MOTOROLA

# MC68A39 (1.5 MHz) MC68B39 (2.0 MHz)

## Advance Information

### FLOATING-POINT ROM

The MC6839 standard product ROM provides floating point capability for the MC6809 or MC6809E MPU. The MC6839 implements the entire *IEEE Proposed Standard for Binary Floating Point Arithmetic Draft 8.0,* providing a simple, economical and reliable solution to a wide variety of numerical applications. The single- and double-precision formats provide results which are bit-for-bit reproducible across all Draft 6.0 implementations, while the extended format provides the extra precision needed for the intermediate results of long calculations, in particular the implementation of transcendental functions and interest calculations. All applications benefit from extensive error-checking and well-defined responses to exceptions, which are strengths of the IEEE proposed standard.

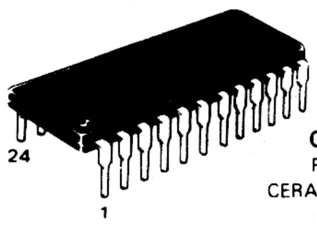
The MC6839 takes full advantage of the advanced architectural features of the MC6809 microprocessor. It is position-independent and re-entrant, facilitating its use in real-time, multi-tasking systems.

- Totally Position Independent
  - Operates in any Contiguous 8K Block of Memory
- Re-Entrant
  - No Use of Absolute RAM
  - All Memory References are made Relative to the Stack Pointer
- Flexible User Interface
  - Operands are Passed to the Package by One of Two Methods
    1) Machine Registers are used as Pointers to the Operands
    2) The Operands are Pushed onto the Hardware Stack
  - The Latter Method Facilitates the use of the MC6839 in High-Level Language Implementations
- Easy to Use Two/Three Address Architecture
  - The User Specifies Addresses of Operands and Result and Need Not be Concerned with any Internal Registers or Intermediate Results
- A Complete Implementation of the Proposed IEEE Standard Draft 6.0
  - Includes All Precisions, Modes, and Operations Required or Suggested by the Standard
  - Single, Double, and Extended Formats
  - Includes the Following Operations:
    - Add
    - Subtract
    - Multiply
    - Divide
    - Remainder
    - Square Root
    - Integer Part
    - Absolute Value
    - Negate
    - Predicate Compares
    - Condition Code Compares
    - Convert Integer ↔ Floating Point
    - Convert Binary Floating Point ↔ Decimal String

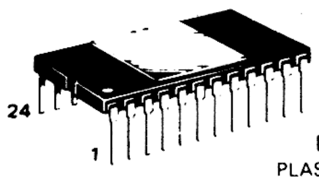
**MOS**

**(N-CHANNEL, SILICON-GATE)**

**FLOATING-POINT READ-ONLY MEMORY**

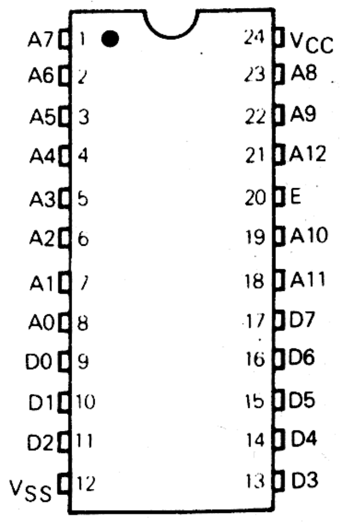


**C SUFFIX**
FRIT-SEAL
CERAMIC PACKAGE
CASE 716

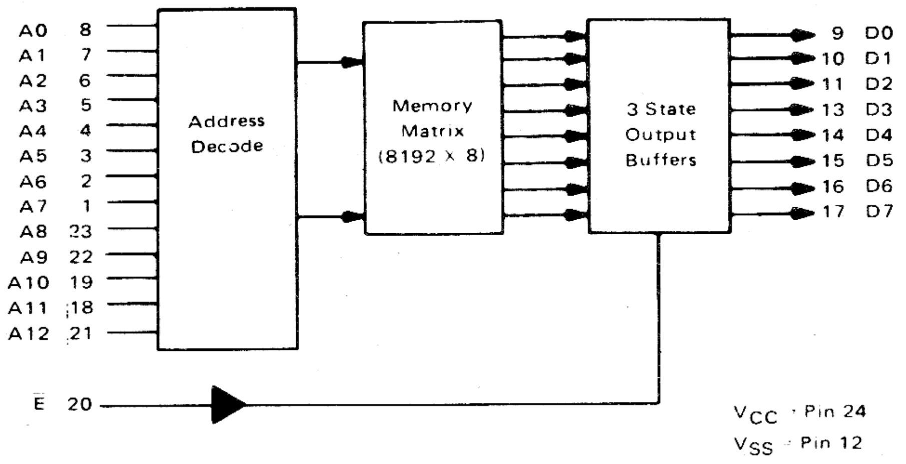


**P SUFFIX**
PLASTIC PACKAGE
CASE 709

**PIN ASSIGNMENT**



**BLOCK DIAGRAM**



## ABSOLUTE MAXIMUM RATINGS

| Rating | Symbol | Value | Unit |
|---|---|---|---|
| Supply Voltage | $V_{CC}$ | −0.5 to +7.0 | V |
| Input Voltage | $V_{in}$ | −0.5 to +7.0 | V |
| Operating Temperature Range | $T_A$ | 0 to +70 | °C |
| Storage Temperature Range | $T_{stg}$ | −65 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 (e.g., either $V_{SS}$ or $V_{CC}$).

