# IBM System/370 RPQ

# High Accuracy Arithmetic

Publication Number
SA22-7093-0

File Number
S370-01

## PREFACE

This publication describes an IBM System/370 RPQ, high-accuracy arithmetic. The facility performs the floating-point operations of addition, subtraction, multiplication, division, and scalar product (the sum of products) with the maximum accuracy possible within the floating-point-number representation. The instructions of this facility are used, in conjunction with other System/370 floating-point instructions, in either the System/370 mode or the ECPS:VSE mode, if the model provides the mode.

The reader should be familiar with the IBM System/370 Principles of Operation, GA22-7000, or the IBM 4300 Processors Principles of Operation for ECPS:VSE Mode, GA22-7070, as appropriate, and particularly with Chapter 9, "Floating-Point Instructions," of either of those publications.

The facility discussed in this publication is not available on all models. At the time of publication, it is provided on the IBM 4361 Processor. The information published herein should not be construed as implying any intention by IBM to provide the facility on models other than those for which it is announced. For more information concerning the availability of this facility on any particular model, refer to the latest edition of the Functional Characteristics manual for the model.

First Edition (January 1984)

Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM equipment, refer to the latest IBM System/370 and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Product Publications, Department B98, PO Box 390, Poughkeepsie, NY, U.S.A. 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The high-accuracy-arithmetic facility provides instructions which perform arithmetic on floating-point numbers in the System/370 hexadecimal short and long formats with a choice of one of four different rounding options, as well as instructions which permit such floating-point numbers and their products to be accumulated without rounding errors in floating-point accumulators. These instructions are used in conjunction with other floating-point instructions, which perform the load, store, compare, and sign-change operations.

Any floating-point arithmetic operation may introduce a rounding error not exceeding one unit in the last place. This is true of all floating-point arithmetic instructions, but the high-accuracy-arithmetic instructions contain additional functions which are designed particularly to support interval arithmetic. Interval arithmetic produces at each computational step a pair of results, which are the upper and lower bounds for the exact result. These bounds provide a direct indication, not available with only a single result, of the magnitude of the rounding errors which are building up.

The high-accuracy-arithmetic facility also provides an accumulator function, which allows the scalar product (the sum of products) of two vectors to be formed with the same high accuracy as the basic floating-point operations of addition, subtraction, multiplication, and division. The scalar product is produced by the instruction MULTIPLY AND ACCUMULATE and placed in a special accumulator, which has enough digit positions to contain the exact sum without rounding. Only a single rounding error of at most one unit in the last place is introduced when the completed scalar product is returned to one of the floating-point registers.

The facility is described in three parts. The first part covers the methods of rounding and other common aspects of the basic arithmetic instructions which are subject to rounding. The second part discusses floating-point accumulators. The third part contains the individual instruction descriptions.

## FLOATING-POINT INSTRUCTIONS WITH ROUNDING OPTIONS

The floating-point instructions with rounding options add, subtract, multiply, or divide floating-point numbers in the long or short hexadecimal formats, the normalized result being rounded in one of four ways, depending on a specified rounding mode. There is also an instruction which rounds from the long to the short format with the same four rounding options.

The floating-point instructions with rounding options are in addition to and do not replace the floating-point instructions described in Chapter 9, "Floating-Point Instructions," of the appropriate Principles of Operation. To distinguish between the two types of instruction, those described in the Principles of Operation are referred to as instructions without rounding options. The instructions without rounding options either truncate or round the result, but there is only one method for each instruction, and that method, in general, is not the same as any of the four rounding options.

Other instructions, which are described in the Principles of Operation and which are not affected by rounding, are needed in conjunction with the floating-point instructions with rounding options to load, store, compare, or change the sign of floating-point numbers.

All instructions with rounding options have the RRE format. Their arithmetic operands and results reside in floating-point registers. The rounding mode is specified by the contents of general register 0.

The floating-point instructions with rounding options differ from the instructions without rounding options primarily in the following respects:

- The result is rounded according to the rounding mode in general register 0. The results of corresponding operations may differ by one unit in the last place.
- The instructions are all four bytes long.
- No RX-format operations are provided.
- No significance exception is recognized.
- All floating-point instructions with rounding options normalize their operands before the arithmetic operation starts, not just MULTIPLY and DIVIDE.

- MULTIPLY WITH ROUNDING with short-format operands (MERN) produces a rounded result in the short format, instead of the exact result in the long format produced by MULTIPLY (MER).
- LOAD WITH ROUNDING (LERN) produces a rounded result in the short format, as does LOAD ROUNDED (LRER), but the result of LERN is subject to the rounding mode and may differ from the result produced by LRER. If the operand is unnormalized, the results also differ because LERN normalizes the operand before rounding and LRER does not.

Programming Notes

1. No floating-point instruction with rounding options corresponds to HALVE (HDR and HER). The equivalent result can be obtained by using instructions with rounding options to divide the operand by 2 or multiply the operand by 0.5.
2. There is no comparison operation with rounding options. The floating-point COMPARE instruction should be used in conjunction with the floating-point instructions with rounding options only if all operands are normalized. When used on unnormalized operands that are nearly equal, COMPARE may indicate equal when SUBTRACT WITH ROUNDING with the same operands would produce a nonzero result.
3. No instructions with rounding options correspond to ADD UNNORMALIZED and SUBTRACT UNNORMALIZED, which may produce results that are not normalized.

## NORMALIZATION

All floating-point instructions with rounding options normalize their operands before performing the arithmetic: a nonzero operand is normalized at the start of the operation, and an operand with a zero fraction is treated as if it were a true zero. The instructions thus produce the same results for normalized and unnormalized operands which have the same values. The instructions leave the registers containing the unnormalized operand unchanged, unless that register is also the target of the result. All nonzero results are in normalized form.

## ROUNDING

When the exact result of a floating-point operation with rounding options does not fit in the specified result format, the result is rounded. The method of rounding depends on the rounding mode chosen. The results produced by a given operation with a given set of operands but different rounding methods never differ by more than one unit in the rightmost bit position to be retained in the result.

There are four rounding modes: round to zero, round to nearest, round down, and round up.

Rounding Modes

Conceptually, the exact result of an operation is the result that would be obtained if the format could have an infinite number of fraction bits. If such an exact result can be represented accurately in the result format, then the result register receives the exact result, and there is no error. If the exact result cannot be represented accurately by the number of fraction bits in the result format, then the rounding modes provide a choice of one of the two representable values which are the immediate neighbors of the exact value, there being no other representable number between those two neighbors. One of the neighbors is greater and the other smaller than the exact result.

