most part, by the MPU. In order to calculate the effective execution time for these instructions, Table 8-20 shows the termination timing for the MPU. The termination processing starts four MPU clock cycles after the end of the response CIR read and ends when the MPU begins execution of the next instruction. Note that the allowed overlap time in this table is always zero, since the FPCP is in the idle state when these instructions reach the termination phase. However, if multiple coprocessors

Table 8-20. Conditional Termination Times Values

Instruction Type		Best Case	Cache Case	Worst Case
FBcc.W	Branch Taken	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	9/0 (2/0/0/0/0)
	Branch Not Taken	4/0 (0/0/0/0/0)	4/0 (0/0/0/0/0)	5/0 (1/0/0/0/0)
FBcc.L	Branch Taken	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	9/0 (2/0/0/0/0)
	Branch Not Taken	4/0 (0/0/0/0/0)	4/0 (0/0/0/0/0)	7/0 (2/0/0/0/0)
FDBcc	True, Not Taken	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	7/0 (1/0/0/0/0)
	False, Not Taken	10/0 (0/0/0/0/0)	10/0 (0/0/0/0/0)	15/0 (3/0/0/0/0)
	False, Taken	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	9/0 (2/0/0/0/0)
FScc	Dn	4/0 (0/0/0/0/0)	4/0 (0/0/0/0/0)	4/0 (1/0/0/0/0)
	(An)+ or −(An)*	6/0 (0/0/1/0/0)	8/0 (0/0/1/0/0)	8/0 (1/0/1/0/0)
	Memory**	4/0 (0/0/1/0/0)	6/0 (0/0/1/0/0/)	6/0 (1/0/1/0/0)
FTRAPcc	Trap Taken	24/0 (0/1/4/0/0)	25/0 (0/1/4/0/0)	30/0 (2/1/4/0/0)
	Trap Not Taken	4/0 (0/0/0/0/0)	4/0 (0/0/0/0/0)	5/0 (1/0/0/0/0)
FTRAPcc.W	Trap Taken	26/0 (0/1/4/0/0)	27/0 (0/1/4/0/0)	28/0 (2/1/4/0/0)
	Trap Not Taken	6/0 (0/0/0/0/0)	6/0 (0/0/0/0/0)	6/0 (1/0/0/0/0)
FTRAPcc.L	Trap Taken	28/0 (0/1/4/0/0)	29/0 (0/1/4/0/0)	35/0 (3/1/4/0/0)
	Trap Not Taken	8/0 (0/0/0/0/0)	8/0 (0/0/0/0/0)	10/0 (2/0/0/0/0)

*For condition true; subtract one clock for condition false.
**Add the appropriate effective address calculation time.

are used in a system, the execution of other coprocessors may overlap with the execution of these instructions.

In order to determine the execution time for a conditional operation performed by a processor other than an MPU, it is necessary to know the timing for the conditional evaluation by the FPCP. This value is shown in Table 8-20 (in FPCP clock cycles) and indicates the best-case time from the start of the condition CIR write to the end of the response CIR read (which are the only two coprocessor accesses required).

8.5.2.8 MULTIPLE REGISTER TRANSFER. Table 8-21 lists the number of clock cycles and bus cycles required for the MPU to perform a multiple register transfer to or from the FPCP. These transfers occur during the FMOVEM instruction for either the floating-point control register or floating-point data register form of the instruction. The timing values shown in the table include the processing time for either the evaluate effective address and transfer data or transfer multiple coprocessor registers primitive (for the control or data register form, respectively) with CA=1. Assuming that the main processor is an MC68020 or MC68030, the time required to process the null (CA=0) primitive after the transfer is complete must be included.

For the transfer of multiple control registers, the register select list is included in the instruction, and all of the selected registers are transferred as a single operand (from the perspective of the main processor). For the transfer of multiple data registers, the MPU must read the register select mask before starting the register transfer. The amount of time required by the MPU to read and process the register mask is included in the Table 8-21 entries. If a dynamic register list is used, the time required by the MPU to process the transfer single main processor register primitive must be included and is shown at the top of the table.

Table 8-21. Multiple Register Transfer Time Values

Transfer Type		Timing
Move Single Control Register	To an MC68020 Register	17/0 (0/0/0/2/0)
	To Memory*	19/0 (0/0/1/2/0)
	From an MC68020 Register	14/0 (0/0/0/1/1)
	From Memory	19/0 (0/1/0/1/1)
	#(data)	19/0 (1/0/0/1/1)*
Move Multiple Control Registers	To Memory	13+6n/0 (0/0/n/1+n/0)
	From Memory	13+6n/0 (0/n/0/1/n)
	#(data)	12+6n/0 (n/0/0/1/n)*
Move Multiple Data Registers	To Memory	23+25n/0 (0/0/3n/2+3n/0)
	From Memory	21+23n/0 (0/3n/0/2/3n)

n – is the number of registers transferred.
*If the immediate operand resides in the MPU cache, the number of clock cycles is reduced by 3n and the number of instruction prefetch bus cycles is zero.

8.5.2.9 STATE FRAME TRANSFER. Table 8-22 lists the number of clock cycles and bus cycles required for the MPU to transfer an internal state frame to or from the MC68881. These transfers occur during the FSAVE and FRESTORE instructions. The timing values shown in the table include the time from the end of the instruction start-up operation to the end of the last operand write cycle, assuming that the main processor is an MC68020 or MC68030.

Table 8-22. State Frame Transfer Time Values

Operation	Frame Type	Timing
State Save	Idle	36/0 (0/0/6/6/0)
	Busy	270/0 (0/0/45/45/0)
State Restore	Idle	36/0 (0/6/0/0/6)
	Busy	270/0 (0/45/0/0/45)

Before the transfer of a state frame to the FPCP during an FRESTORE instruction, the MPU must read the format word from memory, write it to the restore CIR, and verify that it is valid by reading the restore CIR. Likewise, during an FSAVE instruction, the MPU must read the format word from the save CIR and store it in memory. The instruction start-up timing table entries include these operations for MC68020/MC68030-based systems.

During an FSAVE operation, the FPCP may require the main processor to wait until the current instruction is completed or a save boundary is reached before starting the state frame transfer. The maximum time that the main processor can be forced to wait is shown at the top of Table 8-22, and should be included in the calculation of the worst-case FSAVE execution time.

In order to calculate overall execution time for the MPU during a FSAVE or FRESTORE instruction, the instruction termination processing time must be included. Table 8-23 lists the timing values for this processing, which is from the end of the last operand write cycle to the beginning of the execution of the next instruction by the MPU.

Table 8-23. Instruction Termination Processing Time Values

Instruction Type	Best Case	Cache Case	Worst Case
FSAVE	1/0 (0/0/0/0/0)	1/0 (0/0/0/0/0)	3/0 (1/0/0/0/0)
FRESTORE	3/0 (0/0/0/0/0)	3/0 (0/0/0/0/0)	4/0 (1/0/0/0/0)

8.5.2.10 EXCEPTION PROCESSING. Table 8-24 indicates the time required for exception processing related to the execution of FPCP instructions. The values in the table for the second and third entries indicate the time from the start of processing the take exception primitive until the MPU resumes normal instruction execution in the appropriate exception handler.

Table 8-24. Exception Processing Time Values

Operation	Best Case	Cache Case	Worst Case
Pass Program Counter	2/2* (0/0/0/0/1)	3/3* (0/0/0/0/1)	3/3* (0/0/0/0/1)
Take Pre-Instruction Exception	22/0 (0/1/4/0/0)	22/0 (0/1/4/0/0)	24/0 (2/1/4/0/0)
Take Mid-Instruction Exception	32/0 (0/1/7/0/0)	32/0 (0/1/7/0/0)	38/0 (2/1/7/0/0)
Process Pre-Instruction Interrupt (I stack)	26/26 (0/2/4/0/0)	26/26 (0/2/4/0/0)	33/33 (2/2/4/0/0)
Process Pre-Instruction Interrupt (M stack)	41/41 (0/2/8/0/0)	41/41 (0/2/8/0/0)	48/48 (2/2/8/0/0)
Process Mid-Instruction Interrupt (I stack)	35/35 (0/2/6/0/0)	36/36 (0/2/6/0/0)	42/42 (2/2/6/0/0)
Process Mid-Instruction Interrupt (M stack)	46/46 (0/2/9/0/0)	47/47 (0/2/9/0/0)	53/53 (2/2/9/0/0)
Process FSAVE Interrupt (I stack)	26/26 (0/2/4/0/0)	26/26 (0/2/4/0/0)	33/33 (2/2/4/0/0)
Process FSAVE Interrupt (M stack)	41/41 (0/2/8/0/0)	41/41 (0/2/8/0/0)	48/48 (2/2/8/0/0)
Format Error, FRESTORE Instruction	23/0 (0/1/4/0/0)	24/0 (0/1/4/0/0)	29/0 (2/1/4/0/0)
RTE, Pre-Instruction Frame	20/20 (0/4/0/0/0)	21/21 (0/4/0/0/0)	24/24 (2/4/0/0/0)
RTE, Mid-Instruction Frame	31/24 (0/6/0/1/0)	32/25 (0/6/0/1/0)	33/26 (1/6/0/1/0)
RTE, Throwaway Frame	15/15 (0/4/0/0/0)	16/16 (0/4/0/0/0)	19/19 (0/4/0/0/0)

*Overlap is allowed only for floating-point register-to-register and register-to-external operations.

To determine the overall exception latency for a pre-instruction exception, the instruction start-up time (for the arithmetic or conditional instruction that is pre-empted by the exception) is added to the exception processing time from Table 8-24. The exception processing time for a take mid-instruction exception primitive is added to the overall execution time for the FMOVE to memory instruction that caused the exception. For conditional instructions that cause a BSUN exception, the pass program counter time shown in the table is also added to the instruction start-up and exception processing time to calculate the overall exception latency for the instruction.

For the take interrupt operations, the values in Table 8-24 include the time from the end of the processing of a response primitive that allows interrupts to the resumption of normal MPU instruction execution in the interrupt handler. (The possible responses are the null (CA=1, IA=1) and null (CA=0, IA=1, PF=0) primitives, or the not ready format code.) If an interrupt is processed during an FMOVE to memory instruction or when the main processor is in the trace mode and receives a null (CA=0, IA=1, PF=0) primitive, a mid-instruction stack frame is used. A pre-instruction stack frame is used for interrupts processed during an FSAVE instruction. The M-stack and I-stack designation indicates whether the M bit of the MPU status register was set or clear, respectively, before the interrupt occurred.

The processing time for an FRESTORE format error includes the time from the end of the FRESTORE start-up operation to when the MPU resumes normal instruction execution in the format error exception handler. Since the characteristics of an FSAVE format error exception are not predictable (and since such an occurrence is catastrophic), execution timing required to handle the error is not included in the table.

The entries in the table for the return from exception (RTE) instruction include the time from the beginning of the execution of the RTE by the MPU to the resumption of the previously aborted operation. If the RTE instruction processes a pre-instruction frame, the time in the table includes the time required to restore the processor context and prepare to execute the instruction at the address in the stack frame program counter image. For the mid-instruction frame, the time in the table includes the time required to restore the processor context and read the response CIR to continue the previously suspended operation. The "RTE, throwaway frame" entries include the time required to read and process the throwaway stack frame (normally from the top of the interrupt stack) and then perform RTE processing for the stack frame on top of the resulting active stack (normally either the master or user stack). Thus, if the MPU must return from an interrupt that occurred while the M bit in the MPU status register was set, a throwaway frame is first processed from the interrupt stack, followed by the processing of the appropriate frame from the master stack (which returns the processor to the context saved by the interrupt processing). For such a case, the "RTE, throwaway frame" times are added to the RTE execution times for the second stack frame to derive the overall execution times for the operation.

In addition to the occurrence of an exception, whether exceptions are enabled or not, can also affect instruction execution time. This is because the FPCP requests the transfer of the program counter at the start of any arithmetic instruction if any exception (other than the BSUN exception) is enabled. If the source operand resides in a floating-point data register, the transfer of the PC does not affect overall execution timing, since it takes place concurrently with the execution of the operation by the FPCP. However, for source operands external to the FPCP, the MPU first passes the PC, and then passes the operand; thus, execution time is affected for this case.

8.6 MAIN PROCESSOR INSTRUCTION OVERLAP TIMING

The MPU overlap allowed table for the MC68881 applies to overlap between MPU and floating-point instructions. Table 8-25 lists the overlap time allowed by the MC68881. The MPU overlap time allowed by the MC68882 is shown in the T columns of Table 8-3.

Table 8-25. Overlap Allowed Times — Arithmetic Operations

Operation Type	FPm Source	Memory Source or Destination Operand Format				
		Integer	Single	Double	Extended	Packed
FPn Destination*	–3	–22	–22	–28	–34	–34
Move to Dn or Memory**	—	41	29	29	9	113

*Subtract these numbers from the overall execution time value in the previous table to determine the allowed overlap time for a particular instruction.

**These numbers represent the amount of time in the middle of the instruction during which the MPU can process interrupts.

~ Indicates a typical time for the binary-to-decimal conversion.

SECTION 9
FUNCTIONAL SIGNAL DESCRIPTIONS

This section contains a brief description of the input and output signals for the MC68881/MC68882 (FPCP) floating-point coprocessor. The signals are functionally organized into groups as shown in Figure 9-1.

NOTE

The terms **assertion** and **negation** are used extensively to avoid confusion when describing "active-low" and "active-high" signals. The term assert or assertion is used to indicate that a signal is active or true, regardless of whether that level is represented by a high or low voltage. The term negate or negation is used to indicate that a signal is inactive or false.

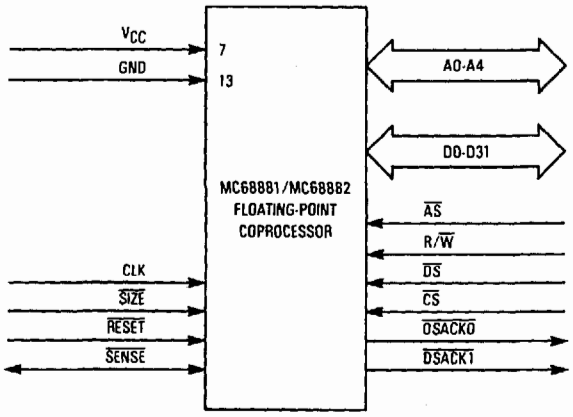


Figure 9-1. MC68881/MC68882 Input/Output Signals

9.1 ADDRESS BUS (A4–A0)

These active-high address line inputs are used by the main processor to select the coprocessor interface register locations located in the CPU address space. These lines control the register selection as listed in Table 9-1.

When the FPCP operates with an 8-bit data bus, the A0 pin is used as an address signal for byte accesses of the coprocessor interface registers. When the FPCP operates with a 16- or 32-bit system data bus, both the A0 and SIZE pins are strapped high and/or low as listed in Table 9-2.