## CAPACITANCE

($f = 1.0$ MHz, $T_A = 25$°C, periodically sampled rather than 100% tested)

| Characteristic | Symbol | Max | Unit |
|---|---|---|---|
| Input Capacitance | $C_{in}$ | 8 | pF |
| Output Capacitance | $C_{out}$ | 15 | pF |

## DC OPERATING CONDITIONS AND CHARACTERISTICS

(Full operating voltage and temperature range unless otherwise noted)

### RECOMMENDED DC OPERATING CONDITIONS

| Parameter | Symbol | Min | Nom | Max | Unit |
|---|---|---|---|---|---|
| Supply Voltage ($V_{CC}$ must be applied at least 100 μs before proper device operation is achieved) | $V_{CC}$ | 4.5 | 5.0 | 5.5 | V |
| Input High Voltage | $V_{IH}$ | 2.0 | – | 5.5 | V |
| Input Low Voltage | $V_{IL}$ | −0.5 | – | 0.8 | V |

### DC CHARACTERISTICS

| Characteristic | Symbol | Min | Typ | Max | Unit |
|---|---|---|---|---|---|
| Input Current ($V_{in} = 0$ to 5.5 V) | $I_{in}$ | −10 | – | 10 | μA |
| Output High Voltage ($I_{OH} = -220$ μA) | $V_{OH}$ | 2.4 | – | – | V |
| Output Low Voltage ($I_{OL} = 3.2$ mA) | $V_{OL}$ | – | – | 0.4 | V |
| Output Leakage Current (Three-State) ($E = 2.0$ V, $V_{out} = 0$ V to 5.5 V) | $I_{LO}$ | −10 | – | 10 | μA |
| Supply Current – Active* (Minimum Cycle Rate) | $I_{CC}$ | – | 25 | 40 | mA |
| Supply Current – Standby ($E = V_{IH}$) | $I_{SB}$ | – | 7 | 10 | mA |

*Current is proportional to cycle rate.

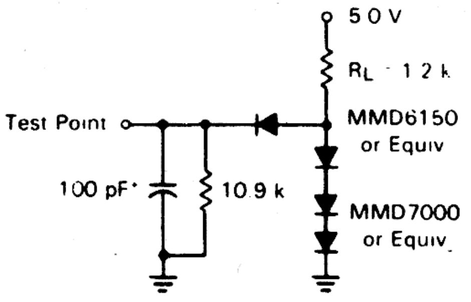
## AC OPERATING CONDITIONS AND CHARACTERISTICS

(Read Cycle)

**RECOMMENDED AC OPERATING CONDITIONS** ($T_A = 0$ to 70°C, $V_{CC} = 5.0$ V ± 10%. All timing with $t_r = t_f = 20$ ns, load of Figure 1)

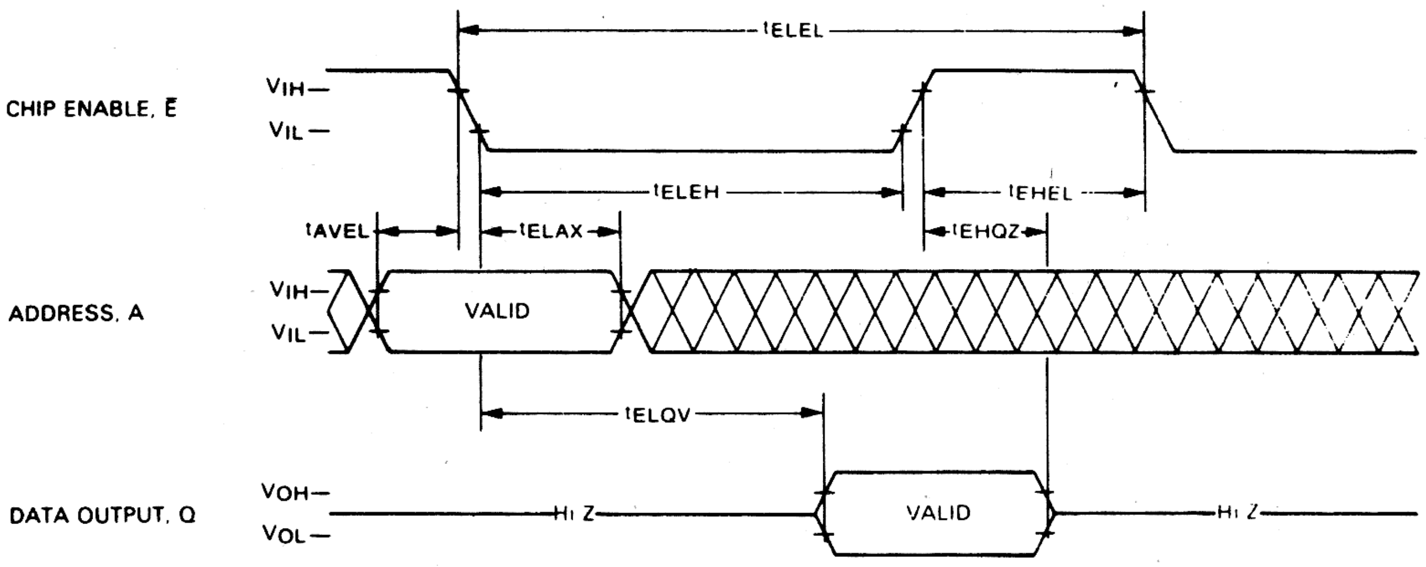
| Parameter | Symbol | MC68A39 | | MC68B39 | | Unit |
|---|---|---|---|---|---|---|
| | | Min | Max | Min | Max | |
| Chip Enable Low to Chip Enable Low of Next Cycle (Cycle Time) | $t_{ELEL}$ | 450 | – | 375 | – | ns |
| Chip Enable Low to Chip Enable High | $t_{ELEH}$ | 300 | – | 250 | – | ns |
| Chip Enable Low to Output Valid (Access) | $t_{ELQV}$ | – | 300 | – | 250 | ns |
| Chip Enable High to Output High Z (Off Time) | $t_{EHQZ}$ | – | 75 | – | 60 | ns |
| Chip Enable Low to Address Don't Care (Hold) | $t_{ELAX}$ | 75 | – | 60 | – | ns |
| Address Valid to Chip Enable Low (Address Setup) | $t_{AVEL}$ | 0 | – | 0 | – | ns |
| Chip Enable Precharge Time | $t_{EHEL}$ | 110 | – | 70 | – | ns |