When the rounding mode is round to zero, the fraction value chosen is the one that is smaller in magnitude.

When the rounding mode is round to nearest, the fraction value chosen is the one that is nearest in value to the exact result. If both neighboring values are equally close to the exact value, the value chosen is the one in which the rightmost fraction bit is zero.

When the rounding mode is round up (towards plus infinity), the fraction value chosen is the algebraically greater of the two, taking the number sign into account.

When the rounding mode is round down (towards minus infinity), the fraction value chosen is the algebraically lesser of the two, taking the number sign into account.

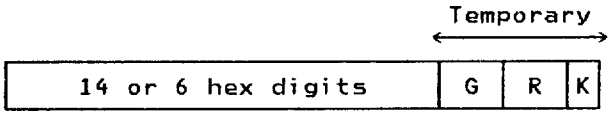
The rounding mode is specified by bits 30 and 31 of general register 0:

| Bit 30 | Bit 31 | Rounding Mode |
|---|---|---|
| 0 | 0 | Round to zero |
| 0 | 1 | Round to nearest |
| 1 | 0 | Round down |
| 1 | 1 | Round up |

Bits 0-29 of general register 0 must be zeros when a floating-point instruction with rounding options is issued. Otherwise, a specification exception is recognized.

## Guard Digit, Rounding Digit, and Sticky Bit

To perform the above rounding operations, most floating-point instructions with rounding options must temporarily retain more information in the intermediate-result fraction to the right of the 14 or six hexadecimal fraction digits of the long or short format, respectively. As a convenient description of the ADD WITH ROUNDING and SUBTRACT WITH ROUNDING instructions, this temporary information is shown as a hexadecimal guard digit, which is the same digit that is used for instructions without rounding options, followed on the right by a hexadecimal rounding digit, and a sticky bit. (The bit in the rightmost temporary bit position is called the "sticky" bit because a right shift causes all bits that are shifted in from the left to be ORed into that bit, and no bits to be shifted out to the right.) An implementation may use a different number of bits to achieve the same result; for example, a single rounding bit may be used in place of a hexadecimal rounding digit.




G: Guard digit
R: Rounding digit
K: Sticky bit

Intermediate-Result Fraction for Addition and Subtraction

The guard digit, rounding digit, and sticky bit are always set to zeros at the beginning of an operation, and their contents are discarded after rounding is completed. Whenever an operand fraction or the intermediate-result fraction is shifted right, digits shifted out of the rightmost fraction digit enter the guard-digit position, digits shifted out of the guard digit enter the rounding-digit position, and the bits of each digit shifted out of the rounding digit are ORed into the sticky bit. Thus, the sticky bit reflects whether any nonzero values were shifted out of the rounding digit, regardless of the amount of the right shift.

During addition or subtraction, the guard digit, rounding digit, and sticky bit of the shifted operand participate in the arithmetic operation, together with zeros for the corresponding digits and the sticky bit of the unshifted operand. When the intermediate-result fraction must be shifted left to eliminate leftmost zero digits, the guard digit moves into the rightmost digit position of the result fraction, the rounding digit moves into the guard-digit position, and zeros are placed in the rounding-digit position.

If, after any left shift, the guard digit, rounding digit, and sticky bit are all zeros, the result fraction is exact, and rounding causes no change in the result, regardless of the rounding mode.

Rounding is performed, after any left shift to delete leftmost zeros, as follows:

- When the rounding mode is round to zero, the guard digit, rounding digit, and sticky bit are simply ignored.
- When the rounding mode is round to nearest, a one is added to the leftmost bit of the guard digit, ignoring the result sign. If, after propagating any carries into the result fraction, all bits of the guard and rounding digits and the sticky bit are zeros (the result was exact to within one-half unit in the rightmost bit position), the rightmost bit to be retained in the result fraction is set to zero.
- When the rounding mode is round up, the result is positive, and any bits of the guard and rounding digits and the sticky bit are ones, then a one is added to the rightmost bit to be retained in the result fraction, ignoring the result sign, and any carries are propagated. Nothing is added when the result is negative or all bits of the guard and rounding digits and the sticky bit are zeros.
- When the rounding mode is round down, the result is negative, and any bits of the guard and rounding digits and the sticky bit are ones, then a one is added to the rightmost bit to be retained in the result fraction, ignoring the result sign, and any carries are propagated. Nothing is added when the result is positive or all bits

of the guard and rounding digits and the sticky bit are zeros.

If rounding causes a carry out of the leftmost hexadecimal digit position of the fraction, the fraction is shifted right one digit position, the new leftmost hexadecimal digit of the fraction is set to one, and the characteristic is increased by one.

For the remaining floating-point instructions with rounding options, the equivalent of the guard and rounding digits and the sticky bit may be obtained as follows.

After developing the quotient digits needed for the format specified for DIVIDE WITH ROUNDING, only one more quotient bit needs to be computed, which is the equivalent of the leftmost bit of the guard digit. If the remainder at that point is all zeros, the quotient has been computed exactly, and rounding proceeds as for a result that has zeros in the rightmost three bit positions of the guard digit and in all positions of the rounding digit and the sticky bit. If the remainder is nonzero, rounding proceeds as for nonzero bits in those positions.

MULTIPLY WITH ROUNDING produces an exact, double-length product as the intermediate-result fraction (28 or 12 digits for the long or short format, respectively) and rounds it to the long or short result format. Similarly, LOAD WITH ROUNDING rounds a long operand fraction, which is considered exact, to a short result fraction. In both cases, the right part of the exact fraction determines how the left part is rounded. The leftmost two digits of the right part may be considered to be the guard and rounding digits; the sticky bit is then considered to be zero if the remaining digits of the right part are all zeros and to be one otherwise.

Programming Notes

1. Since the operands of add and subtract operations are normalized before the arithmetic is performed, their intermediate-result fractions can have more than one leftmost zero digit only when two nearly equal operands of like sign are subtracted (or two operands of nearly equal magnitude and opposite sign are added), with the lesser operand having been right-shifted by no more than one digit position. A subsequent left shift of the intermediate-result fraction by more than one digit position implies that the rounding digit and sticky bit are zeros. A nonzero rounding digit is, therefore, never shifted left beyond the guard-digit position, and it does not matter whether the sticky bit is shifted or remains in position.