Table 9-1. Coprocessor Interface Register Selection

A4-A0	Offset	Width	Type	Register
0000x	$00	16	Read	Response
0001x	$02	16	Write	Control
0010x	$04	16	Read	Save
0011x	$06	16	Read Write	Restore
0100x	$08	16	—	(Reserved)
0101x	$0A	16	Write	Command
0110x	$0C	16	—	(Reserved)
0111x	$0E	16	Write	Condition
100xx	$10	32	Read/Write	Operand
1010x	$14	16	Read	Register Select
1011x	$16	16	—	(Reserved)
110xx	$18	32	Write	Instruction Address
111xx*	$1C	32	Read Write	Operand Address

*Not used by the MC68881 or MC68882

Table 9-2. System Data Bus Size Configuration

A0	Size	Data Bus
—	Low	8-Bit
Low	High	16-Bit
High	High	32-Bit

9.2 DATA BUS (D31–D0)

This 32-bit, bidirectional, three-state bus serves as the general-purpose data path between the MC68020/MC68030 (MPU) and the FPCP. Regardless of whether the FPCP is operating as a coprocessor or a peripheral processor, all interprocessor transfers of instruction information, operand data, status information, and requests for service occur as standard M68000 bus cycles.

The FPCP can operate with an 8-, 16-, or 32-bit system data bus. To operate with the required system data bus size, both the A0 and $\overline{SIZE}$ pins must be connected specifically for that applicable bus size. (Refer to **9.1 ADDRESS BUS (A4–A0)** and **9.3 SIZE ($\overline{SIZE}$)** for further details.)

9.3 SIZE ($\overline{SIZE}$)

This active-low input signal is used in conjunction with the A0 pin to configure the FPCP for operation over an 8-, 16-, or 32-bit system data bus. When the FPCP is configured to operate over a 16- or 32-bit system data bus, the $\overline{SIZE}$ and A0 pins must be strapped as listed in Table 9-2.

9.4 ADDRESS STROBE ($\overline{AS}$)

This active-low input signal indicates that there is a valid address on the address bus, and both the chip select ($\overline{CS}$) and read/write (R/$\overline{W}$) signal lines are valid.

9.5 CHIP SELECT ($\overline{CS}$)

This active-low input signal enables the main processor access to the FPCP coprocessor interface registers. When operating the FPCP as a peripheral processor, the chip-select decode is system dependent (i.e., like the chip select on any peripheral). The $\overline{CS}$ signal must be valid (either asserted or negated) when $\overline{AS}$ is asserted. Refer to **10.3 Chip Select Timing** for further discussion of timing restrictions for this signal.

9.6 READ/WRITE (R/$\overline{W}$)

This input signal indicates the direction of a bus transaction (read/write) by the main processor. A logic high (1) indicates a read from the FPCP, and a logic low (0) indicates a write to the FPCP. The R/$\overline{W}$ signal must be valid when $\overline{AS}$ is asserted.

9.7 DATA STROBE ($\overline{DS}$)

This active-low input signal indicates that there is valid data on the data bus during a write bus cycle.

9.8 DATA TRANSFER AND SIZE ACKNOWLEDGE ($\overline{DSACK1}$, $\overline{DSACK0}$)

These active-low, three-state output signals indicate the completion of a bus cycle to the main processor. The FPCP asserts both the $\overline{DSACK1}$ and $\overline{DSACK0}$ signals when the MPU asserts $\overline{CS}$.

If the bus cycle is a main processor read, the FPCP asserts $\overline{DSACK1}$ and $\overline{DSACK0}$ signals to indicate that the information on the data bus is valid. (Both DSACK signals can be asserted in advance of the valid data being placed on the bus.) If the bus cycle is a main processor write to the FPCP, $\overline{DSACK1}$ and $\overline{DSACK0}$ are used to acknowledge acceptance of the data by the FPCP.

The FPCP also uses $\overline{DSACK1}$ and $\overline{DSACK0}$ signals to dynamically indicate to the MPU the port size (system data bus width) on a cycle-by-cycle basis. Depending upon which of the two DSACK pins is asserted for a bus cycle, the MPU assumes data has been transferred to/from an 8-, 16-, or 32-bit wide data port. Table 9-3 lists the DSACK assertions that are used by the FPCP for the various bus cycles over the various system data bus configurations. Refer to **10.1 BASIC TRANSFER MECHANISM OVERVIEW** for details of data bus utilization by the FPCP.

Table 9-3 indicates that all accesses using a 32-bit bus with A4 equal to zero are to 16-bit registers. The FPCP implements all 16-bit coprocessor interface registers on data lines D31–D16 (to eliminate the need for on-chip multiplexors); however, the MPU expects 16-bit registers that are located in a 32-bit port at odd word addresses (A1 = 1) to be implemented on data lines D15–D0. For accesses to these registers when configured for 32-bit

Table 9-3. DSACK Assertions

Data Bus	A4	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$	Comments
32-Bit	1	L	L	Valid Data on D31–D0
32-Bit	0	L	H	Valid Data on D31–D16
16-Bit	x	L	H	Valid Data on D31–D16 or D15–D0
8-Bit	x	H	L	Valid Data on D31–D24, D23–D16, D15–D8, or D7–D0
All	x	H	H	Insert Wait States in Current Bus Cycle

Table 9-3 indicates that all accesses using a 32-bit bus with A4 equal to zero are to 16-bit registers. The FPCP implements all 16-bit coprocessor interface registers on data lines D31–D16 (to eliminate the need for on-chip multiplexors); however, the MPU expects 16-bit registers that are located in a 32-bit port at odd word addresses (A1 = 1) to be implemented on data lines D15–D0. For accesses to these registers when configured for 32-bit bus operation, the MC68881/M68882 generates DSACK signals as listed in Table 9-3 to indicate to the MPU that the valid data is on D31–D16 instead of on D15–D0.

External holding resistors are required to maintain both $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ high between bus cycles. In order to reduce the signal rise time, the $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ lines are actively pulled up (negated) by the FPCP following the rising edge of $\overline{\text{AS}}$ or $\overline{\text{DS}}$, and both DSACK lines are then three-stated (placed in the high-impedance state) to avoid interference with the next bus cycle.

9

9.9 RESET ($\overline{\text{RESET}}$)

This active-low input signal causes the FPCP to initialize the floating-point data registers to nonsignaling not-a-numbers (NANs) and clears the floating-point control, status, and instruction address registers.

When performing a powerup reset, external circuitry should keep the $\overline{\text{RESET}}$ line asserted for a minimum of four clock cycles after V_{CC} is within tolerance. This assures correct initialization of the FPCP when power is applied. For compatibility with all M68000 Family devices, 100 milliseconds should be used as the minimum.

When performing a reset of the FPCP after V_{CC} has been within tolerance for more than the initial powerup time, the $\overline{\text{RESET}}$ line must have an asserted pulse width greater than two clock cycles. For compatibility with all M68000 Family devices, 10 clock cycles should be used as the minimum.

9.10 CLOCK (CLK)

The FPCP clock input is a TTL-compatible signal that is internally buffered for development of the internal clock signals. The clock input must be a constant frequency square wave with no stretching or shaping techniques required. The clock should not be gated off at any time and must conform to minimum and maximum period and pulse width times.

9.11 SENSE DEVICE (SENSE)

This output pin may be used optionally as an additional GND pin or as an indicator to external hardware that the FPCP is present in the system. This signal is internally connected to the GND of the die, but it is not necessary to connect it to the external ground for correct device operation.

Figure 9-2 shows an example of a circuit to sense the presence of an FPCP in a socket prepared for it. The circuit asserts BERR when the MPU selects the coprocessor and no coprocessor is plugged in.

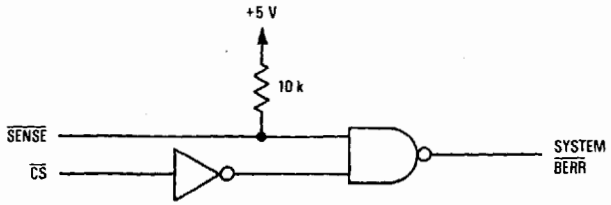


Figure 9-2. Sense Device Circuit Example

9.12 POWER (V_{CC} and GND)

These pins provide the supply voltage and system reference level for the internal circuitry of the FPCP. Care should be taken to reduce the noise level on these pins with appropriate capacitive decoupling.

The FPCP is fabricated in Freescale's advanced HCMOS process and is capable of operating at clock speeds of 25 MHz. Although the use of CMOS for a device containing such a large number of transistors allows significantly reduced power consumption in comparison to an equivalent NMOS device, the high clock speed makes the characteristics of the power supplied to the part quite important. The power supply must be as free from noise as possible, and it must be able to supply large amounts of instantaneous current when the FPCP performs certain operations. In order to meet these requirements, more detailed attention should be given to the power supply connection to the FPCP than is required for older NMOS devices that operate at slower clock rates.

In order to provide a solid power supply interface, four V_{CC} pins, eight primary GND pins, and two secondary GND pins are provided. This allows two V_{CC} and GND pins to supply the power for the data bus, while the remaining V_{CC} and GND pins are used by the internal logic and DSACK drivers. The two secondary GND pins are not intended to provide the main power supply interface, but merely to augment it as required. (One of these pins is the SENSE pin which may be used as an optional GND connection.)

Three V_{CC} and four GND pin positions are reserved for future use by Freescale and should be connected appropriately in order to maintain pin compatability with all future versions of the FPCP. Table 9-4 lists the V_{CC} and GND pin assignments.

In order to reduce the amount of noise in the power supplied to the FPCP, common capacitive decoupling techniques should be observed. While there is no recommended

Table 9-4. V_{CC} and GND Pin Assignments

Devices Supplied	V_{CC}	GND
D31-D16	H8	J8
D15-D00	B8	B7
Internal Logic, $\overline{DSACK1}$, $\overline{DSACK0}$	E2, E9	A2, B2, B3, B4*, C3, E10, K3
Separate	—	C1
Extra	A1, B1, J2	A10, D2, F2, H9

*B4 is the $\overline{SENSE}$ pin and may be used optionally as a ground pin or to detect the presence of the FPCP in the system.

layout for this capacitive decoupling, it is suggested that a combination of low, middle, and high frequency filter capacitors be placed as close to the chip as possible. (For example, a set of 10 μF, 0.1 μF, and 330 pF capacitors in parallel provides filtering for nearly the entire frequency spectrum present in a digital system.) In a system that utilizes the MC68020 as the main processor, these capacitive decoupling practices should also be observed for the main processor. In particular, the 10 μF "tank" capacitor should be reasonably close to both devices (since the two devices are typically placed next to each other on a board) to provide for the high instantaneous current requirements of both the MPU and the FPCP.

In addition to the capacitive decoupling of the power supply, care should be taken to ensure a low resistance connection between the FPCP V_{CC} and GND pins and the system power supply traces. In particular, the connections to pins B7 and J8 (the GND pins for the data bus pins) must have very low resistance. This is necessary because a read of the FPCP can cause the data bus drivers to sink very large amounts of current to ground in order to pull the data bus signals low (if the data pattern that is read contains mostly zeros). If low resistance connections are not provided on pins B7 and J8, the ground potential internal to the package may rise, the fall time of the data signals may be increased, and the low output voltage noise margin may be reduced.

9.13 NO CONNECT (NC)

One pin of the FPCP package is designated as a no connect (NC). This pin position is reserved for future use by Freescale and should neither be used for signal routing nor connected to V_{CC} or GND.

9.14 SIGNAL SUMMARY

Table 9-5 provides a summary of all the FPCP signals described in this section.

Table 9-5. Signal Summary

Signal Name	Mnemonic	Input/Output	Active State	Three State
Address Bus	A4–A0	Input	High	—
Data Bus	D31–D0	Input/Output	High	Yes
Size	$\overline{SIZE}$	Input	Low	—
Address Strobe	$\overline{AS}$	Input	Low	—
Chip Select	$\overline{CS}$	Input	Low	—
Read/Write	R/$\overline{W}$	Input	High/Low	—
Data Strobe	$\overline{DS}$	Input	Low	—
Data Transfer and Size Acknowledge	$\overline{DSACK1}$, $\overline{DSACK0}$	Output	Low	Yes
Reset	$\overline{RESET}$	Input	Low	—
Clock	CLK	Input	—	—
Sense Device	$\overline{SENSE}$	Input/Output	Low	No
Power Input	V_{CC}	Input	—	—
Ground	GND	Input	—	—

SECTION 10
BUS OPERATION

This section describes the functional characteristics of the MC68881/MC68882 (FPCP) bus interface and the mechanisms used to execute data transfers between the FPCP and the main processor. This discussion includes descriptions of the functional characteristics of individual bus cycles as well as descriptions of the operand transfer protocols that require multiple bus cycles.

Although the FPCP is designed primarily for use as a coprocessor to the MC68020/MC68030 (MPU), there are no characteristics of the bus operation that preclude the use of the FPCP as a peripheral device with any other processor. This is because the M68000 Family coprocessor interface utilizes standard bus cycles to transfer instructions and data between the main processor and coprocessors in a system, with no special signals required for these transfers. Because of this general-purpose transfer mechanism, the type of the main processor and the nature of the system bus interface are transparent to the FPCP.

10.1 BASIC TRANSFER MECHANISM OVERVIEW

In order to execute a floating-point instruction, the FPCP and the main processor communicate using a series of bus cycles, instructions, and data according to a predefined protocol as described in **7.5 INSTRUCTION DIALOGS**. Most of these bus cycles transfer an entire item in a single transfer, although large items such as extended precision floating-point numbers require multiple bus cycles to transfer the entire operand. Also, if an FPCP port size of 8 or 16 bits is selected, multiple bus cycles can be required to transfer items that can be transferred with a single cycle over a 32-bit port.

The communications mechanism utilized by the FPCP and the main processor uses a set of mail-box registers, called the coprocessor interface registers (CIRs), to move data, instructions, and control information between the devices. The characteristics of the CIRs and the manner in which they are used by the FPCP and a main processor are described in **SECTION 7 COPROCESSOR INTERFACE**. The discussions in the following paragraphs are not specific to any particular CIR or instruction protocol, except where noted.

When a single bus cycle is able to accommodate an entire item, the transfer mechanism is obviously quite simple and the only requirement that must be met is that the bit alignment of the FPCP and main processor match. Figure 10-1 shows the bit assignment and significance of the 32-bit data bus of the FPCP, which must be matched to the main processor (for the MPU, this matching is accomplished by connecting D31 of the FPCP to D31 of the MC68020, D30 to D30, etc.).