**FIGURE 1 — AC TEST LOAD**



*Includes Jig Capacitance

**FIGURE 2 — TIMING DIAGRAM**



## INTRODUCTION

Since the earliest days of computers it has has been obvious that no computer was capable of doing all desirable mathematical operations in binary integer arithmetic. To meet the needs of those applications requiring the manipulation of real numbers, floating point (FP) evolved and became widely used. Unfortunately, each computer manufacturer created his own floating point (FP) representation and the ensuing wide variation in formats, accuracy, and exception handling almost guarantees that a program executed on one computer will get different results if executed on another computer.

Meanwhile, research has been completed which formulates an optional binary floating point representation. Unfortunately, the existing manufacturers have far too much money invested in software and hardware to incur the costs of conversion to a new standard. Powerful microprocessors, on the other hand, were in their infancy and the floating point experts saw the opportunity to standardize a floating point format for microprocessors. The IEEE appointed a committee to address the standard and their work resulted in the *IEEE Proposed Standard for Binary Floating Point Arithmetic Draft 8.0.*

The MC6839 represents a complete implementation of the IEEE proposed standard. Since hardware implementations of floating point (FP) are always several orders of magnitude faster (and more expensive) than software implementations, the MC6839 substitutes increased functionality for speed. Therefore, the MC6839 supports all precisions, modes, and operations required or suggested by the IEEE proposed standard.

From its very inception, the M6809 microprocessor was designed to support a concept of ROMable software by an improved instruction set and addressing modes. It was felt that the only way to reduce the escalating cost of software was for the silicon manufacturer to supply software on silicon. Since the manufacturer can amortize the cost of developing the software over a very large volume, the cost of this software, above normal masked ROM costs, will be low. Also, to be useful in many diverse systems, the ROM must be position-independent and re-entrant.

The intent of this Advance Information (data) Sheet is to provide the reader with enough information to make an intelligent decision as to whether the MC6839 is applicable to his system. The intent is not to provide all the details necessary to interface or program the MC6839; a users manual is available for that purpose. A familiarity with the MC6809 instruction set is assumed in this document.

## PHYSICAL CHARACTERISTICS

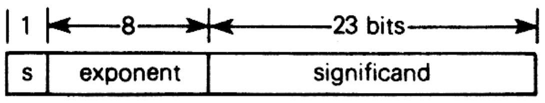
The MC6839 is housed in one 24-pin 8K-by-8 mask programmable ROM: the MCM68364. This ROM uses a single 5 V power supply and is available with access times of 250 or 350 ns. The MC6839 is designed to be used in MC6809 or MC6809E systems with up to 2 MHz internal clocks. Full device characteristics can be found at the front of this data sheet.

## FLOATING POINT FORMATS

The MC6839 supports the three precisions suggested by the IEEE Proposed Floating Point Standard: single, double, and extended. The values occupy 32, 64, and 80 bits (4, 8, and 10 bytes) respectively in the users memory. The formats of the three precisions are described in the following paragraphs.

### SINGLE FORMAT

All single precision numbers are represented in four bytes as:



The exponent is biased by +127. That is exponent of: $2^0$ is 127, $2^2$ is 129, and $2^{-2}$ is 125. The significand is stored in sign magnitude rather than twos complement form. The equation for the single form representation is:

$x = (-1)^s \times 2^{(exp-127)} \times (1.\ \text{significand})$

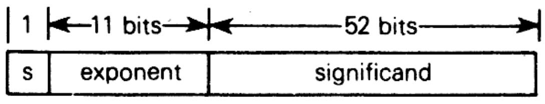
| | |
|---|---|
| s | = sign of the significand |
| exp | = biased exponent |
| significand | = bit string of length 23 encoding the significant bits of the number that follow the binary point, yielding a 24-bit significant digit field for the number that always begins "1______." |

Examples:

| | | | | | |
|---|---|---|---|---|---|
| +1.0= | $1.0 \times 2^0 =$ | \$3F | 80 | 00 | 00 |
| +3.0= | $1.5 \times 2^1 =$ | \$40 | 40 | 00 | 00 |
| −1.0= | $-1.0 \times 2^0 =$ | \$BF | 80 | 00 | 00 |

### DOUBLE FORMAT

All double precision numbers are represented by an 8-byte string as:



For double formats the exponent is biased by +1023. The rest of the interpretation is the same as for single format. The equation for double format is:

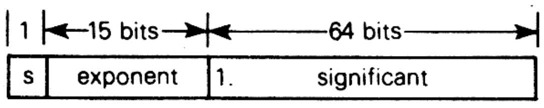
$x = (-1)^s \times 2^{(exp-1023)} \times (1.\ \text{significand})$

Examples:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7.0= | $1.75 = 2^2 =$ | \$40 | 1C | 00 | 00 | 00 | 00 | 00 | 00 |
| −30.0= | $-1.875 \times 2^4 =$ | \$C0 | 3E | 00 | 00 | 00 | 00 | 00 | 00 |
| 0.25= | $1.0 \times 2^2 =$ | \$3F | D0 | 00 | 00 | 00 | 00 | 00 | 00 |

### EXTENDED FORMAT

Single- and double-formats should be used to represent the bulk of floating point (FP) numbers in the user's system (e.g., storage of arrays). Extended should only be used for intermediate calculations such as occur in the evaluation of a complex expression. In fact, extended may not be used at all by most users, but since it is required internally, it is optionally provided. Extended numbers are represented in 10 bytes as:



A notable difference between this format and single and double is the 1.0 is explicitly present in the significand and the exponent contains no bias and is in twos complement form. The equation for double extended is:

$x = (-1)^s \times 2^{exp} \times \text{significand}$

where the significand contains the explicit 1.0.