2. With normalized operands, rounding to zero is similar to the truncation provided by the arithmetic instructions without rounding options. The result is the same for multiplication, division, and addition (ADD with operands of like sign, or SUBTRACT with operands of opposite sign). The result may differ, however, for subtraction (ADD with operands of opposite sign, or SUBTRACT with operands of like sign).

   When using the instructions without rounding options for subtraction, any ones to the right of the guard digit are dropped during the initial right shift of the operand with the smaller characteristic. This may cause a carry to be lost during the subtraction, so that the result may be the neighboring value that is larger in magnitude rather than smaller.

   The instructions with rounding options require the machine to keep more temporary bits, described above as a rounding digit and a sticky bit. These extra bits allow the machine to generate the same carry during subtraction that would have occurred if all bits had been retained during the initial right shift of the operand fraction corresponding to the smaller characteristic. The carry ensures that the result is the one that is smaller in magnitude, as required when rounding towards zero.

ARITHMETIC EXCEPTIONS

No significance exception is recognized for the instructions ADD WITH ROUNDING and SUBTRACT WITH ROUNDING. The result of subtracting two operands of equal value is a true zero, and no interruption occurs, regardless of the value of the significance mask in the PSW.

The other arithmetic exceptions -- exponent overflow, exponent underflow, and floating-point divide -- are the same for the floating-point instructions with rounding options as for those without rounding options, except for a difference in the default value supplied when the CPU is disabled for exponent underflow.

### Default Result for Exponent Underflow

When the exponent-underflow mask bit is zero and exponent underflow occurs, a default value is placed in the result location, the value depending on the rounding mode and on the sign.

If the rounding mode is round up and the sign of the intermediate result is positive, or if the rounding mode is round down and the sign of the intermediate result is negative, the default result for exponent underflow is the normalized number with the smallest absolute value and the sign of the intermediate result ($\pm 16^{-65}$). The default result is a true zero if the rounding mode is round to zero, round to nearest, round up with a negative intermediate result, or round down with a positive intermediate result.

### Programming Note

The default result for exponent underflow ensures that the result is always equal to or greater than the exact result when rounding up and that it is always equal to or less than the exact result when rounding down.

## FLOATING-POINT ACCUMULATOR

A floating-point accumulator allows the exact sum of a great many products of floating-point numbers to be produced without errors due to rounding or truncation. Only at the end of a sequence of such operations is there a single rounding error when the accumulator contents are returned to a floating-point register in order to store the result or to use it in other floating-point operations. Thus, the scalar product of two vectors in storage, having elements in either the long or short floating-point format, may be computed exactly, by using the instruction MULTIPLY AND ACCUMULATE, regardless of the magnitude of the vector elements. The number of elements in each vector is limited only by the amount of storage available.

MULTIPLY AND ACCUMULATE is an interruptible instruction. Its execution proceeds for as many elements as are specified, unless interrupted. If an interruption occurs during execution, parameters associated with instruction execution will have been updated to the point of interruption. Unless the interruption indicates an unrecoverable error in the operation, instruction execution may be resumed from the point of interruption simply by reexecuting the instruction. (See also the section "Interruptible Instructions" in Chapter 5, "Program Execution," of the appropriate Principles of Operation.)

Other instructions are available to clear the accumulator, to add or subtract a single floating-point number to or from the accumulator, and to place the rounded accumulator contents in a floating-point register. The accumulator is represented as an area in storage, which is specified by the program. Thus, a program may have more than one accumulator, and instructions are provided to add or subtract the contents of two accumulators.

Instructions which access a storage area as an accumulator, as described in this document, are referred to here as accumulator instructions, so as to distinguish them from other instructions, which may access the same storage locations but not necessarily as an accumulator.

## VECTORS

The MULTIPLY AND ACCUMULATE instruction forms the scalar product S of two N-element vectors, A and B, in storage:

$$S = A(1)*B(1) + A(2)*B(2) + \ldots + A(N)*B(N)$$

where * indicates multiplication. A(1), A(2), ..., A(N) are the N elements of vector A, and B(1), B(2), ..., B(N) are the N elements of vector B. Each element is a floating-point number in either the long format (eight bytes) or short format (four bytes). Successive elements of a vector may be contiguous (in successive storage locations), or they may be separated by one or more element positions in storage.

After each pair of vector elements is multiplied and the product is added to the accumulator, the operation advances to the next pair of elements. The number of element positions in storage by which the operation advances for each vector element is called the stride of the vector. Thus, when the stride is specified as 3, the instruction accesses every third element position in storage. Advancing to the next element of a vector consists in adding an increment to the current element address. This address increment is the stride shifted left by three bit positions for the long format or two bit positions for the short format. For example, when the instruction specifies the long format and the stride is 10, the address is incremented by 80 after each element operation.

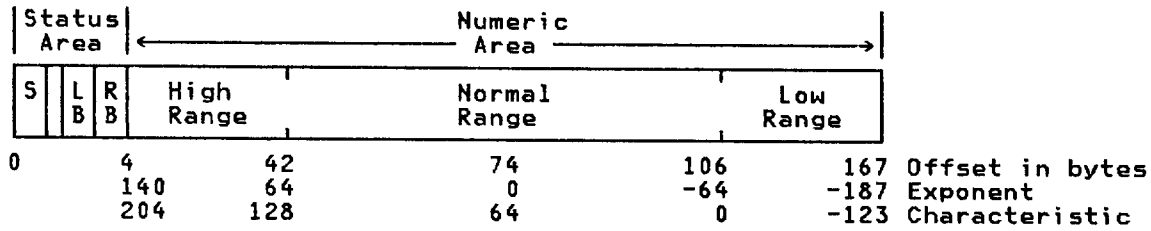
A stride of 1 means that the vector elements are contiguous. A stride of 0 causes the same vector element to be used repeatedly. The stride can be negative; a negative stride means that the operation proceeds backward through storage.

To illustrate, consider an N-by-N matrix that is stored in column order, the convention used for IBM System/370 FORTRAN programs. The elements of a column vector are contiguous, so that the column vector has a stride of 1. The elements of a row vector, however, are not contiguous, and a row vector has a stride of N. The vector of elements along the major diagonal of the matrix has a stride of N+1. All three types of vector contain N vector elements.

The MULTIPLY AND ACCUMULATE instruction requires up to six general registers, two of the register assignments being fixed. General register 1 is assigned to contain the location in storage of the accumulator to be used by the instruction to accumulate the products. General register 2 is assigned to contain N, a signed integer which specifies the number of elements to be processed. No elements are processed when N is zero or negative. The instruction itself specifies four general registers, which contain the starting address and the stride for each of the vector operands.