When multiple bus cycles are required to transfer an item, the additional requirements of correct transfer order and port alignment must also be met. Figure 10-2 shows the data alignment of the FPCP for each port size. In this figure, if a section of the data bus is shaded for a particular encoding of $\overline{\text{SIZE}}$, A4, A1 and A0, that section of the data bus is active during the transfer (i.e., valid data is expected during a write cycle, and the bus is driven

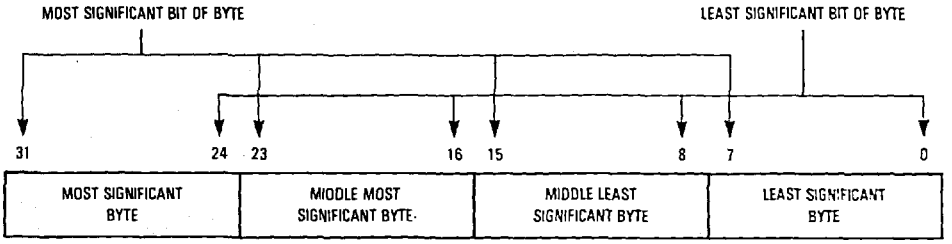


Figure 10-1. FPCP Data Bus Bit Assignments

$\overline{SIZE}$	A4	A1	A0	PORT SIZE	$\overline{DSACK1}$/$\overline{DSACK0}$	ACTIVE DATA BUS SECTIONS
H	0	x	1	32 BITS	L H	
H	1	x	1		L L	
H	0	x	0	16 BITS	L H	
H	1	0	0		L H	
H	1	1	0		L H	
L	0	x	0	8 BITS	H L	
L	0	x	1		H L	
L	1	0	0		H L	
L	1	0	1		H L	
L	1	1	0		H L	
L	1	1	1		H L	

Figure 10-2. Data Bus Activity vs Port Size and Operand Alignment

during a read cycle). Otherwise, it is idle during the transfer. Note that the port size is not determined by the $\overline{SIZE}$ pin alone, but by the combination of the $\overline{SIZE}$ pin and A0. The following paragraphs describe the transfer order for each port size.

10.1.1 32-Bit Port Size

When $\overline{SIZE}$ and A0 are both high, the FPCP port size is defined to be 32 bits. In most cases, this configuration is statically selected by connecting the $\overline{SIZE}$ and A0 pins directly to V_{CC}; although dynamic port size selection is possible if the proper timing constraints are followed for the $\overline{SIZE}$ and A0 pins. Although this configuration selects a 32-bit port size, the FPCP utilizes the dynamic bus sizing capabilities of the MPU to reduce the amount of multiplexing logic on the chip. The value of A4 during a bus cycle determines which bytes of the 32-bit port are used to drive or receive data. Since all of the coprocessor interface registers in the lower half of the CIR address range (A4 = 0, offsets $00 through $0F) are 16-bit registers, dynamic bus sizing is utilized to place all of those CIRs on data bus pins D32–D16, and the

DSACK encoding returned indicates a 16-bit port size. All of the CIRs in the upper half of the CIR address range (A4 = 1, offsets $10–$1F) are either 32-bit registers or 16-bit registers paired with undefined register locations. Therefore, the DSACK encoding used to terminate accesses in this range indicates a 32-bit port (during a read of the register select CIR, data bits 15–0 are undefined, reserved, and are driven high). In both of these cases, A4 determines the DSACK encoding that is returned, and A1 selects the appropriate word location. A0 is always one, to select a 32-bit FPCP port size, and thus individual bytes cannot be accessed in this configuration. Furthermore, the FPCP always expects a full 16 or 32 bits of data to be transferred during a bus cycle when $\overline{SIZE}$ is high; A0 is one, and A4 is zero or one, (with the exception of immediate byte or word operands, as discussed in the next paragraph).

When the FPCP is used in a 32-bit configuration, most CIR accesses transfer an entire instruction or data item in a single bus cycle. The one exception to this is for accesses to the operand CIR, which is used to transfer large items such as floating-point numbers and state frames. When an item is larger than four bytes, multiple accesses of the operand CIR are required to complete the transfer. In this case, the correct transfer order must be observed, in addition to the bit and byte alignment previously discussed. In all cases, each part of an item is transferred with the most significant bit aligned with bit 31 of the operand CIR (i.e., they are transferred across D31–D24, D31–D16, or D31–D0 for bytes, words, or long words, respectively). With the exception of byte and word immediate operands, the FPCP never requests the transfer of an item that is not a multiple of four bytes in length. An immediate byte or word operand is transferred in a single bus cycle and is left-aligned with the operand CIR. All other operands are transferred through the operand CIR in 32-bit units until the entire item is transferred.

When multiple bus cycles are required to transfer an item, the first operand CIR access transfers the most significant long word of the item; each successive access transfers the next least significant long word. For example, when an extended precision number is moved, the first operand CIR access is used to transfer bits 95–64 of the operand, the second access transfers bits 63–32, and the third access transfers bits 31–0 to complete the operand transfer. Note that the manner in which the operand is read from or written to memory is transparent to the FPCP, which allows the operand to be stored in memory in the native format of the main processor.

The amount of data transferred with each access to the operand CIR is dependent on the state of an instruction dialog and is determined by the FPCP, not the main processor. For example, if the FPCP issues an evaluate effective address and transfer data primitive with a length of 12 bytes, three accesses of the operand CIR are expected (with each access transferring four bytes). Thus, for a 32-bit port, the main processor is not allowed to transfer the operand with a series of word or byte transfers, but must use long-word transfers to move the operand.

10.1.2 16-Bit Port Size

When $\overline{SIZE}$ is high and A0 is low, the FPCP port size is defined to be 16 bits. In most cases, this configuration is statically selected by connecting the $\overline{SIZE}$ and A0 pins directly to V_{CC} and GND, respectively, although dynamic port size selection is possible if the proper timing constraints are followed for the $\overline{SIZE}$ and A0 pins. Although A0 = 0 in this case, this value is not specifically used to select even byte addresses; rather, it is used to configure the data port to be 16 bits wide. When the FPCP is configured in this manner, all CIR accesses

are assumed to transfer a full 16 bits to the word address selected by A1 (except for the case of an immediate byte operand, as discussed in a following paragraph). The DSACK encoding returned always indicates that the port is 16 bits wide; individual bytes cannot be accessed in this configuration.

In order to eliminate the need for on-chip multiplexing, the FPCP drives data on or receives data from only 16 bits of the data bus, depending on the encoding of A1 and A4 (thus allowing D31 and D15 of the FPCP to be tied together, D30 to be tied to D14, D29 to D13, etc., as described in **SECTION 11 INTERFACING METHODS**. For all accesses with A4 equal to zero, or with A4 equal to one and A1 equal to zero, data is transferred across D31–D16. Data is transferred across D15–D0 when A4 and A1 are both equal to one.

When the FPCP is used in the 16-bit configuration, most CIR accesses transfer an entire instruction or data item in a single bus cycle. The one exception to this is for accesses to the operand CIR, which is used to transfer large items such as floating-point numbers and state frames. When an item is larger than two bytes, multiple accesses of the operand CIR are required to complete the transfer. In this case, the correct transfer order must be observed, in addition to the bit and byte alignment previously discussed. In all cases, each part of an item is transferred with the most significant bit aligned with bit 31 or bit 15 of the operand CIR, depending on the value of A4 and A1 as described in the previous paragraph. With the exception of byte and word immediate operands, the FPCP never requests the transfer of an item that is not a multiple of four bytes in length. Immediate byte operands are transferred in a single bus cycle and are left-aligned with the operand CIR (i.e., they are transferred across D31–D24). All other operands are transferred through the operand CIR in 16-bit units until the entire item is transferred.

When multiple bus cycles are required to transfer an item, the first operand CIR access transfers the most significant word of the item; each successive access transfers the next least significant word. For example, when an extended precision number is moved, the first operand CIR access is used to transfer bits 95–80 of the operand, the second access transfers bits 79–64, and the third through sixth accesses transfer bits 63–48, 47–32, 31–16 and 15–0, respectively, to complete the operand transfer. Note that the manner in which the operand is read from or written to memory is transparent to the FPCP, which allows the operand to be stored in memory in the native format of the main processor.

The amount of data transferred with each access to the operand CIR is dependent on the state of an instruction dialog and is determined by the FPCP, not the main processor. For example, if the FPCP issues an evaluate effective address and transfer data primitive with a length of 12 bytes, six accesses of the operand CIR are expected (with each access transferring two bytes). Thus, for a 16-bit port, the main processor is not allowed to transfer the operand with a series of long-word or byte transfers, but must use word transfers to move the operand.

10.1.3 8-Bit Port Size

When the $\overline{\text{SIZE}}$ signal is low, the FPCP port size is defined to be eight bits. In most cases, this configuration is statically selected by connecting the $\overline{\text{SIZE}}$ pin directly to GND, although dynamic port size selection is possible if the proper timing constraints are followed for the $\overline{\text{SIZE}}$ and A0 pins. In this case, the value of A0 is used to select the correct byte address, rather than to configure the data port size. When the FPCP is configured in this manner, all CIR accesses transfer one byte to the address selected by A4–A0, and the DSACK

encoding returned always indicates that the port is 8 bits wide. In order to eliminate the need for on-chip multiplexing, the FPCP drives data on or receives data from only 8 bits of the data bus, depending on the encoding of A0, A1 and A4 (thus allowing D31, D23, D15 and D7 of the FPCP to be tied together; D30 to be tied to D22, D14 and D6; D29 to D21, D13 and D5, etc., as described in **SECTION 11 INTERFACING METHODS**). Figure 10-2 shows which bytes of the data bus are driven or received for each encoding of the A0, A1, and A4 lines.

When the FPCP is used in the 8-bit configuration, most transfers require multiple CIR transfers to move an entire instruction or data item. The one exception to this is for accesses to the operand CIR to transfer a byte immediate operand. When an item is larger than one byte, multiple accesses of the appropriate CIR are required to complete the transfer. In this case, the correct transfer order must be observed, in addition to the bit and byte alignment previously discussed. In all cases, each part of an item is transferred with the most significant bit aligned with bit 31, 23, 15, or 7 of the FPCP, depending on the value of A0, A1, and A4 as described in the previous paragraph. With the exception of byte and word immediate operands, the FPCP never requests the transfer of an item that is not a multiple of four bytes in length. Immediate byte operands are transferred in a single bus cycle and are left-aligned with the operand CIR (i.e., they are transferred across D31–D24). All other operands are transferred through the appropriate CIR in 8-bit units until the entire item is transferred.

When multiple bus cycles are required to transfer an item, the first operand CIR access transfers the most significant word of the item; each successive access transfers the next least significant word. For example, when an extended precision number is moved, the first operand CIR access is used to transfer bits 95–88 of the operand, the second access transfers bits 87–80, and the third through twelfth accesses transfer bits 79–72, 71–64, 63–56, 55–48, 47–40, 39–32, 31–24, 23–16, 15–8 and 7–0, respectively, to complete the operand transfer. Note that the manner in which the operand is read from or written to memory is transparent to the FPCP, which allows the operand to be stored in memory in the native format of the main processor.

The amount of data transferred with each access to the operand CIR is dependent on the state of an instruction dialog and is determined by the FPCP, not the main processor. For example, if the FPCP issues an evaluate effective address and transfer data primitive with a length of 12 bytes, 12 accesses of the operand CIR are expected (with each access transferring one byte). Thus, for an 8-bit port, the main processor is not allowed to transfer the operand with a series of word or long-word transfers, but must use byte transfers to move the operand.

10.2 RESET OPERATION

Before the FPCP can be used for any operation after power has been applied to the system, it must be initialized using a hardware reset function. This is done when power is initially applied to the system by asserting $\overline{\text{RESET}}$ for at least four clock cycles (with reference to the FPCP CLK signal) after V_{CC} has reached the nominal operating level. After power has been stable and the FPCP has executed a power-up reset operation, a subsequent reset operation may be initiated by asserting $\overline{\text{RESET}}$ for at least two cycles of the FPCP CLK signal. Note that in order to maintain compatibility with all M68000 Family devices, the power-on reset pulse for a system should be a minimum of 100 ms, while a 10 clock minimum (with respect to the clock signal of the slowest M68000 Family device in the system) should be used for reset operations after power is stable.

When a hardware reset operation is performed, the FPCP immediately aborts any operation that may have been in progress and returns to the idle state. All of the floating-point data registers are loaded with nonsignaling NANs, and the FPCR and FPSR are cleared to all zeros (thus clearing any old status information and selecting the IEEE standard default operating modes). An identical operation may be performed under software control by a FRESTORE of a null state frame (although a hardware reset must be executed at power-up in order to initialize the FPCP).

One consideration that should be given to the $\overline{\text{RESET}}$ signal of the FPCP is the treatment of a RESET instruction by an M68000 Family processor. When the RESET instruction is executed by an M68000 Family processor, the internal state of the processor is not affected, but the external system should respond to the reset operation. Since the FPCP is considered to be part of the internal state of the main processor, prudent system design suggests that the FPCP should not respond to the assertion of the $\overline{\text{RESET}}$ signal by the main processor. This can be accomplished in many ways, depending on the requirements of the system. A simple circuit to support this operation is shown in Figure 10-3. If a software RESET function is needed, it is suggested that this be implemented by an FRESTORE of a null frame.

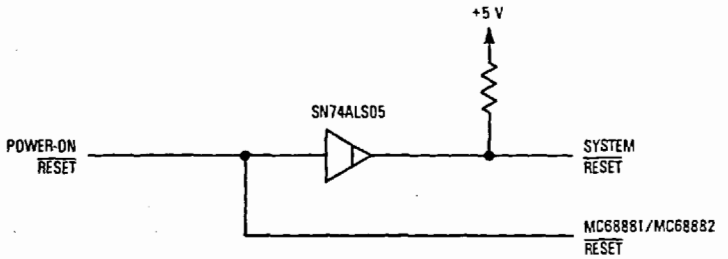


Figure 10-3. FPCP Reset Logic Example

10

10.3 CHIP SELECT TIMING

Most of the bus cycle timing requirements of the FPCP are straightforward, with all signal timing following the normal M68000 Family conventions. The only signal timing that is specific to the FPCP bus interface is the relationship of the assertion of chip select ($\overline{\text{CS}}$) to the assertion of the address strobe ($\overline{\text{AS}}$) and data strobe ($\overline{\text{DS}}$). Unlike most M68000 Family peripherals that require the assertion of $\overline{\text{CS}}$ to follow the assertion of $\overline{\text{AS}}$ or $\overline{\text{DS}}$, the FPCP allows the $\overline{\text{CS}}$ assertion to precede the assertion of the $\overline{\text{AS}}$ and $\overline{\text{DS}}$.