Examples:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $0.5 =$ | $1.0 \times 2^{-1} =$ | \$7F | FF | 80 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| $-1.0 =$ | $-1.0 \times 2^{0} =$ | \$80 | 00 | 80 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| $384.0 =$ | $1.5 \times 2^{8} =$ | \$00 | 08 | C0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

## BCD STRINGS

A BCD string is the input to the BCD-to-Floating-Point conversion operation and the output of the Floating-Point-to-BCD conversion operation. All BCD strings have the following format:

| 0 | 1 ... 5 | 5 | 6 ... 24 | 25 |
|---|---|---|---|---|
| se | 4 digit BCD exponent | sf | 19 digit BCD fraction | p |

se = sign of the exponent. \$00 = plus, \$0F = minus. (one byte)
sf = sign of the fraction. \$00 = plus, \$0F = minus. (one byte)
p = number of fraction digits to the right of the decimal point. (one byte)

All BCD digits are *unpacked* and right justified in each byte:

| 7 | 0 |
|---|---|
| 0 0 0 0 | 0-9 |

The byte ordering of the fraction and exponent is consistent with all Motorola processors in that the most-significant BCD digit is in the lowest memory address.

Examples:

$2.0 = 2.0 \times 10^{0}$ (p = 0)

```
     Address   Data
     0000      00                      {se = +}
     0001      00   00   00   00       {exponent = 0}
     0005      00                      {sf = +}
     0006      00   00   00   00   00  {fraction = 2}
     000B      00   00   00   00   00
     0010      00   00   00   00   00
     0015      00   00   00   02
     0019      00                      {p = 0}
```

or $2.0 = 20{,}000 \times 10^{-4}$ (p = 0)

```
     Address   Data
     0000      0F                      {se = -}
     0001      00   00   00   04       {exponent = 4}
     0005      00                      {sf = +}
     0006      00   00   00   00   00  {fraction = 20000}
     000B      00   00   00   00   00
     0010      00   00   00   00   02
     0015      00   00   00   00
     0019      00                      {p = 0}
```

(The above might be the output of a Floating-Point-to-BCD with k = 5)

or $2.0 = 2.0 \times 10^{0}$ (p = 10)

```
     Address   Data
     0000      00                      {se = +}
     0001      00   00   00   00       {exponent = 0}
     0005      00                      {sf = +}
     0006      00   00   00   00   00  {fraction = 20000000000}
     000B      00   00   00   02   00
     0010      00   00   00   00   00
     0015      00   00   00   00
     0019      0A                      {p = 10}
```

### INTEGERS

Two sizes of integers are supported; short and double. Short integers are 16 bits long and double integers are 32 bits long. The byte ordering is consistent with all Motorola processors in that the most-significant bits are in the lowest address.

### SPECIAL VALUES

No derivable floating point format can represent the infinite number of possible real numbers, so it is very useful if some special numbers are recognized by a floating point package. These numbers are: +0, −0, + infinity, − infinity, very small (almost zero) numbers, and in some cases unnormalized numbers. Also, it is convenient to have a sepcial format which indicates that the contents of memory do not contain a valid floating point number. This "not a number" might occur if a variable is defined in a HLL and is used before it is initialized with a value. The most positive and negative exponents of each format are reserved to represent these special vaues.

The detailed description of these special values is given in a later section.

## ARCHITECTURE

All floating point operations are of the "two address" or "three address" variety; all the user need supply are the addresses of the operand(s) and the result. The package looks for operands at the specified location(s) and delivers the result to the specified destination. For example,

```
      Arg1    +    Arg2     →      Result
   <source>     <source>     <destination>
```

Intermediate results are never presented to the user; therefore, there are no internal "registers" to be concerned about, keeping the interface as simple as possible. The end result is ease of use.

There is a user defined floating point control block (fpcb) that defines the mode of the package. This control block is much like the control blocks frequently used to define I/O or operating system operations. The fpcb is discussed in detail in a later section.

## SUPPORTED OPERATIONS

The MC6839 supports the following operations. On any particular call to the floating point ROM a 1-byte opcode which immediately follows the LBSR instruction chooses the desired operation. Below are short descriptions of the functions implemented in the MC6839 along with suggested menmonics. A table containing the opcodes and calling sequences for these functions is presented at the end of this data sheet.

| **ASCII Mnemonic** | **Description** |
|---|---|
| FADD | Add arg1 to arg2 and store the result. |
| FSUB | Subtract arg2 from arg1 and store the result. |
| FMUL | Multiply arg1 times arg2 and store the result. |
| FDIV | Divide arg1 by arg2 and store the result. |
| FREM | Take the remainder of arg1 divided by arg2 and store the result. The remainder is biased to lie in the range −arg2/2 < remainder < +arg2/2, instead of the usual range of 0 ≤ remainder < arg2. This bias makes the function more useful in the implementation of trigonometric and other functions. |
| FCMP | Compare arg1 with arg2 and set the condition codes to the result of the compare. Arg1 and arg2 can be of different precisions. |
| FTCMP | Compare arg1 with arg2 and set the condition codes to the result of the compare. In addition, trap if an unordered exception occurs regardless of the state of the UNOR (unordered) bit in the trap enable byte of the fpcb. |
| FPCMP | A predicate compare; this means compare arg1 with arg2 and affirm or disaffirm the input predicate (e.g., 'is arg1 = arg2' or 'is arg1 < arg2'). |
| FTPCMP | A trapping predicate compare; same as the predicate compare except trap on an unordered exception regardless of the state of the UNOR (unordered) bit in the trap enable byte of the fpcb. |
| FSQRT | Returns the square root of arg2 in the result. |
| FINT | Returns the interger part of arg2 in the result. The result is still a floating point number. For example, the integer part of 3.14159 is 3.00000. |
| FFIXS | Convert arg2 to a short (16-bit) binary integer. |
| FFIXD | Convert arg2 to a long (32-bit) binary integer. |
| FFLTS | Convert a short binary integer to a floating point result. |
| FFLTD | Convert a long binary integer to a floating point result. |
| BINDEC | Convert a binary floating point value to a BCD decime string. |
| DECBIN | Convert a BCD decimal string to a binary floating point result. |
| FABS | Return the absolute value of arg2 in the result. |
| FNEG | Return the negative of arg2 in the result. |
| FMOV | Move (or convert) arg1 → arg2. This function is useful for changing precisions (e.g., single to double) with full exception checking for possible overflow or underflow. |