## ACCUMULATOR LAYOUT

A floating-point accumulator occupies a 168-byte storage area that is aligned on a 256-byte boundary. The layout of an accumulator in storage is as follows:




An accumulator consists of a four-byte status area on the left, followed by a 164-byte numeric area. When a floating-point number is added to the accumulator, the fraction is shifted to a position in the numeric area that corresponds to the characteristic; the fraction is added at that point, and any carries are propagated to the left as far as necessary.

Arithmetically, an accumulator may be considered to contain a fixed-point number of 328 hexadecimal digits and a separate sign, with the radix point located between the bytes at offsets 73 and 74 (that is, between the bytes at addresses 73 and 74 relative to the left end of the accumulator). If the floating-point number has an exponent of 0 (characteristic of 64), it is added such that the leftmost two fraction digits are added at offset 74; the remaining fraction digits are added to the right of offset 74. Numbers with a positive exponent are added further toward the left of the accumulator, those with a negative exponent further toward the right, the distance depending on the size of the exponent.

The specific relationship between the offset where the most significant fraction digit is added and the characteristic C of a floating-point number is F = (212 - C)/2. The integral part of F is the offset. If the fractional part of F is zero, the most significant fraction digit is added in the left digit at that byte location. If the fractional part of F is not zero, the right digit location is chosen.

The figure "Accumulator Mapping" illustrates in more detail the relationship between byte offset, exponent, and characteristic. The characteristic is shown in hexadecimal notation. (Negative characteristics or characteristics greater than 7F can occur only for intermediate products added by the MULTIPLY AND ACCUMULATE instruction.) The column at the right gives the order of magnitude of numbers whose most significant digit lies in the indicated area of an accumulator.

| Byte Offset | Left Digit | | Right Digit | | Number Range |
|---|---|---|---|---|---|
| | Exp (Dec) | Char (Hex) | Exp (Dec) | Char (Hex) | |
| 0-3 | Status area | | | | Accumulator overflow ($\geq 1/16*16^{141}$) |
| 4 | 140 | CC | 139 | CB | High range ($\geq 1/16*16^{64}$) |
| 5 | 138 | CA | 137 | C9 | |
| . | . | . | . | . | |
| . | . | . | . | . | |
| 41 | 66 | 82 | 65 | 81 | |
| 42 | 64 | 80 | 63 | 7F | Normal range ($\geq 1$) |
| 43 | 62 | 7E | 61 | 7D | |
| 44 | 60 | 7C | 59 | 7B | |
| . | . | . | . | . | |
| . | . | . | . | . | |
| 72 | 4 | 44 | 3 | 43 | |
| 73 | 2 | 42 | 1 | 41 | |
| 74 | 0 | 40 | -1 | 3F | Normal range ($<1$) |
| 75 | -2 | 3E | -3 | 3D | |
| . | . | . | . | . | |
| . | . | . | . | . | |
| 104 | -60 | 4 | -61 | 3 | |
| 105 | -62 | 2 | -63 | 1 | |
| 106 | -64 | 0 | -65 | -1 | Low range ($<1/16*16^{-64}$) |
| 107 | -66 | -2 | -67 | -3 | |
| . | . | . | . | . | |
| . | . | . | . | . | |
| 166 | -184 | -78 | -185 | -79 | |
| 167 | -186 | -7A | -187 | -7B | |

Accumulator Mapping

Negative numbers are represented in an accumulator in two's-complement form. Bit 0 of the status area contains the sign bit (S).

When adding large numbers, including the products of very large vector elements, the result may extend into the area labeled "high range" in the diagram. This area corresponds to results which, if placed in a floating-point register, would cause exponent overflow. No exponent overflow occurs, however, because the numeric area is large enough to allow any reasonable number of products of the largest representable floating-point numbers to be accumulated. The possibility of entering the area labeled "accumulator overflow" is extremely remote, except possibly as the result of *a program error*, as discussed in the section "Accumulator Overflow."

When adding the products of very small vector elements, the result may enter the area labeled "low range" in the diagram. The corresponding products would cause exponent underflow if placed in floating-point registers. Again, no exponent underflow occurs, because the numeric area is large enough on the right to accommodate the product of the smallest representable floating-point numbers. A zero value can occur only as the result of clearing an accumulator, of subtracting a number that is exactly equal in value to the accumulator contents, or of adding zero operands.

## Accumulator Status Area

The status area in the leftmost four bytes of an accumulator contains three fields: the sign (S), the left bound (LB), and the right bound (RB).

The sign (bit 0) is zero when the current accumulator value is equal to or greater than zero; it is one when the accumulator value is less than zero.

The left bound (bits 16-23) contains the offset of the leftmost byte of the part of the numeric area where valid numeric information is currently located.

The right bound (bits 24-31) contains the offset of the rightmost byte of the part of the numeric area where valid numeric information is currently located.

The previous value of bits 1-15 of the status area is ignored during execution of accumulator instructions. When an accumulator instruction is due to change any part of the status area, it sets bits 1-7 to zeros and leaves the value of bits 8-15 unpredictable, except that CLEAR ACCUMULATOR sets the entire status area to zeros.

Any digits in the numeric area to the left of the byte designated by the left bound and any digits to the right of the byte designated by the right bound are ignored by the accumulator instructions and do not affect the value of the current accumulator contents.

When LB is zero and RB is zero, the accumulator is considered cleared, and its value is zero, regardless of the contents of the numeric area and the sign bit. When an accumulator instruction, other than CLEAR ACCUMULATOR, produces a zero result in the accumulator, it depends on the model whether the bounds are set to zeros or whether the numeric area within nonzero bounds is set to zeros instead.

Logically, as far as the accumulator instructions are concerned, an accumulator with valid nonzero contents may be considered to have its left bound fixed at 4 and its right bound fixed at 167. The actual positions of the left and right bounds following an accumulator instruction are model-dependent. The bounds serve to improve performance by not requiring the machine to search the entire numeric area for significant digits. Thus, there may or may not be nonsignificant hexadecimal zero digits (if positive) or hexadecimal F digits (if negative) to the left of the byte containing the most significant digit, with the left bound being set accordingly. Likewise, there may or may not be nonsignificant zeros to the right of the byte containing the least significant digit, with the right bound being set accordingly.

During the execution of an accumulator instruction, other than CLEAR ACCUMULATOR, the machine may increase or decrease the current accumulator width by inserting or deleting nonsignificant digits at the left or right bounds. Any such boundary changes are model-dependent.