In order to detect the start or end of an access, the FPCP monitors the $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{CS}}$, and R/$\overline{\text{W}}$ signals. A cycle start is detected when $\overline{\text{AS}}$, $\overline{\text{CS}}$, and $\overline{\text{DS}}$ or R/$\overline{\text{W}}$ (for a write cycle) are asserted, and a cycle end is detected when the first strobe ($\overline{\text{AS}}$ or $\overline{\text{DS}}$) is negated. The order in which these signals are sequenced is not critical to correct operation, and in the case of $\overline{\text{CS}}$ the occurrence of a negated or asserted edge is not needed to detect a new access. For example, it is not required that $\overline{\text{CS}}$ be negated between successive accesses to the FPCP, since the negation and assertion of the $\overline{\text{AS}}$ and $\overline{\text{DS}}$ signals causes the FPCP to detect the end of one access and the start of the next.

The FPCP conditions the $\overline{\text{DSACK}}$ generation logic internally with $\overline{\text{AS}}$ and $\overline{\text{CS}}$. To ensure that $\overline{\text{DSACKx}}$ assertion is not delayed longer than necessary, $\overline{\text{CS}}$ should be asserted before

$\overline{AS}$ is asserted (since $\overline{CS}$ is system dependent but $\overline{AS}$ is MPU dependent). This design is called "early chip select". On the other hand, when $\overline{CS}$ is asserted after $\overline{AS}$ has been asserted, the design is called "late chip select". A late chip-select design may add wait states to the FPCP accesses.

A timing restriction on $\overline{CS}$ occurs on a FPCP access followed immediately by a non-FPCP access. $\overline{CS}$, which is asserted during the FPCP access, must negate in time for it to occur before the assertion of $\overline{AS}$ of the subsequent non-FPCP access.

To satisfy this timing restriction with an early chip select, neither $\overline{AS}$ or $\overline{DS}$ can be used to generate $\overline{CS}$. Figure 10-4 shows some circuits that correctly generate an early chip-select signal for MPU-based systems. Note that in these circuits only the following terms are included in the $\overline{CS}$ equation:

- FC2-FC0 = 7 — CPU Space
- A19-A16 = 2 — Coprocessor Communications
- A15-A13 = 1 — Cp-ID One (Freescale Assembler Default)

For systems that use the MC68020 or the MC68030 with an FPCP, the maximum time for an early chip select is:

$$t_{AVCS} = t_{AVSA}$$

where:

t_{AVCS} = Address/function code valid to $\overline{CS}$ asserted (maximum).
t_{AVSA} = MC68020/MC68030 address/function code valid to $\overline{AS}$ asserted (AC electrical specification #11 minimum).

For a 20-MHz MPU and a 25-MHz FPCP:
t_{AVCS} = 10 ns maximum.

The 74AS02 and 74AS30 implementation shown in Figure 10-4 or a PAL implementation witha maximum decode delay time of 10 ns may be used.

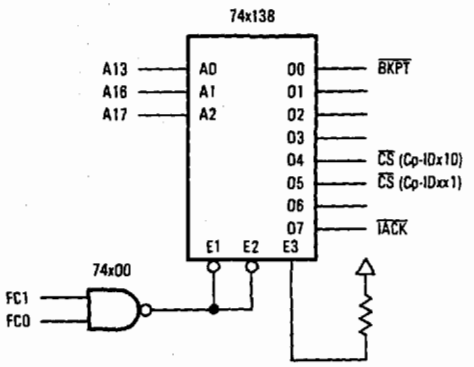
For a 25-MHz and 33-MHz MPU/FPCP system, refer to EB116 entitled *Chip-Select Generation for a 33.33-MHz MC68030 Microprocessor and a 33.33-MHz MC68882 Floating-Point Co-processor.*

A late chip-select design can use slower (and, therefore, less expensive) logic. To implement this design, the $\overline{CS}$ generation logic should include $\overline{AS}$. Another consideration in using slower logic is that if a non-FPCP access follows an FPCP access, the $\overline{CS}$ for the FPCP must not remain asserted inadvertently during the non-FPCP access. However, $\overline{AS}$ should be included in the decode logic as shown in Figure 10-5(A), along with the timing that the logic provides. When $\overline{AS}$ is used in an AND gate with the decode logic output as shown Figure 10-5(B), $\overline{CS}$ is asserted after the start of the non-FPCP access, as the timing diagram shows. This implementation in Figure 10-5(B) is incorrect.

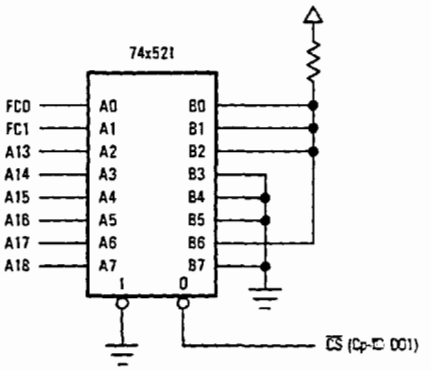
10.4 BUS CYCLE FUNCTIONAL DESCRIPTIONS

The FPCP executes three types of bus cycles, according to the direction of the transfer and the CIR that is selected by the main processor. The three bus cycle types are: synchronous read cycles, asynchronous read cycles, and asynchronous write cycles. In this context, the terms synchronous and asynchronous convey slightly different meanings than when they

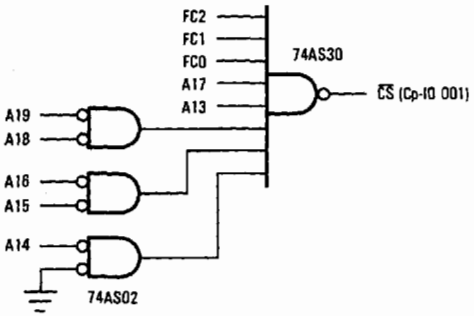
FC0-FC2 —$7
CPU Space (A16-A19) —$2
Cp-ID (A13-A15) —$1 (default)



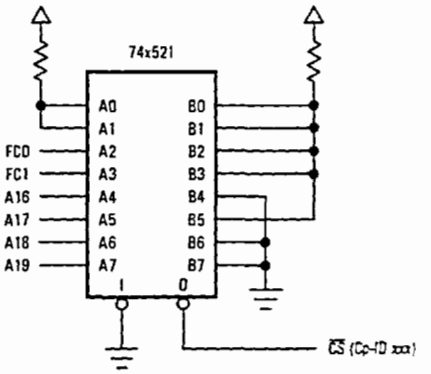
Up to two coprocessors in the system, with additional decode for BKPT and IACK cycles.



Up to seven coprocessors in the system.



Up to seven coprocessors in the system.



Only one coprocessor in the system.

Decode Delay Times*

Programmable Array Logic (PAL)	= 10 - 25 ns max.
74x00/74x138 combined propagation delay	= 9.5 ~ 12.8 ns max.
74x521 compare delay	= 5.5 ~ 11 ns max.
74AS02/74AS30 combined propagation delay	= 9 ns max.

*The 'x' represents various combinations of logic families including F, AS, or fast CMOS.

Figure 10-4. Example of Early Chip Select Circuits

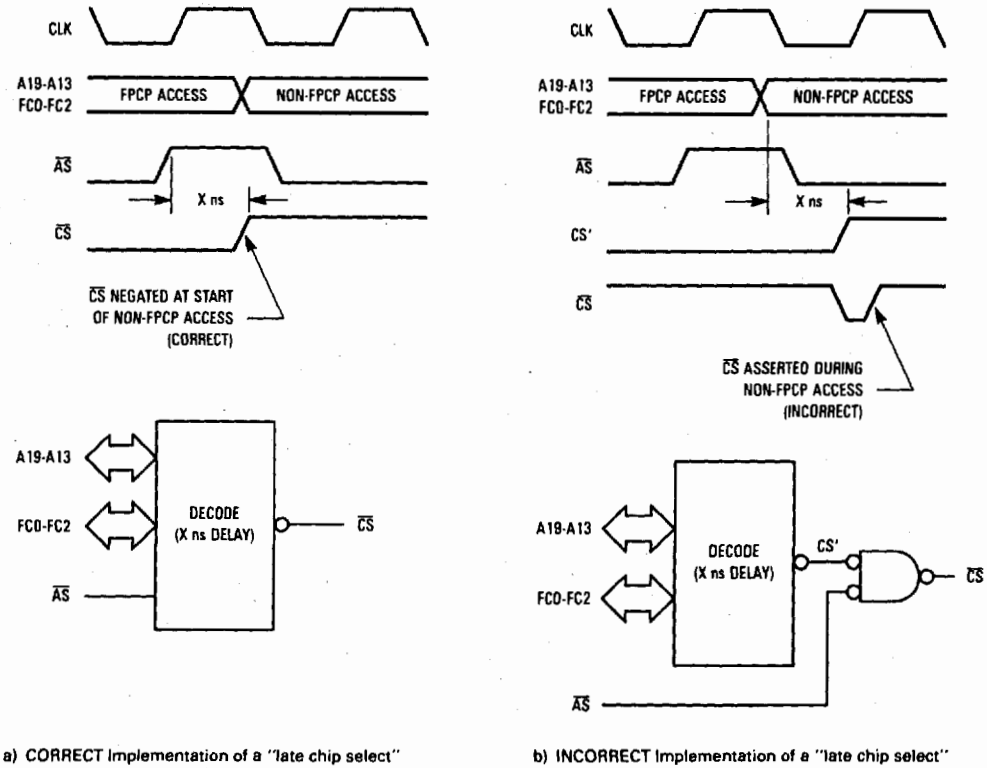


Figure 10-5. Example of Late Chip-Select Circuit

are used to describe the bus transfer characteristics of a microprocessor (e.g., where the MC68030 microprocessor and an external device perform either synchronous or asynchronous cycles, depending on the relationship between the clocks used in the MPU and the external device). Here, the terms synchronous and asynchronous are used with respect to the FPCP clock. The following paragraphs describe the functional characteristics of each bus cycle type (for an AC parametric description of the FPCP bus interface, refer to **SECTION 12 ELECTRICAL SPECIFICATIONS**).

In the following discussions, the main processor is assumed to be an MC68020 or MC68030, with the FPCP and the MPU both driven by the same clock signal. Thus, the terminology and conventions used are identical to the bus description for the MPU. This clock frequency relationship is not required, but the following discussions are simplified by assuming that both devices use the same clock signal. Where appropriate, references are made to variations in bus cycle operations if the main processor is not an MC68020 or MC68030.

10.4.1 Synchronous Read Cycles

When the main processor performs a read access to either the response or save CIR, the FPCP responds by executing a synchronous read bus cycle. In this context, the term synchronous signifies that the bus cycle timing is directly related to the FPCP clock signal, but

the FPCP clock is not required to be synchronous with the main processor clock during the transfer. By synchronizing the bus cycle to the FPCP clock, the appropriate response primitive or format word is always returned based on the current status of the FPCP. Also, since these bus cycles are used to transmit service requests to the main processor, the synchronous bus cycle timing allows the main processor and FPCP to be synchronized at critical points in an instruction dialog, without requiring synchronous clock signals for the two devices.

The functional timing for the synchronous read cycle is shown in Figure 10-6. The FPCP detects the start of a synchronous read cycle when chip select and address strobe are asserted, read/write is high, and the address pins are encoded to $00 (to select the response CIR) or $04 (to select the save CIR). When either of these conditions is met, the FPCP begins to sample the address strobe, data strobe, and chip-select lines on each rising edge of the CLK signal. When all three of these signals are sampled as asserted, the FPCP latches certain internal state flags and uses those flags to determine the appropriate response primitive or format word to be placed on the data bus. One and one-half clock cycles later, the FPCP begins to drive the data value onto the bus and assert the appropriate data transfer and size acknowledge encoding. The data value remains on the data pins and $\overline{\text{DSACKx}}$ remains asserted until the first of the two signals, $\overline{\text{AS}}$ or $\overline{\text{DS}}$, is negated; then the bus cycle is terminated by placing the data bus in the high-impedance state and negating $\overline{\text{DSACKx}}$.

As shown in Figure 10-6, this type of bus cycle requires five clock cycles (two wait cycles) when the MPU and the FPCP share the same clock. Under certain conditions, these bus

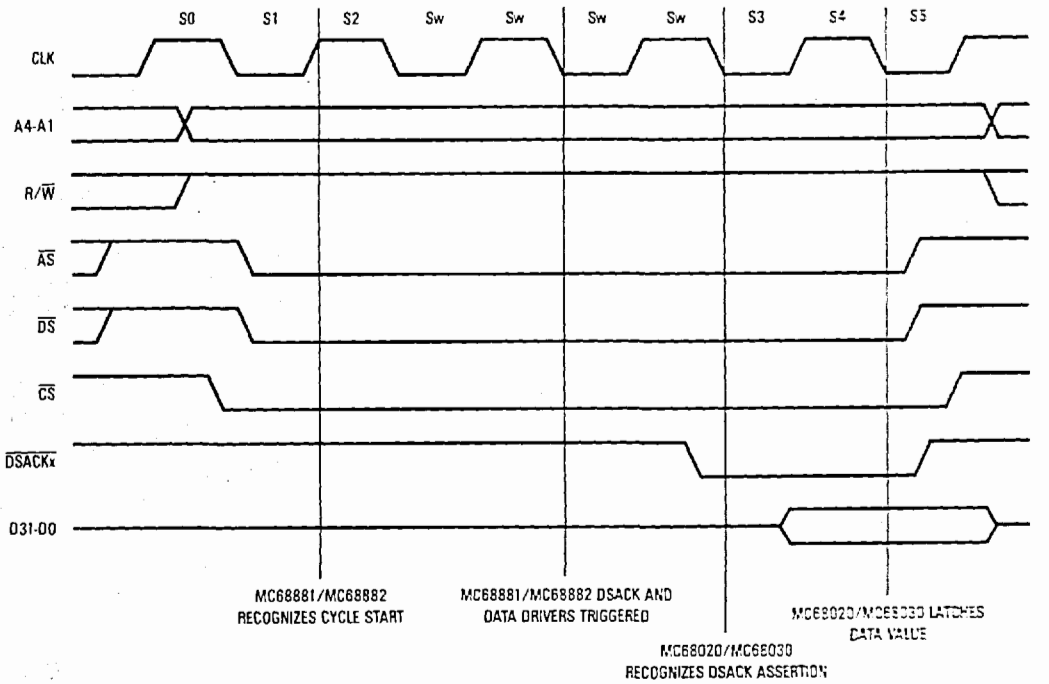


Figure 10-6. Synchronous Read-Cycle Timing Diagram

cycles may require as many as six or seven clock cycles. Two separate mechanisms determine whether additional clock cycles are required for this type of a bus cycle:

1. The relationship between the assertion of $\overline{AS}$, $\overline{DS}$, or $\overline{CS}$ and the rising edge of the FPCP clock signal
2. The relationship between the assertion of $\overline{DSACKx}$ by the FPCP and the falling edge of the MPU clock signal

As previously described, $\overline{DSACKx}$ is triggered to assert one and one-half clock cycles after $\overline{AS}$, $\overline{DS}$, and $\overline{CS}$ are sampled as asserted; thus, the best-case timing occurs when all three of these signals are asserted as early in the bus cycle as possible. Since the MPU triggers the assertion of $\overline{AS}$ and $\overline{DS}$ with the falling edge of the CLK signal (which is assumed to be the same for both devices) and the FPCP samples those signals, along with $\overline{CS}$, on the rising edge of the CLK signal, the best-case cycle timing occurs only if $\overline{AS}$, $\overline{DS}$, and $\overline{CS}$ are all asserted to provide the required setup time to the next rising edge of the clock. Thus, the maximum assertion and propagation delays for these signals must be less than the clock pulse width low in order to guarantee the best-case bus cycle timing. Although the maximum specifications for the assertion, by the MPU, of $\overline{AS}$ or $\overline{DS}$ from the falling edge of the clock do not guarantee the best-case timing for operation at 16.67 MHz under worst-case system environments, the best-case timing normally occurs under typical system conditions. In order to assure the possibility that the best-case timing occurs, system designers should utilize the $\overline{CS}$ generation methods described in **10.3 CHIP SELECT TIMING** to prevent propagation delays of the $\overline{CS}$ logic from lengthening the bus cycle by one clock.