All routines, except FMOV and the compares, accept arguments of the same precision and generate a result with the same precision. For moves and compares the sizes of the arguments are passed to the package in a parameter word.

Details of each operation can be found in the *MC6839 Users Manual.*

## MODES OF OPERATION

In addition to supporting a wide range of precisions and operations, the MC6839 supports all modes required or suggested by the IEEE Proposed Floating Point Standard. These include rounding modes, infinity closure modes, and exception handling modes. The various modes are selected by bits in the floating point control block (fpcb) that resides in user memory. Thus, each user or task can have a unique set of modes in effect for his calculations. The selection bits are defined in a later section on the fpcb.

### ROUNDING MODES

Four rounding modes are suggested by the IEEE Proposed Floating Point Standard. They are:

| | |
|---|---|
| 1. Round to nearest | (RN) |
| 2. Round toward zero | (RZ) |
| 3. Round toward plus infinity | (RP) |
| 4. Round toward minus infinity | (RN) |

Round nearest will be used by most users because it provides the most accurate answers for most calculations. Round towards zero (truncate) is useful when the MC6839 implements real numbers in some high level languages that require truncation (i.e., FORTRAN). Round towards plus and minus infinity are used in interval arithmetic.

Normally a result is rounded to the precision of its destination. However, when the destination is Extended, the user can specify that the result significand be rounded to the precision of the basic format – single, double, or extended – of his choice, although the exponent range remains extended.

**NO DOUBLE ROUNDING** – The MC6839 is implemented such that no result will undergo more than one rounding error.

### INFINITY CLOSURE MODES

The way in which infinity is handled in a floating point package may limit the number of applications in which the package can be used. To solve this problem, the proposed IEEE standard requires two types of infinity closures. A bit in the control byte of the Floating Point Control Block (fpcb) will select the type of closure that is in effect at any time.

**AFFINE CLOSURE** – In affine closure:

minus infinity < {every finite number} < plus infinity

Thus, infinity takes part in the real number system in the same manner as any other signed quantity.

**PROJECTIVE CLOSURE** – In projective closure:

infinity = minus infinity = plus infinity

and all comparisons between infinity and a floating point number involving order relations other than equal ( = ) or not equal ( ≠ ) are invalid operations. In projective closure the real number system can be thought of as a circle with zero at the top and infinity at the bottom.

### NORMALIZE MODE

The purpose of the normalize mode is to prevent unnormalized results from being generated, which can otherwise happen. Such an unnormalized result arises when a denormalized operand is operated on such that its fraction remains not normalized but its exponent is no longer at its original minimum value. By transforming denormalized operands to normalized, internal form upon entering each operation, unnormalized results are guaranteed not to occur.

Thus, when operating in this mode the user can be assured that no attempt will be made to return an unnormalized value to a single or double destination. A bit in the control byte of the fpcb selects whether or not this mode is in effect. This mode is forced whenever the round mode is either round toward plus or minus infinity. Unnormalized numbers entering an operation are not affected by this mode, only denormalized ones are. Unnormalized and denormalized operands are discussed in a later section.

## EXCEPTIONS

One of the greatest strengths of the IEEE Proposed Floating Point Standard is the regular and consistant handling of exceptions. Existing floating point implementations are quite varied in the way they handle exceptions, so the proposed IEEE standard has very carefully prescribed how exceptions must be handled and what constitutes an exception. Seven types of exceptions will be recognized by the MC6839. Only the first 5 are required by the proposed IEEE standard. They are:

1. Invalid Operation – a general exception that arises when an operation has gone so wrong that the program cannot return any reasonable result or fit the exception into any of the other more specific classes.
2. Underflow – arises when an operation generates a result that is too small to fit into the desired result precision.
3. Overflow – arises when an operation generates a result that is too large to fit into the desired result precision.
4. Division by Zero – arises when division by zero is attempted.
5. Inexact Result – arises when the result of an operation was not exact and therefore was rounded to the desired precision before being returned to the user.
6. Integer Overflow – arises when the binary integer result of a FIXS(D) operation cannot fit into 16(32) bits.
7. Comparison of Unordered Values – arises when one of the arguments to a compare operation is a "NAN" or an infinity in the projective closure mode. (See the Infinity and Not a Number paragraphs for further explanation of NANs and infinity.)

For each exception the caller will be given the option of specifying whether the package should: (1) trap to a user supplied trap routine to process the exception, or (2) deliver a default result specified by the proposed standard and proceed with execution. For most users the default result is adequate and the user need not write any trap handlers. Regardless of whether a trap is specified or not, a status bit will be set in the status byte of the fpcb and will remain set until cleared by the caller's program. Selection of whether to trap or to continue will be made by setting bits in the trap enable byte of the fpcb. For more details on the fpcb see the section on the Floating Point Control Block (fpcb).