The left bound (LB) and right bound (RB) are subject to the following restrictions:

$$4 \leq LB \leq RB \leq 167$$
$$\text{or} \quad LB = RB = 0$$

If these conditions are not met when an accumulator instruction, other than CLEAR ACCUMULATOR, is executed, a specification exception is recognized.

## Accumulator Overflow

When the left bound is 4, accumulator overflow occurs when either:

- A carry from the leftmost digit of the byte at offset 4 enters the sign bit but no carry is propagated out of the sign bit, or
- No carry enters the sign bit, but a carry is propagated out of the sign bit.

Such an accumulator overflow occurs only when the result would be equal to or greater than $+16^{140}$ or less than $-16^{140}$.

When accumulator overflow occurs, the left bound in the status area is set to zero, but the numeric area retains the value it would have had if no overflow had occurred. Instruction execution is completed, and condition code 3 is set.

If accumulator overflow occurs during execution of MULTIPLY AND ACCUMULATE, the current unit of operation is completed with the described result, and no more element pairs are processed.

Because the left and right bounds no longer meet the restrictions specified above, executing a subsequent accumulator instruction, other than CLEAR ACCUMULATOR, causes a specification exception. CLEAR ACCUMULATOR clears the overflow condition by setting both bounds to zero.

## Storage-Operand Consistency

For all accumulator instructions, multiple accesses may be made to all or some of the bytes of an accumulator in storage.

Thus, unlike instructions for which single-access references are guaranteed, intermediate results of an accumulator instruction modifying any single byte location may be observed by channel programs and other CPU programs accessing the same location concurrently. (See the section "Storage-operand Consistency" in Chapter 5, "Program Execution," of the appropriate Principles of Operation.)

## Programming Notes

1. The accumulator contents are equivalent to a 1313-bit signed binary integer, whose sign bit has been

separated from the numeric bits. Its actual value is obtained by multiplying the integer contents by a scale factor of $16^{-188} = 2^{-752}$.

2. Although accumulator arithmetic is described as hexadecimal, it is indistinguishable from binary-integer arithmetic, except that the shifts needed to line up floating-point operand fractions and the accumulator contents occur only in multiples of four bits.

3. Clearing an accumulator by executing the instruction CLEAR ACCUMULATOR is logically equivalent to moving or storing a word of zeros into its status area. (CLEAR ACCUMULATOR also checks whether the accumulator address is on a 256-byte boundary and, depending on the model, it may or may not set the numeric area to zeros.)

4. A floating-point number may be loaded into the accumulator by first clearing the accumulator and then using the instruction ADD TO ACCUMULATOR.

5. The accumulator range is large enough that numerically meaningful operations cannot ordinarily cause the accumulator to overflow. Thus, when starting with a cleared accumulator, continuous use of the MULTIPLY AND ACCUMULATE instruction to accumulate repeatedly the product of the largest representable floating-point numbers would require $16^{14}$, or approximately $7 \times 10^{16}$, executions of the instruction before the accumulator would overflow. If, during this string of operations, the result is ever returned to a floating-point register, exponent overflow would occur long before accumulator overflow is reached.

   Two types of situations, which are most likely to arise from program errors, could cause accumulator overflow during program execution:

   - Accumulator instructions are used without first clearing the accumulator. Significant digits remaining in the leftmost part of the numeric area may cause a subsequent accumulator overflow.

   - The instructions ADD ACCUMULATOR TO ACCUMULATOR or SUBTRACT ACCUMULATOR FROM ACCUMULATOR are used repeatedly so as to allow the accumulator contents to grow indefinitely. This includes specifying the same accumulator for both operands, which doubles its contents each time.

6. The restrictions on the left and right bounds are not checked when executing instructions which address the accumulator as a storage operand. The use of such instructions, or not clearing the accumulator initially, may leave the accumulator in a state which causes a specification exception during the execution of subsequent accumulator instructions.

7. A PER event for storage alteration is recognized, and a program interruption occurs, whenever the CPU is enabled for such an event and execution of an accumulator instruction causes storing within the storage area designated by control registers 10 and 11. Such storing, and the resulting recognition of the PER storage-alteration event, may be specified in the definition of the instruction, or it may be the result of model-dependent action. For example, during execution of CLEAR ACCUMULATOR, a PER storage-alteration event is always recognized if the storage area designated by control registers 10 and 11 includes all or part of the accumulator status area, which the instruction sets to zero; a PER storage-alteration event may or may not be recognized if the designated storage area excludes the accumulator status area but includes all or part of the numeric area, because alteration of the numeric area is model-dependent.

## INSTRUCTIONS

The high-accuracy-arithmetic instructions and their mnemonics, formats, and operation codes are listed in the figure "Summary of High-Accuracy-Arithmetic Instructions. The figure also indicates when the condition code is set and the exceptional conditions in operand designations, data, or results that cause a program interruption.

In the instruction descriptions, the register fields are indicated as follows:

FR Floating-point register

GR General register containing a storage address

RT General register containing a stride

| Name | Mnemonic | Characteristics | | | | Op Code |
|---|---|---|---|---|---|---|
| ADD ACCUMULATOR TO ACCUM. | AACAC | RRE C HA | A SP | | ST | B2D8 |
| ADD TO ACCUMULATOR (long) | AACDR | RRE C HA | A SP | | ST | B2D0 |
| ADD TO ACCUMULATOR (short) | AACER | RRE C HA | A SP | | ST | B2D1 |
| ADD WITH ROUNDING (long) | ADRN | RRE C HA | SP | EU EO | | B2C0 |
| ADD WITH ROUNDING (short) | AERN | RRE C HA | SP | EU EO | | B2C1 |
| CLEAR ACCUMULATOR | CLAC | RRE HA | A SP | | ST | B2DA |
| DIVIDE WITH ROUNDING (long) | DDRN | RRE HA | SP | EU EO FK | | B2C6 |
| DIVIDE WITH ROUNDING (short) | DERN | RRE HA | SP | EU EO FK | | B2C7 |
| LOAD WITH ROUNDING (l. to s.) | LERN | RRE HA | SP | EU EO | | B2C8 |
| MULTIPLY AND ACCUMULATE (l.) | MACD | RRE C HA | A SP | II | R ST | B2D4 |
| MULTIPLY AND ACCUMULATE (s.) | MACE | RRE C HA | A SP | II | R ST | B2D5 |
| MULTIPLY WITH ROUNDING (long) | MDRN | RRE HA | SP | EU EO | | B2C4 |
| MULTIPLY WITH ROUNDING (s.) | MERN | RRE HA | SP | EU EO | | B2C5 |
| ROUND FROM ACCUMULATOR (long) | RACD | RRE C HA | A SP | EU EO | ST | B2D6 |
| ROUND FROM ACCUMULATOR (s.) | RACE | RRE C HA | A SP | EU EO | ST | B2D7 |
| SUBTRACT ACCUM. FROM ACCUM. | SACAC | RRE C HA | A SP | | ST | B2D9 |
| SUBTRACT FROM ACCUM. (long) | SACDR | RRE C HA | A SP | | ST | B2D2 |
| SUBTRACT FROM ACCUM. (short) | SACER | RRE C HA | A SP | | ST | B2D3 |
| SUBTRACT WITH ROUNDING (long) | SDRN | RRE C HA | SP | EU EO | | B2C2 |
| SUBTRACT WITH ROUNDING (s.) | SERN | RRE C HA | SP | EU EO | | B2C3 |