In the same manner as just described (where the FPCP misses the assertion of $\overline{AS}$, $\overline{DS}$, or $\overline{CS}$), one clock cycle may be added to the bus cycle timing if the MPU misses the assertion of $\overline{DSACKx}$ by the FPCP. The assertion of $\overline{DSACKx}$ by the FPCP is triggered by the falling edge of the clock, and the propagation delay for this assertion can be quite long (slightly longer than one 16.67 MHz clock cycle under worst-case system conditions). Since the MPU samples $\overline{DSACKx}$ on the falling edge of the clock, the assertion of $\overline{DSACKx}$ triggered by a given falling clock edge may not be completed ahead of the setup time to the next falling clock edge. There is very little that a system designer can do to assure that the $\overline{DSACKx}$ assertion is recognized on the first falling clock edge after it is triggered, since the propagation delay is dependent on individual device characteristics as well as system conditions such as temperature and power supply levels.

Due to the nature of the two mechanisms just described, it is possible that for an individual system the bus cycle timing for synchronous read cycles may be different under varying system conditions. For example, when a system is first turned on (and thus the devices are at room temperature) it is quite likely that synchronous read cycles require five clock cycles as shown in Figure 10-6. As the temperature increases to the normal operating range, the synchronous read cycle timing may change to six clock cycles. If the temperature rise affects both of the synchronization mechanisms enough (particularly if the $\overline{CS}$ generation logic causes the assertion of $\overline{CS}$ to follow the assertion of $\overline{AS}$ and/or $\overline{DS}$), the timing for these operations may increase to seven clock cycles or even vary on a cycle-by-cycle basis between six and seven clock cycles. Some other factors that may affect the timing for synchronous reads cycles are the power supply levels for the FPCP and MPU, the individual device characteristics (due to manufacturing variances), and the capacitive loading of the control signals.

It should be noted that the timing variances for synchronous read cycles do not affect the overall performance of a system significantly. Specifically, one or two additional clock

cycles per synchronous read cycle results in a small percentage change in the overall execution time for an instruction (since most instructions typically require over 50 clock cycles to execute). The only environment where these timing variances may be of concern is when a programmer is attempting to optimize an instruction sequence for maximum overlap. In this case, these factors should be added to the instruction execution timing variability mechanisms discussed in **SECTION 8 INSTRUCTION EXECUTION TIMING**.

10.4.2 Asynchronous Read Cycles

When the main processor performs any access to the FPCP with R/$\overline{W}$ high other than a read of the response or save CIR, the FPCP responds by executing an asynchronous read cycle. In this context, the term asynchronous signifies that the bus cycle timing is not related to the FPCP or MPU clock signals in any way. The FPCP supports this type of operation by implementing all of the CIRs, except the response and save CIRs, as dual ported structures. Thus, the main processor can access these CIRs at the maximum speed regardless of the clock frequency of the FPCP, while the FPCP internally accesses these CIRs in a synchronous manner.

The functional timing for the asynchronous read cycle is shown in Figure 10-7. The FPCP detects the start of an asynchronous read cycle when chip select, address strobe, and data strobe are asserted; read/write is high; and the address pins are not encoded to S00 or $04 (which selects the response or save CIR, respectively). When this condition is met, the FPCP responds by placing the data from the selected CIR on the data bus and asserting the appropriate data transfer and size acknowledge encoding. The data value remains on the data pins and $\overline{DSACKx}$ remains asserted until the first of the two signals, $\overline{AS}$ or $\overline{DS}$, is negated at which time the bus cycle is terminated by placing the data bus in the high-impedance state and negating $\overline{DSACKx}$.

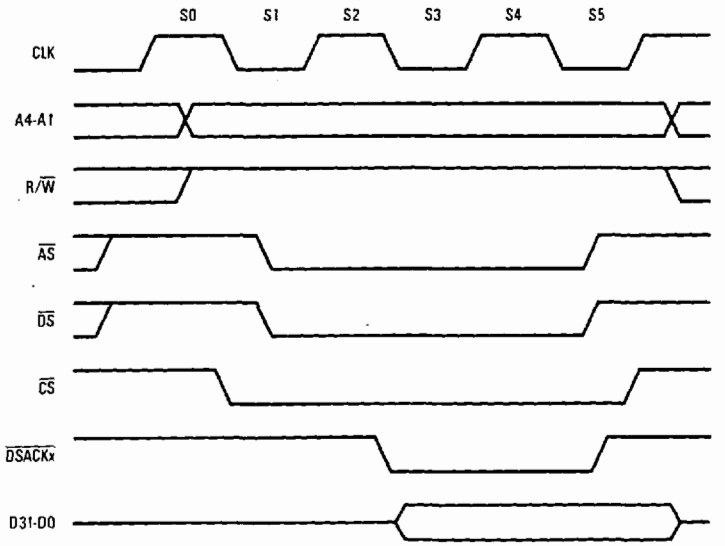


Figure 10-7. Asynchronous Read Cycle Timing Diagram

As shown in Figure 10-7, this type of bus cycle requires three clock cycles (no wait cycles) when the MPU and the FPCP share the same clock. Due to the asynchronous timing of the data transfer and size acknowledge assertion by the FPCP, this bus cycle timing does not depend on the clock frequency of the FPCP (there are some exceptions to this rule, as discussed in **10.5 INTER-CYCLE TIMING RESTRICTIONS**). For example, if the MC68881 clock frequency is 12.5 MHz and the MPU clock frequency is 16.67 MHZ, this bus cycle requires three MPU clock cycles since the assertion of $\overline{DSACKx}$ is recognized by the MPU on the falling edge of S2. This assumes that the chip-select logic causes the assertion of $\overline{AS}$ and $\overline{DS}$ so that the $\overline{AS}$/$\overline{DS}$ assertion delay is not lengthened by the chip-select logic propagation time.

10.4.3 Asynchronous Write Cycles

When the main processor performs any access to the FPCP with R/$\overline{W}$ low, the FPCP responds by executing an asynchronous write cycle. The definition of asynchronous in the first paragraph of the preceding subsection applies also to asynchronous write cycles.

The functional timing for the asynchronous write cycle is shown in Figure 10-8. The FPCP detects the start of an asynchronous write cycle when chip select and address strobe are asserted and read/write is low. When this condition is met and an asserted pulse occurs on $\overline{DS}$, the FPCP responds by asserting the appropriate data transfer and size acknowledge encoding and latching the value of the data bus into the selected CIR. The $\overline{DSACKx}$ encoding remains asserted until $\overline{AS}$ is negated; then the bus cycle is terminated by negating $\overline{DSACKx}$.

As shown in Figure 10-8, this type of bus cycle requires three clock cycles (no wait cycles) when the MPU and the FPCP share the same clock. Due to the asynchronous timing of the data transfer and size acknowledge assertion by the FPCP, this bus cycle timing does not depend on the clock frequency of the FPCP (there are some exceptions to this rule, as

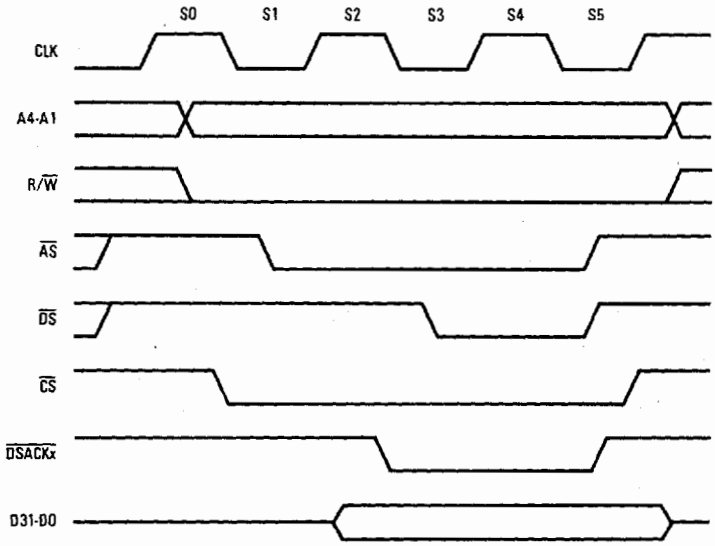


Figure 10-8. Asynchronous Write Cycle Timing Diagram

discussed in **10.5 INTER-CYCLE TIMING RESTRICTIONS**). For example, if the MC68881 clock frequency is 12.5 MHz and the MPU clock frequency is 16.67 MHz, this bus cycle requires three MPU clock cycles since the assertion of $\overline{\text{DSACKx}}$ is recognized by the MPU on the falling edge of S2. This assumes that the chip-select logic causes the assertion of $\overline{\text{CS}}$ to precede the assertion of $\overline{\text{AS}}$ and $\overline{\text{DS}}$ so that the $\overline{\text{AS}}$/$\overline{\text{DS}}$ assertion to $\overline{\text{DSACKx}}$ assertion delay is not lengthened by the chip-select logic propagation time.

10.5 INTER-CYCLE TIMING RESTRICTIONS

The bus interface of the MC68881 is designed to operate satisfactorily at any reasonable clock frequency relationship between the MC68881 and the main processor. In most cases, differences in the clock frequency of the two devices does not affect the operation of the bus; and particularly, it does not affect the timing of individual bus cycles. However, there are some cases where the timing of a bus cycle is modified if the MC68881 is overrun by the main processor.

During coprocessor interface dialogs, certain bus cycles trigger actions by the FPCP on the negated edge of data strobe. Operations internal to the FPCP that are initiated in this manner are completed within four clock cycles after the negation of $\overline{\text{DS}}$, but the main processor may initiate a subsequent asynchronous bus cycle before those internal operations are completed. In these cases, the MC68881 delays the subsequent asynchronous access by not responding to the bus cycle (and thus not asserting $\overline{\text{DSACKx}}$) until the internal operations are completed. Synchronous accesses (i.e., accesses to the response or save CIR) execute in the normal manner regardless of preceding accesses. The following is a list of the bus cycles that initiate internal operations on the negated edge of $\overline{\text{DS}}$, where a subsequent asynchronous bus cycle might overrun the FPCP and necessitate a delay in the assertion of $\overline{\text{DSACKx}}$:

1. A write cycle to the least significant byte of the control CIR
2. A write cycle to the least significant byte of the restore CIR
3. The last write cycle to the least significant byte of the operand CIR during a restore operation with a busy state frame
4. The first read from the least significant byte of the operand CIR during a save operation with an idle or busy state frame

In all of these cases, the term **least significant byte** indicates a transfer of any size that includes the least significant byte of the referenced CIR and does not indicate that only byte transfers cause conditions that require delays in subsequent bus cycles.

In addition to the cases just described, the possibility exists that the main processor may overrun the FPCP if the main processor clock frequency is greater than that of the FPCP. There are two cases where this might occur:

1. The main processor reads the operand or register select CIR before the FPCP has data ready for transfer to the main processor.
2. The main processor writes to the operand CIR before the data from the previous write cycle has been stored internally.

In both of these cases, the FPCP does not respond to the initiation of an asynchronous bus cycle until the internal data transfers are completed (synchronous bus cycles are not delayed).

As just described, the FPCP delays asynchronous bus cycles, if necessary, until internal operations are completed. However, even though the response to these bus cycles is delayed, the FPCP bus interface unit control logic does detect the beginning of each access regardless of the state of the execution unit. Thus, it is possible that an access to a CIR may be detected before the bus interface unit has completed previous operations and updated status flags to reflect the state of an instruction dialog. This can result in spurious protocol violations if the coprocessor interface protocol is not strictly observed.

The most important protocol that must be observed is that the come-again request included by the FPCP in every evaluate effective address and transfer data primitive must not be ignored by the main processor. For example, if the come-again request is ignored and the main processor clock is much faster than the FPCP clock, the following situation might occur:

1. The main processor receives the evaluate effective address and transfer data request primitive, processes it, and begins to transfer the operand.
2. The last operand part is written to the operand CIR.
3. The main processor ignores the come-again request and begins execution of the next instruction immediately.
4. The next instruction is an FPCP instruction that the main processor initiates by writing the command word to the command CIR.
5. Since the internal operand transfer is not complete, the BIU flags still indicate that the next expected access is to the operand CIR; thus the access to the command CIR is deemed illegal, and a protocol violation occurs.

In this case, if the main processor follows the protocol and services the come-again request by reading the response CIR immediately after the last operand CIR access, a null (CA = 1, IA = 1) primitive may be returned by the MC68881. Since the response CIR read-cycle timing is synchronous with the MC68881 clock signal, this read cycle allows the main processor to be synchronized to the MC68881 internal operations. Thus, the next read of the response CIR normally occurs after internal operations are completed. At that time, the response encoding is changed to null (CA = 0) to allow the main processor to proceed, and the subsequent access to the command CIR is a legal access.

In addition to the previously mentioned restrictions, the MC68882 may return an evaluate <ea> and transfer data primitive with CA = 0, which does not require the main processor to read the response register before proceeding to the next instruction. After the last write to the operand CIR, if the next instruction is another MC68882 instruction, the write to the command CIR can occur immediately without adverse effects. However, if the read of the response CIR (which normally follows the write of the command CIR) occurs sooner than three MC68882 clocks after the completion of the previous operand CIR write operation, a protocol violation occurs. Therefore, if the main processor is an MC68020 or MC68030, its clock frequency cannot be more than 1.5 times the frequency of the MC68882 clock. Otherwise, the read response CIR operation might occur too soon and cause a protocol violation. A main processor other than an MC68020 or MC68030 must also observe this timing requirement to avoid a protocol violation.

SECTION 11
INTERFACING METHODS

This section contains information about the interface logic required to connect the MC68881/MC68882 (FPCP) to an MC68020/MC68030 (MPU) as a coprocessor, or to an MC68000, MC68008, or MC68010 as a peripheral processor.