If a trap is taken, the floating point package supplies a pointer that points to an area on the stack containing the following diagnostic information:

1. Event that caused the trap (overflow, etc.)
2. Where in the caller's program
3. Opcode
4. The input operands
5. The default result in internal format

In the event more than one exception occurs during the same operation, only one trap is invoked according to the following precedence.

1. Invalid Operation
2. Overflow
3. Underflow
4. Division by Zero
5. Unordered
6. Integer Overflow
7. Inexact Result

The user supplied trap routine (if any) will usually do 1 of 3 things:

1. Fix the result
2. Do nothing to the result and allow the floating point package to deliver the default value to the result.
3. Abort execution.

Sufficient detils on how to write a trap routine are furnished in the *MC6839 Users Manual.*

## USER INTERFACE

There are two types of calls to the floating point package: register calls and stack calls. For register calls the user loads the machine registers with pointers (addresses) to the operand(s) and to the result; the call to the package is then performed. For stack calls the operand(s) is pushed on the stack and the call to the package is performed with the result replacing the operands on the stack after completion. The operand(s) must be pushed least-significant bytes first; this is consistent with the other Motorola architectures in that the most-significant byte resides in the lowest address. The two types of calls look like:

General form of a register call:

```
load registers
LBSR fpreg   register call
FCB  opcode
```

Example of a position-independent call to the add routine:

```
LEAU   arg1, pcr
LEAY   arg2, pcr
LEAX   fpcbptr, pcr      pointer to fpcb
TFR    x, d
LEAX   result, pcr
LBSR   fpreg
FCB    fadd
```

General form of a stack call:

```
push arguments
LBSR   fpstak      stack call
FCB    opcode
pull   result
```

Example of a stack call to the add routine:

```
push argument 1
push argument 2
push   fpcbptr      pointer to fpcb
LBSR   fpstak
FCB    fadd
pull   result
```

Details of the calling sequence for every type of operation can be found in the *MC6839 Users Manual;* a reference table of calling sequences and opcodes can be found at the end of this data sheet.

## STACK REQUIREMENTS

When the MC6839 is called by the user, the package reserves local storage on the hardware stack. It then moves the input arguments from user memory to the local storage area and expands them into a convenient internal format. The operations use these "internal" numbers to arrive at an "internal" result which is then converted to the memory format of the result and returned to the user. For this reason, the user must insure that adequate memory exists on the hardware stack before calling the MC6839. The maximum stack sizes that any particular function will ever find necessary are:

| | |
|---|---|
| register calls | 150 bytes |
| stack calls | 185 bytes |

## FLOATING POINT CONTROL BLOCK (fpcb)

The fpcb is a user-defined block that contains information needed by the floating point package. The fpcb is also used to pass status back to the caller or to invoke the trap routine. The fpcb must reside in the user RAM space to insure that the package can remain re-entrant. The caller of the floating point package must pass the address of the fpcb on each call. The format of the fpcb is:

| | |
|---|---|
| control byte | 0 |
| trap enable byte | 1 |
| status byte | 2 |
| secondary status byte | 3 |
| address of trap routine | 4 |
| | 5 |

The meaning of the various bit fields within the fpcb are discussed in detail in the following paragraphs.

**CONTROL BYTE** — The control byte configures the floating point package for the caller's operation and is written by the user. Various fields in the byte set the precision, round, infinity closure, and normalize modes.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Precision | | | x | NRM | Round Mode | | A/P |

Bit 0 Closure (A/P) Bit
0 = projective closure
1 = affine closure

Bits 1-2 Round Mode
00 = round to nearest (RN)
01 = round to zero (RZ)
10 = round to plus infinity (RP)
11 = round to minus infinity (RM)

Bit 3 Normalize (NRM) Bit
1 = normalize denormalized numbers while in internal format before using. Precludes the creation of unnormalized numbers.
0 = do not normalize denormalized numbers (warning mode)

**NOTE**

If the rounding mode is RM or RP then normalize mode is forced. Unnormalized numbers are not affected by bit 3.

Bit 4 Undefined, reserved

Bits 5-7 Precision Mode
000 = Single
001 = Double
010 = Extended with no forced rounding of result
011 = Extended — force round result to single
100 = Extended — force round result to double
101 = Undefined, reserved
110 = Undefined, reserved
111 = Undefined, reserved

Note that if the control byte is set to zero by the user, all defaults in the IEEE Proposed Floating Point Standard will be selected.

**STATUS BYTE**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | INX | IOV | UN | DZ | UNF | OVF | IOP |

The bits in the status byte are set if any errors have occurred. Each bit of the status byte is a "sticky" bit in that it must be manually reset by the user. The FP package writes bits into the status byte but never clears existing bits. This is done so that a long calculation can be completed and the status need only be checked once at the end.

| | |
|---|---|
| Bit 0 | Invalid opertion (see secondary status) |
| Bit 1 | Overflow |
| Bit 2 | Underflow |
| Bit 3 | Division by zero |
| Bit 4 | Unordered |
| Bit 5 | Integer overflow |
| Bit 6 | Inexact result |
| Bit 7 | Undefined, reserved |

**TRAP ENABLE BYTE**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | INX | IOV | UNOR | DZ | UNF | OVF | IOP |

A "1" in any bit of the trap enable byte enables the FP package to trap if that error occurs. The bit definitions are the same as for the status byte. Note that if a trapping compare is executed and the result is unordered, then the unordered trap will be taken regardless of the state of the UNOR bit in the trap enable byte.

**SECONDARY STATUS (SS)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | x | x | Invalid Operation Type | | | | |

The FP package will write a status into this byte any time a new IOP occurs. As is the case with the status bytes, it is up to the caller to reset the "IOP type" field.