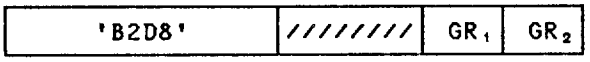
Explanation:

| | |
|---|---|
| A | Access exceptions for logical addresses |
| C | Condition code is set |
| EO | Exponent-overflow exception |
| EU | Exponent-underflow exception |
| FK | Floating-point-divide exception |
| HA | High-accuracy-arithmetic facility |
| II | Interruptible instruction |
| R | PER general-register-alteration event |
| RRE | RRE instruction format |
| SP | Specification exception |
| ST | PER storage-alteration event |

Summary of High-Accuracy-Arithmetic Instructions

ADD ACCUMULATOR TO ACCUMULATOR

AACAC $GR_1,GR_2$ [RRE]




The contents of the accumulator at the second-operand location are added to the contents of the accumulator at the first-operand location, and the sum is placed in the first-operand location. The value of the accumulator contents at the second-operand location remains unchanged.

The $GR_1$ and $GR_2$ fields designate general registers which contain the storage addresses of the two accumulators.

If either accumulator is not designated on a 256-byte boundary or if the left and right bounds in the status area of either accumulator do not satisfy the bound restrictions, a specification exception is recognized.

Resulting Condition Code:

0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 Accumulator overflow

Program Exceptions:

Access (fetch and store, operands 1 and 2)
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

Programming Note

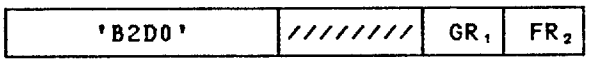
For this and other accumulator instructions where the second operand is fetched from an accumulator, it is

possible for a store-type access exception to occur and a PER storage-alteration event to be recognized, even though the accumulator is not the result target. Depending on the model, the machine may narrow the accumulator bounds and remove nonsignificant digits on the left or right while fetching the contents of the accumulator. This changes the accumulator contents in storage, but not its value.

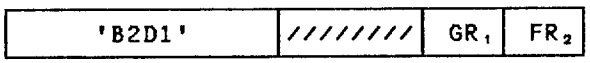
Consequently, accumulators should not be located in a protected area of storage.

ADD TO ACCUMULATOR

AACDR $GR_1,FR_2$ [RRE, Long Operands]




AACER $GR_1,FR_2$ [RRE, Short Operands]




The second operand is a floating-point number which is added to the accumulator at the first-operand location.

The $GR_1$ field designates a general register which contains the storage address of the accumulator. The $FR_2$ field designates a floating-point register which contains the second operand.

The $FR_2$ field must designate floating-point registers 0, 2, 4, or 6; the accumulator must be designated on a 256-byte boundary; and the left and right bounds in the status area of the accumulator must satisfy the bound restrictions. Otherwise, a specification exception is recognized.

Resulting Condition Code:

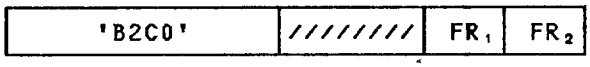
0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 Accumulator overflow

Program Exceptions:

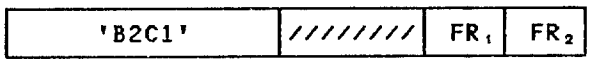
Access (fetch and store, operand 1)
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

ADD WITH ROUNDING

ADRN $FR_1,FR_2$ [RRE, Long Operands]




AERN $FR_1,FR_2$ [RRE, Short Operands]




The second operand is added to the first operand. The normalized and rounded sum is placed in the first-operand location.

Addition of two floating-point numbers consists in characteristic comparison, fraction alignment, signed fraction addition, and rounding. Nonzero operands are first normalized to eliminate any leftmost hexadecimal zero digits, and operands with a zero fraction are replaced by true zeros.

Both fractions are extended on the right with zeros in the guard-digit, rounding-digit, and sticky-bit positions. The characteristics of the two normalized operands are compared. The larger characteristic is used as the characteristic of an intermediate sum. The fraction accompanying the smaller characteristic is aligned with the other fraction by right shifts, its characteristic being increased by one for each hexadecimal digit of shift, until the characteristics are equal. Each digit shifted out of the rightmost digit position of the fraction enters the guard-digit position, each digit shifted out of the guard-digit position enters the rounding-digit position, and all bits shifted out of the rounding-digit position are ORed into the sticky bit. The right-extended fractions with signs are then added algebraically to form the right-extended fraction of the intermediate sum.

If the fraction addition produces a carry out of the leftmost hexadecimal digit of the intermediate-sum fraction, the fraction is shifted right one digit position, the new leftmost hexadecimal digit of the fraction is set to one, and the characteristic is increased by one. The digit shifted out of the rightmost digit position of the fraction enters the guard-digit position, the digit shifted out of the guard-digit position enters the rounding-digit position, and the bits shifted out of the rounding-digit position are ORed into the sticky bit.

If one or more of the leftmost hexadecimal digits of the right-extended

intermediate-sum fraction are zeros, but not all of its digits are zeros, the fraction is shifted left until the leftmost digit is nonzero. The guard digit moves into the rightmost fraction-digit position, the rounding digit moves into the guard-digit position, and zeros are placed in the rounding-digit position. The characteristic is reduced by the number of hexadecimal digits of shift.