11.1 FPCP AND MPU INTERFACING

The following paragraphs describe the connecting of the FPCP to an MPU for coprocessor operation using an 8-, 16-, or 32-bit data bus.

11.1.1 32-Bit Data Bus Coprocessor Connection

Figure 11-1 illustrates the coprocessor interface connection of an FPCP to an MPU using a 32-bit data bus. The FPCP is configured to operate over a 32-bit data bus when both the A0 and $\overline{SIZE}$ pins are connected to V_{CC}.

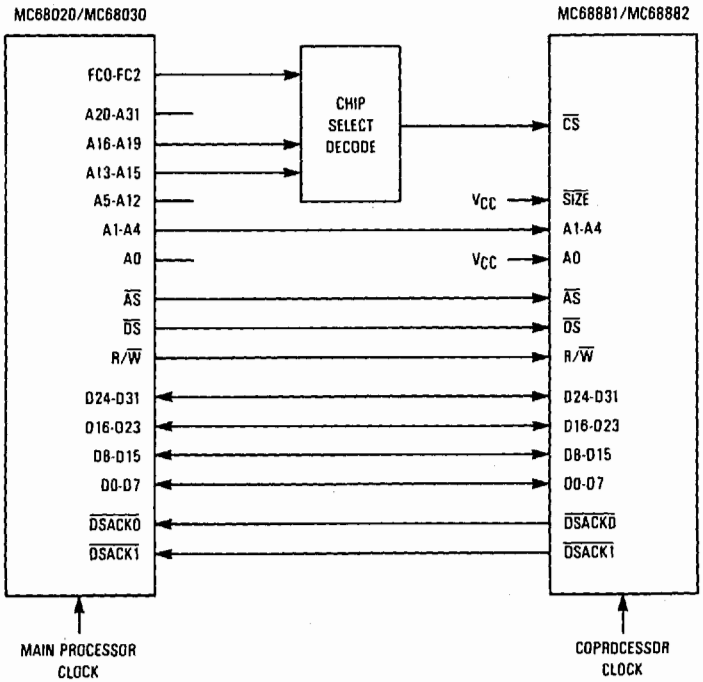


Figure 11-1. 32-Bit Data Bus Coprocessor Connection

11.1.2 16-Bit Data Bus Coprocessor Connection

Figure 11-2 illustrates the coprocessor interface connection of an FPCP to an MPU using a 16-bit data bus. The FPCP is configured to operate over a 16-bit data bus when the $\overline{SIZE}$ pin is connected to V_{CC}, and the A0 pin is connected to GND. The sixteen least significant data pins (D15–D0) must be connected to the sixteen most significant data pins (D31–D16) when the FPCP is configured to operate over a 16-bit data bus (i.e., connect D0 to D16, D1 to D17, . . . and D15 to D31). The $\overline{DSACKx}$ pins of the two devices are directly connected, although it is not necessary to connect the $\overline{DSACK0}$ pin since the FPCP never asserts it in this configuration.

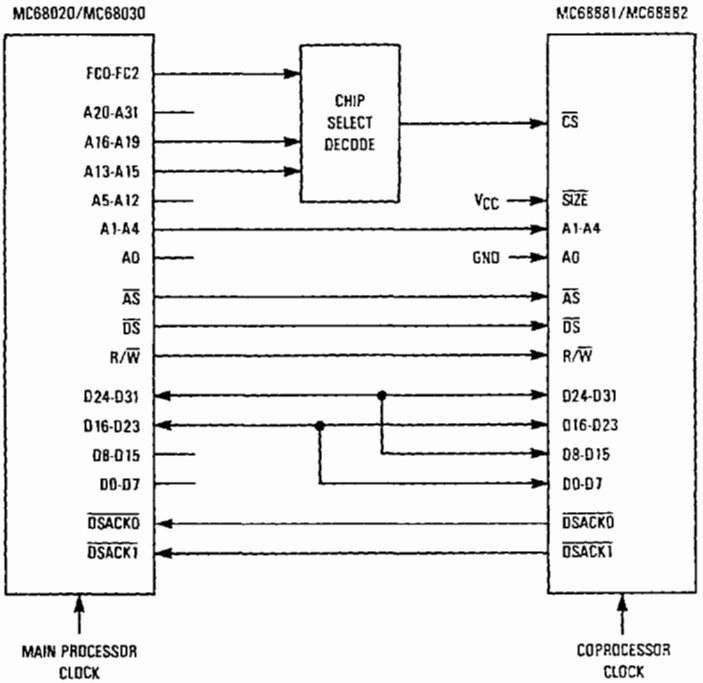


Figure 11-2. 16-Bit Data Bus Coprocessor Connection

11.1.3 8-Bit Data Bus Coprocessor Connection

Figure 11-3 illustrates the connection of an FPCP to an MPU as a coprocessor over an 8-bit data bus. The FPCP is configured to operate over an 8-bit data bus when the $\overline{SIZE}$ pin is connected to GND. The 24 least significant data pins (D23–D0) must be connected to the eight most significant data pins (D31–D224) when the FPCP is configured to operate over an 8-bit data bus (i.e., connect D0 to D8, D16 and D24; D1 to D9, D17, and D25; . . . and D7 to D15, D23, and D31). The $\overline{DSACKx}$ pins of the two devices are directly connected, although it is not necessary to connect the $\overline{DSACK1}$ pin since the FPCP never asserts it in this configuration.

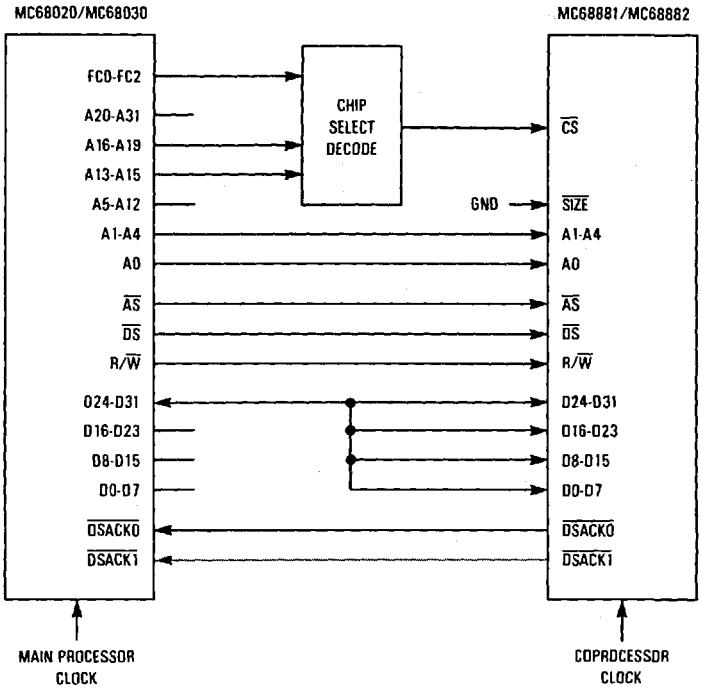


Figure 11-3. 8-Bit Data Bus Coprocessor Connection

11.2 INTERFACING THE FPCP AS A PERIPHERAL

The following paragraphs describe the connecting of the FPCP to an MC68000, MC68008, or MC68010 processor for operation as a peripheral using an 8- or 16-bit data bus.

11

11.2.1 16-Bit Data Bus Peripheral Processor Connection

Figure 11-4 illustrates the connection of an FPCP to an MC68000 or MC68010 as a peripheral processor over a 16-bit data bus. The FPCP is configured to operate over a 16-bit data bus when the $\overline{SIZE}$ pin is connected to V_{CC}, and the A0 pin is connected to GND. The 16 least significant data pins (D15–D0) must be connected to the 16 most significant data pins (D31–D16) when the FPCP is configured to operate over a 16-bit data bus (i.e., connect D0 to D16, D1 to D17, . . . and D15 to D31). The $\overline{DSACK1}$ pin of the FPCP is connected to the $\overline{DTACK}$ pin of the main processor, and the $\overline{DSACK0}$ pin is not used.

When connected as a peripheral processor, the FPCP chip select ($\overline{CS}$) decode is system dependent. If the MC68000 is used as the main processor, the FPCP $\overline{CS}$ must be decoded in the supervisor or user data spaces. However, if the MC68010 is used for the main processor, the MOVES instruction can be used to emulate any CPU space access that the MPU generates for coprocessor communications. Thus, the $\overline{CS}$ decode logic for such systems may be the same as in an MC68020 or MC68030 system; that is, the FPCP does not use any part of the data address spaces.

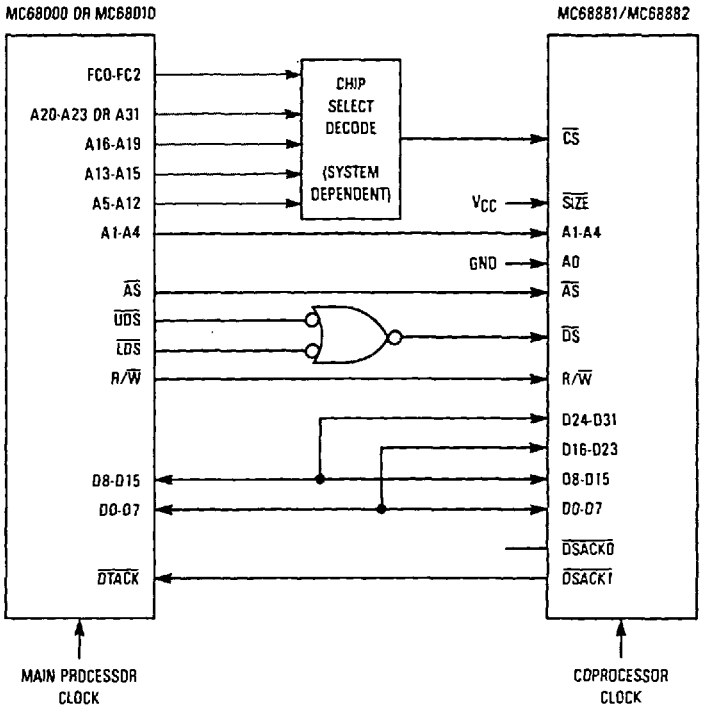


Figure 11-4. 16-Bit Data Bus Peripheral Processor Connection

11.2.2 8-Bit Data Bus Peripheral Processor Connection

Figure 11-5 illustrates the connection of an FPCP to an MC68008 as a peripheral processor over an 8-bit data bus. The FPCP is configured to operate over an 8-bit data bus (i.e., connect D0 to D8, D16, and D24; D1 to D9, D17, and D25; . . . and D7 to D15, D23, and D31). The $\overline{\text{DSACK0}}$ pin of the FPCP is connected to the $\overline{\text{DTACK}}$ pin of the MC68008, and the $\overline{\text{DSACK1}}$ pin is not used.

11

When connected as a peripheral processor, the FPCP chip-select ($\overline{\text{CS}}$) decode is system dependent, and the $\overline{\text{CS}}$ must be decoded in the supervisor or user data spaces.

11.3 PERIPHERAL PROCESSOR OPERATION

The FPCP can be used as a peripheral processor on systems where the main processor does not have a coprocessor interface by using instruction sequences that emulate the protocol of the coprocessor interface. When an FPCP instruction is encountered by an MC68000, MC68008, or MC68010, the instruction causes an F-line emulator trap to be taken. The trap handler then emulates the coprocessor interface protocol. Refer to **SECTION 7 COPROCESSOR INTERFACE** for details of the communications protocol.

The FPCP requests services from the main processor via coprocessor interface response register primitives. **SECTION 7 COPROCESSOR INTERFACE** describes the main processor service requests required for the execution of each FPCP instruction type. Also included in **SECTION 7 COPROCESSOR INTERFACE** is a summary of all FPCP response primitives.

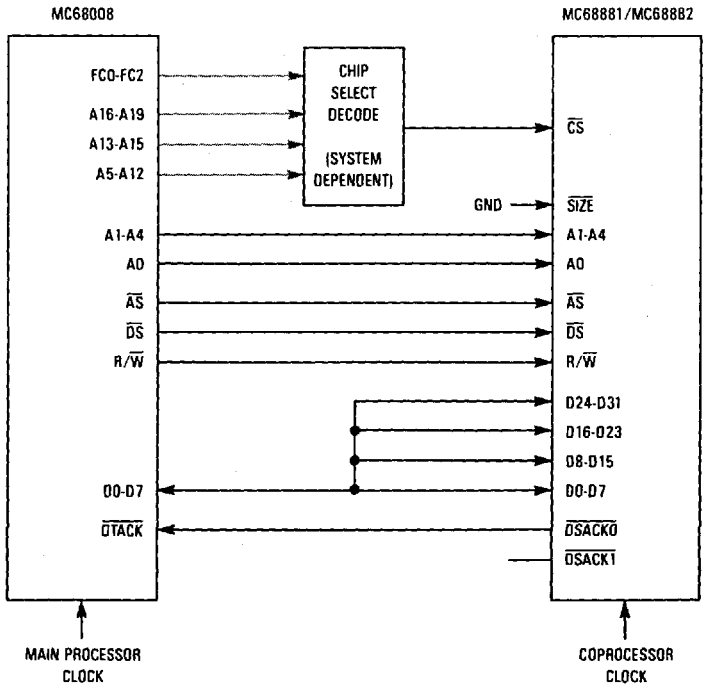


Figure 11-5. 8-Bit Data Bus Peripheral Processor Connection

SECTION 12
ELECTRICAL SPECIFICATIONS

This section contains electrical specifications and associated timing information for the MC68881/MC68882 (FPCP).

12.1 MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.5 to +7.0	V
Operating Temperature	T_A	0 to 70	°C
Storage Temperature	T_{stg}	−55 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum-rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either GND or V_{CC}).

12.2 THERMAL CHARACTERISTICS — PGA PACKAGE

Characteristic	Symbol	Value	Rating
Thermal Resistance — Ceramic			°C/W
Junction to Ambient	θ_{JA}	30*	
Junction to Case	θ_{JC}	15*	

*Estimated

12.3 POWER CONSIDERATIONS

The average chip-junction temperature, T_J, in °C can be obtained from:

$$T_J = T_A + (P_D \cdot \theta_{JA}) \quad (1)$$

where:

T_A = Ambient Temperature, °C
θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, °C/W
P_D = $P_{INT} + P_{I/O}$
P_{INT} = $I_{CC} \times V_{CC}$, Watts — Chip Internal Power
$P_{I/O}$ = Power Dissipation on Input and Output Pins — User Determined

For most applications $P_{I/O} < P_{INT}$ and can be neglected.

The following is an approximate relationship between P_D and T_J (if $P_{I/O}$ is neglected):

$$P_D = K \div (T_J + 273°C) \quad (2)$$

Solving equations (1) and (2) for K gives:

$$K = P_D \cdot (T_A + 273°C) + \theta_{JA} \cdot P_D^2 \quad (3)$$

where K is a constant pertaining to the particular part. K can be determined from equation (3) by measuring P_D (at equilibrium) for a known T_A. Using this value of K, the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A.