Bits 0-4 Invalid Operation Type Field

0 = no IOP error
1 = square root of a negative number, infinity in projective mode, or a not normalized number
2 = (+ infinity) + (− infinity) in affine mode
3 = tried to convert NAN to binary integer
4 = in division: 0/0, infinity/infinity or divisor is not normalized and the dividend is not zero and is finite
5 = one of the input arguments was a trapping NAN
6 = unordered values compared via predicate other than = or ≠
7 = k out of range for BINDEC or p out of range for DECBIN
8 = projective closure use of + / − infinity
9 = 0 × infinity
10 = in REM arg2 is zero or not normalized or arg1 is infinite
11 = unused, reserved
12 = unused, reserved
13 = BINDEC integer too big to convert
14 = DECBIN cannot represent input string
15 = tried to MOV a single denormalized number to a double destination
16 = tried to return an unnormalized number to single or double (invalid result)
17 = division by zero with divide by zero trap disabled

**TRAP VECTOR** — If any of the traps occur, the FP package will *jump* indirectly through the trap address in the fpcb with an index in the A accumulator indicating the trap type:

0 = Invalid Operation
1 = Overflow
2 = Underflow
3 = Divide by Zero
4 = Unnormalized
5 = Integer Overflow
6 = Inexact Result

If more than 1 enabled trap occurs, the MC6839 will return the index of the highest priorty enabled error. Index = 0 = invalid operation is the highest priority, and, index = 6 is the lowest.

## SPECIAL VALUES (SINGLE- AND DOUBLE-FORMAT)

The encoding of the special values are given below. Generally, when used as operands, the special values flow through an operation creating a predictable result. Note that as with normalized numbers the extended format differs slightly from the single- and double-formats.

### ZERO

Zero is represented by a number with both a zero exponent and a zero significand. The sign is significant and differentiates between plus or minus zero.

| | | |
|---|---|---|
| s | 0 | 0 |

### INFINITY

The infinities are represented by a number with the maximum exponent and a zero significand. The sign differentiates plus or minus infinity.

| | | |
|---|---|---|
| s | 1111......1111 | 0 |

### DENORMALIZED (SMALL NUMBERS)

When a number is so small that its exponent is the smallest allowable normal biased value (1), and it is impossible to normalize the number without further decrementing the exponent, then the number will be allowed to become denormalized. The format for denormalized numbers has a zero exponent and a non-zero significand. Note that in this form the implicit bit is no longer 1 but is zero. The interpretation for denormalized numbers is:

Single: $X = (-1)^{s} \times 2^{-126} \times (0.\text{ significand})$
Double: $X = (-1)^{s} \times 2^{-1022} \times (0.\text{ significand})$

Note that the exponent is always interpreted as $2^{-126}$ for single and $2^{-1022}$ for double instead of $2^{-127}$ and $2^{-1023}$ as might be expected. This is necessary since the only way to insure the implicit bit becomes zero is to right shift the significand (divide by 2) and increment the exponent (multiply by 2). Thus, the exponent ends up with the interpretation of $2^{-126}$ or $2^{-1022}$.

The format for denormalized numbers is:

| | | |
|---|---|---|
| s | 0 | non-zero |

Note that zero may be considered a special case of denormalized numbers where the number is so small that the significand has been reduced to zero.

Examples:
Single:
$1.0 \times 2^{-128} = 0.25 \times 2^{-126} =$ \$00 20 00 00
Double:
$1.0 \times 2^{-1025} = 0.125 \times 2^{-1022} =$ \$00 02 00 00 00 00 00 00

## NOT A NUMBER (NAN)

A number containing a NAN indicates that the number is not a valid floating number. NANs can be used to initialize areas in memory to indicate they have not had a valid floating point number stored in them. They are also created by the MC6839 to indicate that an operation could not return a valid result.

The format for a NAN has the largest allowable exponent, a non-zero significand, and an undefined sign. As an implementation feature (not required by the IEEE Proposed Floating Point Standard), the non-zero fraction and undefined sign are further defined:

| | | | | |
|---|---|---|---|---|
| d | 1111....1111 | t | operation address | 00....0000 |

| | | |
|---|---|---|
| d: | 0 = | This NAN has never entered into an operation with another NAN. |
| | 1 = | This NAN has entered into an operation with other NANs. |
| t: | 0 = | This NAN will not necessarily cause an invalid operation trap when operated upon. |
| | 1 = | This NAN will cause an invalid operation trap when operated upon (trapping NAN). |

Operation address:

The 16 bits, immediately to the right of the t bit, contain the address of the instruction immediately following the call to the FP package of the operation that caused the NAN to be created. If d (double NAN) is also set, the address is arbitrarily one of the addresses in the two or more offending NANs.

## SPECIAL VALUES (EXTENDED FORMAT)

### ZERO

Zero is represented by a number with the smallest unbiased exponent and a zero significand:

| | | |
|---|---|---|
| s | 100.....0000 | 0 |

### INFINITY

Infinity has the maximum unbiased exponent and a zero significand:

| | | |
|---|---|---|
| s | 011111.....11 | 0 |

### DENORMALIZED NUMBERS

Denormalized numbers have the smallest unbiased exponent and a non-zero significand:

| | | |
|---|---|---|
| s | 100......000 | 0. non-zero |

The exponent of denormalized extended and internal numbers is interpreted as having the exponent value 1 greater than the smallest unbiased exponent value. Thus, a denormalized number has the exponent − 16384, but has the value:

$(-1)^s \times 2^{-16383} \times 0.f$

Example:

$1.0 \times 2^{-16387} = 0625 \times 2^{-16383} =$ $40 00 08 00 00 00 00 00 00 00

### NANs

NANs have the largest unbiased exponent and a non-zero significand. The operation addresses "t" and "d" are implementation features and are the same as for single- and double-formats.

| | | | | | |
|---|---|---|---|---|---|
| d | 011.....1111 | 0 | t | operation addr. | 00000000 |

The operation address always appears in the 16 bits immediately to the right of the t bit.