The intermediate-sum fraction is then rounded to 14 (ADRN) or six (AERN) hexadecimal digits. Rounding is performed according to the rounding mode specified in general register 0. If rounding produces a carry out of the leftmost hexadecimal digit of the sum fraction, the rounded fraction is shifted right one digit position, the new leftmost hexadecimal digit of the fraction is set to one, and the characteristic is increased by one. The excess digits to the right of the rounded fraction are discarded.

If all digits of the rounded result fraction are zeros, the result is made a true zero.

If the rounded result fraction is not zero, the sign of the result is determined by the rules of algebra.

An exponent-overflow exception is recognized when the correct characteristic of the final result would exceed 127. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for exponent overflow takes place. The result sign and fraction remain correct.

An exponent-underflow exception is recognized when the correct characteristic of a nonzero final result would be less than zero. If the exponent-underflow mask bit is one, the operation is completed by making the result characteristic 128 greater than the correct value. The result sign and fraction remain correct, and a program interruption for exponent underflow takes place. When exponent underflow occurs and the exponent-underflow mask bit is zero, a program interruption does not take place; instead, the operation is completed by placing the default result for the rounding mode in the first-operand location.

The $FR_1$ and $FR_2$ fields must designate floating-point registers 0, 2, 4, or 6, and bits 0-29 of general register 0 must contain zeros. Otherwise, a specification exception is recognized.

Resulting Condition Code:

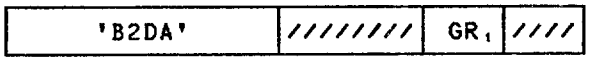
0 Result fraction is zero
1 Result is less than zero
2 Result is greater than zero
3 --

Program Exceptions:

Exponent overflow
Exponent underflow
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

## CLEAR ACCUMULATOR

CLAC $GR_1$ [RRE]




The accumulator at the first-operand location is cleared.

Clearing the accumulator consists in setting the word at the specified storage address, which is the status area of the accumulator, to zeros. The contents of the numeric area of the accumulator are unpredictable; the numeric area may or may not be set to zeros, depending on the model.

The $GR_1$ field designates a general register which contains the storage address of the accumulator. The accumulator must be designated on a 256-byte boundary; otherwise, a specification exception is recognized.

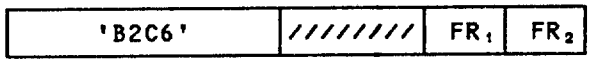
Condition Code: The code remains unchanged.

Program Exceptions:

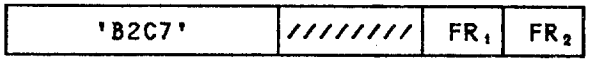
Access (store, operand 1)
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

## DIVIDE WITH ROUNDING

DDRN $FR_1,FR_2$ [RRE, Long Operands]




DERN $FR_1,FR_2$ [RRE, Short Operands]




The first operand (the dividend) is divided by the second operand (the divi-

sor). The normalized and rounded quotient is placed in the first-operand location. No remainder is preserved.

The operation is performed the same as for DIVIDE (DDR or DER), except that, after a right shift, if any, of the intermediate-quotient fraction has been performed, the fraction is rounded according to the rounding mode specified in general register 0.

An exponent-overflow or exponent-underflow exception can be recognized during the division operation as for DIVIDE (DDR or DER). The default result for exponent underflow for DDRN and DERN differs from that produced for DDR and DER.

The $FR_1$ and $FR_2$ fields must designate floating-point registers 0, 2, 4, or 6, and bits 0-29 of general register 0 must contain zeros. Otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

Exponent overflow
Exponent underflow
Floating-point divide
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

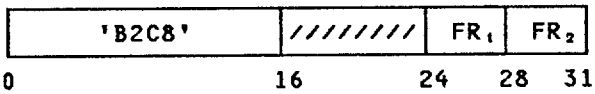
Programming Notes

1. When the rounding mode is round to zero, DIVIDE WITH ROUNDING (DDRN or DERN) produces the same result as DIVIDE (DDR or DER).

2. The operand values which cause exponent overflow or exponent underflow are the same for DIVIDE WITH ROUNDING and DIVIDE. These exceptions can occur only during the division. No operand values can cause rounding of the intermediate-quotient fraction to produce a carry out of the leftmost hexadecimal fraction digit; therefore, rounding does not change the result characteristic.

LOAD WITH ROUNDING

LERN $FR_1,FR_2$
[RRE, Long Operand 2, Short Operand 1]




The second operand is rounded from the long format to the short format, and the result is placed in the first-operand location.

If the second operand has a zero fraction, the result is a true zero. A nonzero operand is first normalized to eliminate any leftmost hexadecimal zero digits. The fraction is then rounded to six hexadecimal digits according to the rounding mode specified in general register 0. If rounding produces a carry out of the leftmost hexadecimal digit of the fraction, the rounded fraction is shifted right one digit position, the new leftmost hexadecimal digit of the fraction is set to one, and the characteristic is increased by one. The excess digits to the right of the rounded fraction are discarded.

The sign of a nonzero result is the same as the sign of the second operand.

An exponent-overflow exception is recognized when the correct characteristic of the final result would exceed 127. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for exponent overflow takes place. The result sign and fraction remain correct.

An exponent-underflow exception is recognized when the correct characteristic of a nonzero final result would be less than zero. If the exponent-underflow mask bit is one, the operation is completed by making the result characteristic 128 greater than the correct value. The result sign and fraction remain correct, and a program interruption for exponent underflow takes place. When exponent underflow occurs and the exponent-underflow mask bit is zero, a program interruption does not take place; instead, the operation is completed by placing the default result for the rounding mode in the first-operand location.

The $FR_1$ and $FR_2$ fields must designate floating-point registers 0, 2, 4, or 6, and bits 0-29 of general register 0 must contain zeros. Otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

Exponent overflow
Exponent underflow
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

Programming Note

When the rounding mode is round to nearest and the second operand is normalized or a true zero, LOAD WITH ROUNDING (LERN) produces the same result as LOAD ROUNDED (LRER), except when the right half of the second operand is 80 00 00 00 in hexadecimal notation, in which case bit 31 of the result is set to zero after the carry has been propagated.

When the rounding mode is round to zero and the second operand is normalized or a true zero, LERN produces the same result as LOAD (LER).

MULTIPLY AND ACCUMULATE

MACD $GR_1(RT_1),GR_2(RT_2)$
[RRE, Long Operands]


MACE $GR_1(RT_1),GR_2(RT_2)$
[RRE, Short Operands]


The products of corresponding floating-point elements of two vectors at the first- and second-operand locations are added to the contents of the accumulator at the location specified by general register 1. The length of each vector is specified by general register 2.