The total thermal resistance of a package (θ_{JA}) can be separated into two components, θ_{JC} and θ_{CA}, representing the barrier to heat flow from the semiconductor junction to the package (case) surface (θ_{JC}) and from the case to the outside ambient (θ_{CA}). These terms are related by the equation:

$$\theta_{JA} = \theta_{JC} + \theta_{CA} \quad (4)$$

θ_{JC} is device related and cannot be influenced by the user. However, θ_{CA} is user dependent and can be minimized by such thermal management techniques as heat sinks, ambient air cooling, and thermal convention. Thus, good thermal management on the part of the user can significantly reduce θ_{CA} so that θ_{JA} approximately equals θ_{JC}. Substitution of θ_{JC} for θ_{JA} in equation (1) will result in a lower semiconductor junction temperature.

Values for thermal resistance presented in this document, unless estimated, were derived using the procedure described in **Freescale** Reliability Report 7843, "Thermal Resistance Measurement Method for MC68XX Microcomponent Devices," and are provided for design purposes only. Thermal measurements are complex and dependent on procedure and setup. User derived values for thermal resistance may differ.

12.4 DC ELECTRICAL CHARACTERISTICS

(V_{CC} = 5.0 Vdc ± 5%; GND = 0 Vdc, T_A = 0°C to 70°C)

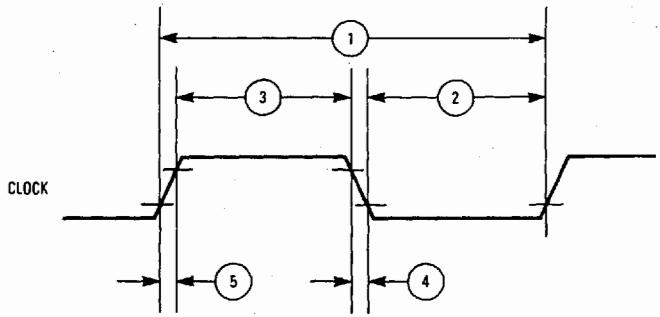
Characteristic		Symbol	Min	Max	Unit
Input High Voltage		V_{IH}	2.0	V_{CC}	V
Input Low Voltage		V_{IL}	GND −0.5	0.8	V
Input Leakage Current @ 5.25 V	CLK, $\overline{RESET}$, R/$\overline{W}$, A0-A4, $\overline{CS}$, $\overline{DS}$, $\overline{AS}$, $\overline{SIZE}$	I_{in}	—	10	μA
Hi-Z (Off State) Input Current @ 2.4 V/0.4 V	$\overline{DSACK0}$, $\overline{DSACK1}$, D0-D31	I_{TSI}	—	20	μA
Output High Voltage (I_{OH} = −400 μA)	$\overline{DSACK0}$, $\overline{DSACK1}$, D0-D31	V_{OH}	2.4	—	V
Output Low Voltage (I_{OL} = 5.3 mA)	$\overline{DSACK0}$, $\overline{DSACK1}$, D0-D31	V_{OL}	—	0.5	V
Output Low Current (V_{OL} = GND)	$\overline{SENSE}$	I_{OL}	—	500	μA
Power Dissipation		P_D	—	0.75	W
Capacitance* (V_{in} = 0, T_A = 25°C, f = 1 MHz)		C_{in}	—	20	pF
Output Load Capacitance		C_L	—	130	pF

*Capacitance is periodically sampled rather than 100% tested.

12.5 AC ELECTRICAL CHARACTERISTICS — CLOCK INPUT

(V_{CC} = 5.0 Vdc ± 5%; GND = 0 Vdc, T_A = 0°C to 70°C) (see Figure 12-1)

Num	Characteristic	16.67 MHz		20 MHz		25 MHz		33.33 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
	Frequency of Operation	8	16.67	12.5	20	12.5	25	16.7	33.33	MHz
1	Cycle Time	60	125	50	80	40	80	30	60	ns
2,3	Clock Pulse Width (Measured from 1.5 V to 1.5 V for 33 MHz)	24	95	20	54	15	59	14	66	ns
4,5	Rise and Fall Times	—	5	—	5	—	4	—	3	ns



NOTE:

1. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through, the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

Figure 12-1. Clock Input Timing Diagram

12.6 AC ELECTRICAL CHARACTERISTICS — READ AND WRITE CYCLES

(V_{CC} = 5.0 Vdc ± 5%; GND = 0 Vdc, T_A = 0°C to 70°C) (see Figures 12-2 through 12-4)

Num	Characteristic	16.67 HMz		20 MHz		25 MHz		33.33 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
6[5]	Address Valid to $\overline{AS}$ Asserted	15	—	10	—	5	—	5	—	ns
6A[5]	Address Valid to $\overline{DS}$ Asserted (Read)	15	—	10	—	5	—	5	—	ns
6B[5]	Address Valid to $\overline{DS}$ Asserted (Write)	50	—	50	—	35	—	26	—	ns
7[6]	$\overline{AS}$ Negated to Address Invalid	10	—	10	—	5	—	5	—	ns
7A[6]	$\overline{DS}$ Negated to Address Invalid	10	—	10	—	5	—	5	—	ns
8[9]	$\overline{CS}$ Negated to $\overline{AS}$ Asserted	0	—	0	—	0	—	0	—	ns
8A[9]	$\overline{CS}$ Negated to $\overline{DS}$ Asserted (Read)	0	—	0	—	0	—	0	—	ns
8B	$\overline{CS}$ Asserted to $\overline{DS}$ Asserted (Write)	30	—	25	—	20	—	15	—	ns
9	$\overline{AS}$ Negated to $\overline{CS}$ Negated	10	—	10	—	5	—	5	—	ns
9A	$\overline{DS}$ Negated to $\overline{CS}$ Negated	10	—	10	—	5	—	5	—	ns
10	R/$\overline{W}$ High to $\overline{AS}$ Asserted (Read)	15	—	10	—	5	—	5	—	ns
10A	R/$\overline{W}$ High to $\overline{DS}$ Asserted (Read)	15	—	10	—	5	—	5	—	ns
10B	R/$\overline{W}$ Low to $\overline{DS}$ Asserted (Write)	35	—	30	—	25	—	25	—	ns
11	$\overline{AS}$ Negated to R/$\overline{W}$ Low (Read) or $\overline{AS}$ Negated to R/$\overline{W}$ High (Write)	10	—	10	—	5	—	5	—	ns
11A	$\overline{DS}$ Negated to R/$\overline{W}$ Low (Read) or $\overline{DS}$ Negated to R/$\overline{W}$ High (Write)	10	—	10	—	5	—	5	—	ns
12	$\overline{DS}$ Width Asserted (Write)	40	—	38	—	30	—	23	—	ns
13	$\overline{DS}$ Width Negated	40	—	38	—	30	—	23	—	ns
13A[4]	$\overline{DS}$ Negated to $\overline{AS}$ Asserted	30	—	30	—	25	—	18	—	ns
14[2]	$\overline{CS}$, $\overline{DS}$ Asserted to Data-Out Valid) (Read)	—	80	—	45	—	45	—	30	ns
15	$\overline{DS}$ Negated to Data-Out Invalid (Read)	0	—	0	—	0	—	0	—	ns
16	$\overline{DS}$ Negated to Data-Out High Impedance (Read)	—	50	—	30	—	30	—	20	ns
17	Data-In Valid to $\overline{DS}$ Asserted (Write)	15	—	10	—	5	—	5	—	ns
18	$\overline{DS}$ Negated to Data-In Invalid (Write)	15	—	10	—	5	—	5	—	ns
19[2]	$\overline{START}$ True to $\overline{DSACK0}$ and $\overline{DSACK1}$ Asserted	—	50	—	35	—	25	—	20	ns
19A[7]	$\overline{DSACK0}$ Asserted to $\overline{DSACK1}$ Asserted (Skew)	−15	15	−10	10	−10	10	—	5	ns
20	$\overline{DSACK0}$ or $\overline{DSACK1}$ Asserted to Data-Out Valid	—	50	—	43	—	32	—	17	ns
21[8]	$\overline{START}$ False to $\overline{DSACK0}$ and $\overline{DSACK1}$ Negated	—	50	—	30	—	30	—	20	ns
22[8]	$\overline{START}$ False to $\overline{DSACK0}$ and $\overline{DSACK1}$ High Impedance	—	70	—	40	—	40	—	30	ns

- Continued -

12.6 AC ELECTRICAL CHARACTERISTICS — READ AND WRITE CYCLES (Continued)

Num	Characteristic	16.67 HMz		20 MHz		25 MHz		33.33 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
23[3,8]	$\overline{START}$ True to Clock High (Synchronous Read)	0	—	0	—	0	—	0	—	ns
24[3]	Clock Low to Data-Out Valid (Synchronous Read)	—	105	—	80	—	60	—	45	ns
25[3,8]	$\overline{START}$ True to Data-Out Valid (Synchronous Read)	— 1.5	105+ 2.5	— 1.5	80+ 2.5	— 1.5	60+ 2.5	— 1.5	45+ 2.5	ns Clks
26[3]	Clock Low to $\overline{DSACK0}$ and $\overline{DSACK1}$ Asserted (Synchronous Read)	—	75	—	55	—	45	—	30	ns
27[3,8]	$\overline{START}$ True to $\overline{DSACK0}$ and $\overline{DSACK1}$ Asserted (Synchronous Read)	— 1.5	75+ 2.5	— 1.5	55+ 2.5	— 1.5	45+ 2.5	— 1.5	30+ 2.5	ns Clks

NOTES:

1. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through, the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.
2. These specifications only apply if the MC68882 has completed all internal operations initiated by the termination of the previous bus cycle when $\overline{DS}$ was negated.
3. Synchronous read cycles occur ***only*** when the save or response CIR locations are read.
4. This specification only applies to systems in which back-to-back accesses (read-write or write-write) of the operand CIR can occur. When the MC68882 is used as a coprocessor to the MC68020/MC68030, this can occur when the addressing mode is Immediate.
5. If the $\overline{SIZE}$ pin is ***not*** strapped to either V_{CC} or GND, it must have the same setup times as do addresses.
6. If the $\overline{SIZE}$ pin is ***not*** strapped to either V_{CC} or GND, it must have the same hold times as do addresses.
7. This number is reduced to 5 nanoseconds if $\overline{DSACK0}$ and $\overline{DSACK1}$ have equal loads.
8. START is not an external signal; rather, it is the logical condition that indicates the start of an access. The logical equation for this condition is $\overline{START} = \overline{CS} + \overline{AS} + R/\overline{W} \cdot \overline{DS}$.
9. If a subsequent access is not a FPCP access, $\overline{CS}$ must be negated before the assertion of $\overline{AS}$ and/or $\overline{DS}$ on the non-FPCP access. These specifications replace the old specifications 8 and 8A (the old specifications implied that in all cases, transitions in $\overline{CS}$ must not occur simultaneously with transitions of $\overline{AS}$ or $\overline{DS}$. This is not a requirement of the /MC68882).

Timing diagrams (Figures 12-2, 12-3, and 12-4) are located on foldout pages at the end of this document.

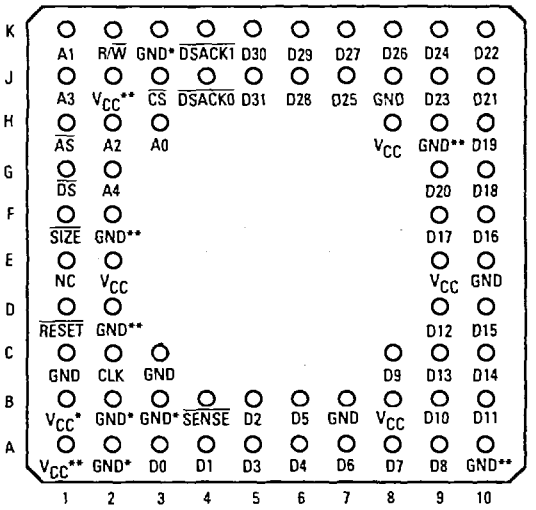
SECTION 13
ORDERING INFORMATION AND MECHANICAL DATA

This section contains the pin assignments and package dimensions of the MC68881/MC68882 (FPCP). In addition, detailed information is provided to be used as a guide when ordering.

13.1 STANDARD MC68881/MC68882 ORDERING INFORMATION

Package Type	Frequency (MHz)	Temperature	Order Number
Pin Grid Array RC Suffix	12.5	0°C to 70°C	MC68881RC12
		−40°C to +85°C	MC68881LRC12
		−55°C to +125°C	MC68881ERC12
	16.67	0°C to 70°C	MC68881RC16
		−40°C to +85°C	MC68881LRC16
		−55°C to +125°C	MC68881ERC16
	20	0°C to 70°C	MC68881RC20
		−40°C to +85°C	MC68881LRC20
		−55°C to +125°C	MC68881ERC20
	25	0°C to 70°C	MC68881RC25
Pin Grid Array RC Suffix	33.33	0°C tp 70°C	MC68882RC33
	16.67	0°C to 70°C	MC68882RC16
	20	0°C to 70°C	MC68882RC20
	25	0°C to 70°C	MC68882RC25

68-PIN GRID ARRAY



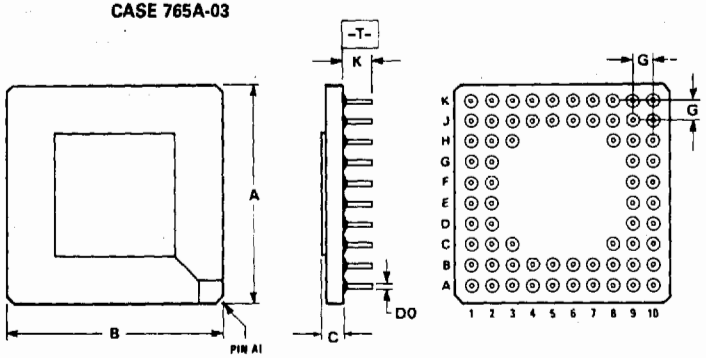
Pin Group	V_{CC}	GND
D31-D16	H8	J8
D15-D00	B8	B7
Internal Logic, $\overline{DSACK1}$, $\overline{DSACK0}$	E2, E9	A2, B2, B3, B4***, C3, E10, K3
Separate	—	C1
Extra	A1, B1, J2	A10, D2, F2, H9

*New assignment for the A93N mask.
**Reserved for future Freescale use.
***$\overline{SENSE}$ pin, may be used as an additional GND pin.