## UNNORMALZIED NUMBERS

Unnormalized numbers occur only in extended or internal format. Unnormalized numbers have an exponent greater than the minimum in the extended format (i.e., they are not denormalized or normal zero) but the explicit leading bit is a zero. If the significand is zero, this is an unnormalized zero. Even though unnormalized numbers and denormalized numbers are handled similarly in most cases, they should not be confused. Denormalized numbers are numbers that are very small — have minimum exponent — and hence have lost some bits of significance. Unnormalized numbers are not necessarily small (the exponent may be large or small) but the significand has lost some bits of significance, hence, the explicit bit and possibly some of the bits to the right of the explicit bit are zero.

| | | |
|---|---|---|
| s | >100...000 | 0. significand |

Note that unnormalized numbers cannot be represented — and hence cannot exist — for single- and double-formats. Unnormalized numbers can only be created when denormalized numbers in single- or double-format are represented in extended or internal formats.

Example:
$.0625 \times 2^2$ (unnorm.) = \$00 02 08 00 00 00 00 00 00 00

### MC6839 CALLING SEQUENCE AND OPCODE REFERENCE TABLE

| Function | Opcode | Register Calling Sequence | Stack Calling Sequence[1] |
|---|---|---|---|
| FADD<br>FSUB<br>FMUL<br>FDIV | \$00<br>\$02<br>\$04<br>\$06 | U←Addr. of Argument #1<br>Y←Addr. of Argument #2<br>D←Addr. of FPCB<br>X←Addr. of Result<br>LBSR FPREQ<br>FCB <opcode> | Push Argument #1<br>Push Argument #2<br>Push Addr. of FPCB<br>LBSR FPSTAK<br>FCB <opcode><br>Pull Result |
| FREM<br>FSQRT<br>FINT<br>FFIXS<br>FFIXD<br>FAB<br>FNEG<br>FFLTS<br>FFLTD | \$08<br>\$12<br>\$14<br>\$16<br>\$18<br>\$1E<br>\$20<br>\$24<br>\$26 | Y←Addr. of Argument<br>D←Addr. of FPCB<br>X←Addr. of Result<br>LBSR FPREG<br>FCB <opcode> | Push Argument<br>Push Addr. of FPCB<br>LBSR FPSTAK<br>FCB <opcode><br>Pull Result |
| FCMP<br>FTCMP<br>FPCMP<br>FTPCMP | \$8A<br>\$CC<br>\$8E<br>\$D0 | U←Addr. of Argument #1<br>Y←Addr. of Argument #2<br>D←Addr. of FPCB<br>X←Parameter Word<br>LBSR FPREG<br>FCB <opcode><br><br>NOTE: Result returned in the CC register. For predicate compares the Z-Bit is set if predicate is affirmed cleared if disaffirmed. | Push Argument #1<br>Push Argument #2<br>Push Parameter Word<br>Push Addr. of FPCB<br>LBSR FPSTAK<br>FCB <opcode><br>Pull Result (if predicate compare,<br>NOTE: Result returned in the CC register for regular compares. For predicate compares a one byte result is returned on the top of the stack. The result is zero if affirmed and −1(\$FF) if disaffirmed. |
| FMOV | \$9A | U←Precision Parameter Word<br>Y←Addr. of Argument<br>D←Addr. of FPCB<br>X←Addr. of Result<br>LBSR FPREG<br>FCB <opcode> | Push Argument<br>Push Precision Parameter Word<br>Push Addr. of FPCB<br>LBSR FPSTAK<br>FCB <opcode><br>Pull Result |
| BINDEC | \$1C | U←k (# of digits in result)<br>Y←Addr. of Argument<br>D←Addr. of FPCB<br>X←Addr. of Decimal Result<br>LBSR FPREG<br>FCB <opcode> | Push Argument<br>Push k<br>Push Addr. of FPCB<br>LBSR FPSTAK<br>FCB <opcode><br>Pull BCD String |
| DECBIN | \$22 | U←Addr. of BCD Input String<br>D←Addr. of FPCB<br>X←Addr. of Binary Result<br>LBSR FPREG<br>FCB <opcode> | Push Addr. of BCD Input String<br>Push Addr. of FPCB<br>LBSR FPSTAK<br>FCB <opcode><br>Pull Binary Result |

[1]All arguments are pushed on the stack least-significant bytes first so that the high-order byte is always pushed last and resides in the lowest address.

Entry points to the MC6839 are defined as follows:
FPREG = ROM start + \$3D
FPSTAK = ROM start + \$3F

**MC6839 EXECUTION TIMES**
**Time in μs Using 2 MHz 6809**

| **Function** | **Single Precision** | **Double Precision** | **Extended Precision** |
|---|---|---|---|
| FADD | 1200 – 3300<br>t = 1200 + 40(A) + 50(N)<br>where:<br>A = # shifts to align operands<br>N = # shifts to normalize result | 1500 – 3700<br>t = 1500 + 40(A) + 50(N) | 1100 – 3800<br>t = 1100 + 40(A) + 50(N) |
| FSUB | ADD + 11 | ADD + 11 | ADD + 11 |
| FMUL | 1400 – 1600 | 4100 – 4300 | 4600 – 4800 |
| FDIV | t = 2700 + 60(Q)<br>where:<br>Q = # of quotient bits which are are a '1' | t = 5000 + 60(Q) | 5 = 6500 + 60(Q) |
| FABS | 540 | 750 | 650 |
| DECBIN<br>(time depends on magnitude of input) | 8500 – 14,000 | 8500 – 23,000 | – |
| BINDEC<br>(time depends on # significand digits requested) | 35,000 – 48,000 | 67,000 – 85,000 | – |