13

13.3 PACKAGE DIMENSIONS

RC SUFFIX
PIN GRID ARRAY
CASE 765A-03



NOTES:
1. DIMENSIONS A AND B ARE DATUMS AND T IS DATUM SURFACE.
2. POSITIONAL TOLERANCE FOR LEADS (68 PLACES)
   ⌖ ⌀ 0.13 (.005) Ⓜ T A Ⓢ B Ⓢ
3. DIMENSIONING AND TOLERANCING PER ANSI Y14.5M, 1982.
4. CONTROLLING DIMENSION: INCH.

	MILLIMETERS		INCHES	
DIM	MIN	MAX	MIN	MAX
A	26.67	27.17	1.050	1.070
B	26.67	27.17	1.050	1.070
C	1.91	2.66	0.075	0.105
D	0.43	0.60	0.017	0.024
G	2.54 BSC		0.100 BSC	
K	4.32	4.82	0.170	0.190

13

APPENDIX A
GLOSSARY

ALGORITHM

A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.

BCD (Binary-Coded-Decimal) Number

The representation of cardinal numbers 0 through 9 by 10 binary codes of any length. The minimum length is four and there are over 29×10^9 possible four-bit BCD codes. However, codes in which the four bits contain the hexadecimal representation of 0 through 9 are the more commonly used codes, for obvious reasons.

BIASED EXPONENT

The sum of the exponent and a constant (bias) chosen to make the biased exponents range non-negative.

BINARY FLOATING-POINT NUMBER

A bit string characterized by three components: a sign, a signed exponent, and a significand. (See single, double, and extended precision.) The numerical value of the bit string is the signed product of the significand and two raised to the power of the exponent.

DENORMALIZED NUMBER

A floating-point number having all zeros in the exponent and a non-zero value in the fraction/mantissa.

DOUBLE PRECISION

A 64-bit binary floating-point operand format composed of three fields: a one-bit sign field, an 11-bit biased exponent field, and an 52-bit fraction (significand) field.

DYADIC OPERATION

An operation on two operands.

E FIELD

See exponent (E field).

EXPONENT

A symbol written above and to the right of a mathematical expression to indicate the operation of rising to a power.

EXPONENT (E FIELD)

The component of a binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. Occasionally, the exponent is called the signed or unbiased exponent.

EXTENDED PRECISION
A 96-bit binary floating-point operand format composed of four fields: a one-bit sign field, an 15-bit biased exponent field, a 16-bit undefined field, and a 64-bit mantissa (significand) field.

F FIELD
See fraction (F field).

FIXED-POINT
Pertaining to a numeration system in which the position of the radix point is fixed with respect to one end of the numerals, according to some convention. The integer data types used by the FPCP are fixed-point numbers.

FLOATING-POINT
Pertaining to a system in which the location of the radix point does not remain fixed with respect to one end of the numerical expressions, but is regularly recalculated. The location of the point is usually given by expressing a power of the base (or radix). The single, double, and extended precision data types used by the FPCP are floating-point numbers.

FRACTION (F FIELD)
The field of the significand that lies to the right of its implied binary point.

INTEGER
Any of the natural numbers, the negatives of these numbers, or zero.

MANTISSA
Mantissa and significand are interchangeable throughout this manual. See significand.

MODULO
A mathematical operation that yields the remainder of division. Thus, 39 modulo 6 = 3.

MONADIC OPERATION
An operation on one operand, for example, negation.

NAN (Not-A-Number)
A symbolic entity encoded in floating-point format. There are two types of NANs; signaling and quiet. Signaling NANs signal the valid operation exception whenever appearing as operands. Quiet NANs propagate through almost every arithmetic operation without signaling exceptions.

NORMALIZE
To convert a floating-point number to one whose significand consists of an integer bit of 1.

A

OPERAND
That which is, or is to be operated upon. An operand is usually identified by an address field of an instruction.

ORTHOGONAL
Statistically independent.

S BIT
See sign bit (S field).

S FIELD
See sign bit (S field).

SIGN BIT (S FIELD)
Denotes the sign of the operand: zero for positive and one for negative. Floating-point numbers are in sign-magnitude form, which means that only the S bit is complemented to change the sign of the represented number.

SIGNIFICAND
The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of the implied binary point and a fraction field to the right of the implied binary point. Significand and mantissa are interchangeable throughout this manual. See fraction (F field).

SINGLE PRECISION
A 32-bit binary floating-point operand format composed of three fields: a one-bit sign field, an 8-bit biased exponent field, and a 23-bit fraction (significand) field.

TRANSCENDENTAL
Being, involving, or representing a function (sine x, log x) that cannot be expressed by a finite number of algebraic operations.

UNNORMALIZED NUMBER
An extended precision external operand that contains an explicit integer part bit (j) of zero and an exponent that is neither the maximum nor the minimum for the format.

APPENDIX B
ABBREVIATIONS AND ACRONYMS

A	Address
Abs	Absolute
AEXC	Accrued exception
ALU	Arithmetic logic unit
APU	Arithmetic processing unit
AS	Address strobe
B	Byte integer
BCD	Binary coded decimal
BIU	Bus interface unit
BSUN	Branch/set on unordered
cc	Condition code
CLK	Clock
CMP	Compare
cp	Coprocessor
CPU	Central processor unit
CPRED	Conditional predicate
CS	Chip select
CU	Conversion unit
d	Displacement
D	Data
D	Double precision binary real floating-point
DIV	Divide
DMA	Direct memory access
DS	Data strobe
DSACK	Data and size acknowledge
DZ	Divide by zero
e	Exponent
ea	Effective address
EQ	Equal
EXC	Exception
EXP	Exponent
ENAB	Enable

f	Fraction
F	False
F	Floating-point
FB	Floating-point branch
FBEQ	Floating-point branch equal
FBGT	Floating-point branch greater than
FBLE	Floating-point branch less than or equal
FBNEQ	Floating-point branch not equal
FBNGT	Floating-point branch not greater than
FDB	Floating-point decrement and branch
FDBEQ	Floating-point decrement and branch equal
FDBGT	Floating-point decrement and branch greater than
FDBLE	Floating-point decrement and branch less than or equal
FDBNEQ	Floating-point decrement and branch not equal
FDBNGT	Floating-point decrement and branch not greater than
FP	Floating-point
FPCC	Floating-point condition code
FPCP	MC68881/MC68882 coprocessor
FS	Floating-point set
FSEQ	Floating-point set equal
FSGT	Floating-point set on greater than
FSLE	Floating-point set on less than or equal
FSNEQ	Floating-point set not equal
FSNGT	Floating-point set on not greater than
FT	Floating-point trap on
FTEQ	Floating-point trap equal
FTGT	Floating-point trap greater than
FTLE	Floating-point trap less than or equal
FTNEQ	Floating-point trap not equal
FTNGT	Floating-point trap not greater than
FTP	Floating-point trap on parameter
GE	Greater than or equal
GLE	Greater or less or equal
GND	Ground
GT	Greater than
I/O	Input/output
I	Infinity
IADDR	Instruction address
ID	Identification
Imm	Immediate
INEX	Inexact
INEX1	Inexact arithmetic
INEX2	Inexact conversion
IOP	Invalid operation
j	Integer part

L	Long word integer
LE	Less than or equal
LSB	Least significant bit/byte
LT	Less than
MANT	Mantissa
MOD	Modulo
MPU	Main processing unit
MPU	MC68020/MC68030 processor
MSB	Most significant bit/byte
MUL	Multiply
n	Number
N	Negative
NAN	Not-a-number
NEQ	Not equal
NGE	Not greater than or equal
NGL	Not greater or less than
NGLE	Not greater or less or equal
NGT	Not greater than
NLE	Not less than or equal
NLT	Not less than
OGE	Ordered greater than or equal
OGL	Ordered greater or less than
OGT	Ordered greater than
OLE	Ordered less than or equal
OLT	Ordered less than
OPERR	Operand error
OR	Ordered
OVFL	Overflow
P	Packed binary coded decimal real string
PC	Program counter
QUOT	Quotient
R/W	Read/write
REM	Remainder
RM	Round toward minus infinity
RN	Round to nearest
RP	Round toward plus infinity
RZ	Round toward zero

s	Sign
S	Single precision binary real floating-point
SE	Sign of exponent
SEQ	Signaling equal
SF	Signaling false
SGL	Single
SM	Sign of mantissa
SNAN	Signaling NAN
SNEQ	Signaling not equal
SOC	Set on condition
ST	Signaling true
SUB	Subtract
T	Trap
T	True
TTL	Transistor-transistor logic
UEQ	Unordered or equal
UGE	Unordered or greater or equal
UGT	Unordered or greater
ULE	Unordered or less or equal
ULT	Unordered or less than
UN	Unordered
UNFL	Underflow
W	Word integer
x	Don't care, irrelevant
X	Extended precision binary real floating-point
Z	Zero
$	Hexadecimal
+inf	Positive infinity
−inf	Negative infinity

INDEX

I

— C —

I

— C —

— D —

— E —

I

— E —

— F —

I

— F —

— G —

— H —

I

I

— M —

— N —

— O —

— P —

I

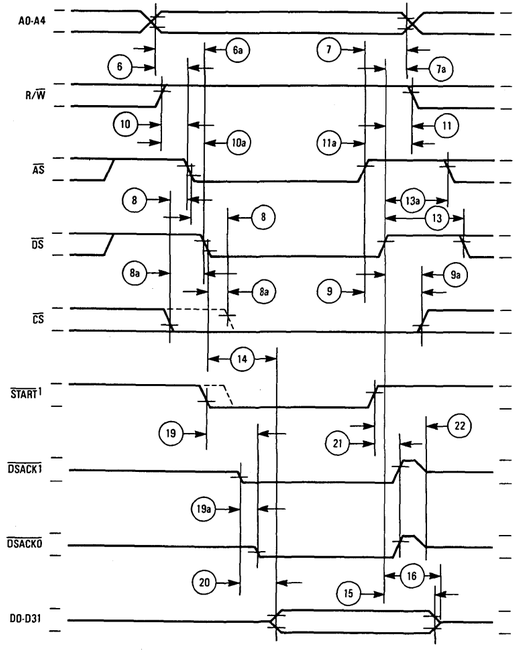
I

— S —

— T —

I

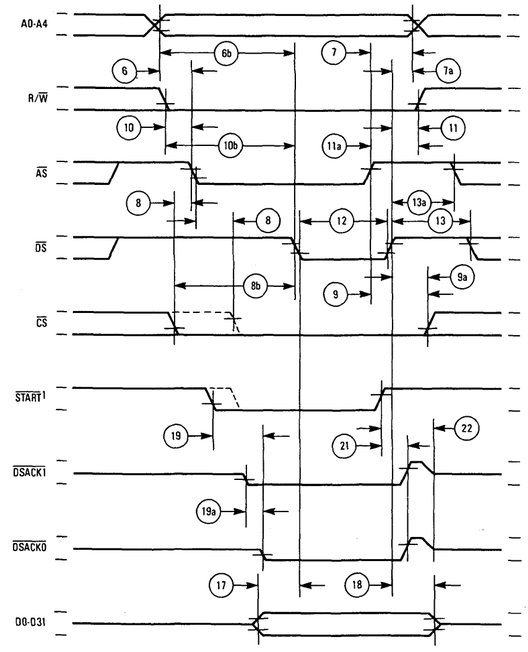
I



NOTES:

1. $\overline{START}$ is actually a logical condition, but is shown as an active low signal for clarity. The logical equation for this signal is $\overline{START} = \overline{CS} + \overline{AS} + (R/\overline{W} \cdot \overline{DS})$.
2. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through, the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

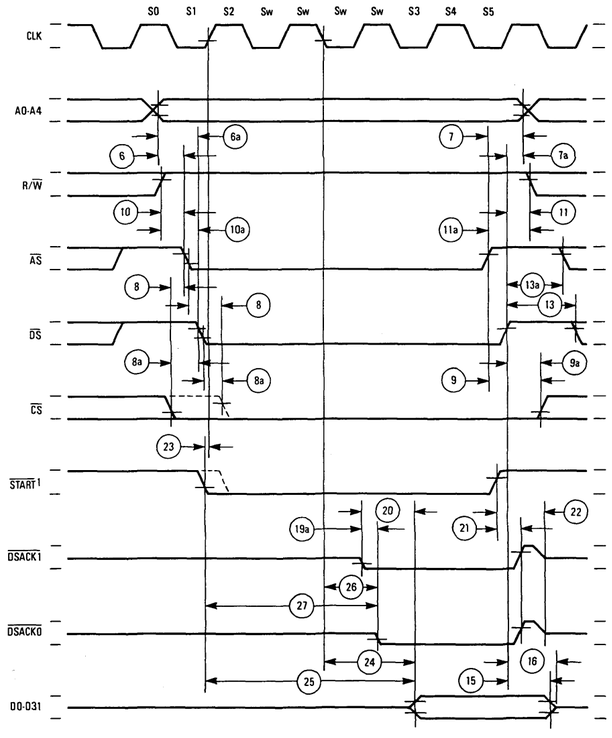
Figure 12-2. Asynchronous Read Cycle Timing Diagram



NOTES:

1. $\overline{START}$ is actually a logical condition, but is shown as an active low signal for clarity. The logical equation for this signal is $\overline{START} = \overline{CS} + \overline{AS} + (R/\overline{W} \cdot \overline{DS})$.
2. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through, the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

Figure 12-3. Asynchronous Write Cycle Timing Diagram



NOTES:

1. $\overline{START}$ is actually a logical condition, but is shown as an active low signal for clarity. The logical equation for this signal is $\overline{START} = \overline{CS} + \overline{AS} + (R/\overline{W} \cdot \overline{DS})$.
2. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through, the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

Figure 12-4. Synchronous Read Cycle Timing Diagram

Timing Diagrams

FREESCALE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 7650 AUSTIN, TX

POSTAGE WILL BE PAID BY ADDRESSEE

FREESCALE
MICROPROCESSOR & MEMORY TECHNOLOGIES GROUP
M68000 TECHNICAL PUBLICATIONS GROUP (OE33)
6501 WILLIAM CANNON DR W
AUSTIN TX 78735-9833

Tear Here

Read Other Side!

Freescale 68000 Family Document Card

Company Name _____

Your Name _____

Title _____

Mailing Address _____ Mail Drop _____

City _____ State _____ Zip _____

Telephone ( ) _____ **Fax No.** _____

M68000 Part Number(s) _____

(Located at Top Corner of Document) _____

_____

FREESCALE

Please send your own input or suggestions to Fax No. **(512) 891-8593**

Tear Here

Attention !

*This card will be used to issue real-time updates and revisions to your 68000 family document(s). By including your **fax number**, immediate updates and revisions can be sent to you as they occur. Your documentation never needs to be out of date again. If the card is not returned to Freescale, we have **no** way of sending updates and revisions to you.*

(This form may also be used to let us know about address / fax # changes).

This material will be sent from the Austin Oak Hill site. Non-US recipients of this book, please insert this card in an envelope and send to the address on the front of the card.

1ATX31094-2 Printed in USA 12/93 BANTA CO. MOTO 11 3,000 MPU YGABAA