The $GR_1$ and $GR_2$ fields designate general registers containing the addresses in storage of the first- and second-operand vectors, which are the addresses of the first elements to be multiplied. The $RT_1$ and $RT_2$ fields designate general registers containing the strides for the first and second operands, which are used to obtain the addresses of subsequent operand elements. Each stride is a 32-bit signed binary integer, which is changed to an address increment by shifting it left by three bits (MACD) or two bits (MACE); any bits shifted out of bit position 0 are ignored, and vacated rightmost bit positions are filled with zeros. The general registers containing the strides remain unchanged. After a pair of elements has been processed, the address increment is added to the general register containing the corresponding vector address, carries out of bit position 0 being ignored. The updated address is used to fetch the next element of that vector.

If the $RT_1$ or $RT_2$ field of the instruction is zero, general register 0 is not used. Instead, a stride of 1 is assumed for the corresponding vector, and its elements are fetched from contiguous storage locations.

If the $GR_1$ and $GR_2$ fields designate the same general register, the same vector is used for both operands. Each vector element is fetched only once, and the address register is updated only once, the $RT_1$ field being used to specify the stride. The $RT_2$ field is ignored.

General register 1 contains the storage address of the accumulator. The accumulator must be designated on a 256-byte boundary, and the left and right bounds in the status area of the accumulator must satisfy the bound restrictions; otherwise, a specification exception is recognized.

General register 2 contains the number of elements in each vector operand that are to be processed. The number is a 32-bit signed binary integer.

Instruction execution consists in a repetition of the following four steps for each pair of vector elements.

1. If general register 2 contains a number equal to or less than zero, condition code 0, 1, or 2 is set, depending on whether the accumulator contents are equal to, less than, or greater than zero, and execution is completed. Otherwise, instruction execution continues with step 2.

2. The exact 28-digit (MACD) or 12-digit (MACE) product of the fractions of each pair of operand elements is added algebraically to the accumulator contents, taking into account the product sign, as determined from the operand signs by the rules of algebra, and the current accumulator sign. The accumulator position at which the fraction product is added is determined by the sum of the two operand characteristics.

3. The address in the general register designated by $GR_1$ and, if the $GR_2$ field is not equal to the $GR_1$ field, the address in the general register designated by $GR_2$, is increased by the corresponding address increment; if the $GR_1$ and

$GR_2$ fields are equal, the address is increased only once. The contents of general register 2 are decreased by one.

4. If accumulator overflow occurs, condition code 3 is set, and instruction execution is completed. Otherwise, instruction execution continues with step 1.

MULTIPLY AND ACCUMULATE is an interruptible instruction. A unit of operation consists of one or more repetitions of the above four steps, and the point of interruption occurs after step 4. If execution is interrupted, the general registers designated by $GR_1$ and $GR_2$ contain the addresses of the next elements to be processed, general register 2 contains the number of elements remaining to be processed, and the condition code is unpredictable.

When instruction execution is completed after processing one or more element pairs, the general registers designated by $GR_1$ and $GR_2$ contain the addresses of what would have been the next pair of elements to be processed if execution had continued. General register 2 contains zero, unless execution was ended prematurely because of accumulator overflow. When execution is completed without processing any elements, the general registers remain unchanged.

When an operand vector overlaps the accumulator in storage, the result is unpredictable.

The $GR_1$ and $GR_2$ fields should not designate general registers 1 or 2. The $RT_1$ field and, if the $GR_2$ field is not equal to the $GR_1$ field, the $RT_2$ field should not designate general registers 1 and 2, nor should they designate the same general register as either the $GR_1$ or $GR_2$ field. Otherwise, the result of executing the instruction is unpredictable.

Resulting Condition Code:

0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 Accumulator overflow

Program Exceptions:

Access (fetch, operands 1 and 2; fetch and store, accumulator)
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

Programming Notes

1. Care should be taken to avoid meaningless register assignments. Thus, the $GR_1$ and $GR_2$ fields of the instruction should not designate general register 1, which contains the accumulator address, or general register 2, which contains the number of elements. Likewise, the $RT_1$ field and, if $GR_1$ is not equal to $GR_2$, the $RT_2$ field should not designate either general register 1 or 2, nor should they specify the same general register as either the $GR_1$ or $GR_2$ field. The effect of any such assignment is unpredictable, because it depends on the model whether these operand parameters are fetched just once at the start of execution or whether they are refetched during execution. Refetching may occur, even when the value has not changed, because of an interruption or without an interruption having taken place, and the contents of general registers that are due to be updated may or may not have been updated at the time. Also, the result may not be repeatable.

   General register 0 may be used as a vector-address register. At the same time, zero may be specified for $RT_1$ or $RT_2$ because no general register is then designated for a stride.

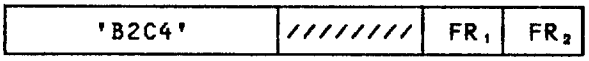
2. The stride may have either sign. A positive stride causes the address to be incremented. A negative stride causes the address to be decremented.

   The range of values for the stride is $+(2^{28} - 1)$ to $-2^{28}$ in the long format and $+(2^{29} - 1)$ to $-2^{29}$ in the short format. No warning is given for extremely large numbers outside of this range if significant bits are lost during the left shift which changes the stride to an address increment.

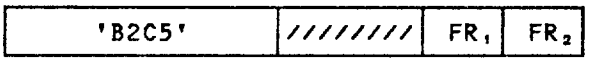
3. No elements are processed, and all general registers remain unchanged, when the number of elements specified by general register 2 is zero or negative.

## MULTIPLY WITH ROUNDING

MDRN $FR_1,FR_2$ [RRE, Long Operands]




MERN $FR_1,FR_2$ [RRE, Short Operands]




The normalized and rounded product of the second operand (the multiplier) and the first operand (the multiplicand) is placed in the first-operand location.

The operation is performed the same as for MULTIPLY (MDR or MER), except that after a left shift, if any, of the intermediate-product fraction has been performed, the fraction is rounded to the long (MDRN) or short (MERN) format according to the rounding mode specified in general register 0. If rounding produces a carry out of the leftmost hexadecimal digit of the product fraction, the rounded fraction is shifted right one digit position, the new leftmost hexadecimal digit of the fraction is set to one, and the characteristic is increased by one. The excess digits to the right of the rounded fraction are discarded.

An exponent-overflow or exponent-underflow exception may be recognized during the multiplication operation, as for MULTIPLY (MDR or MER). The default result for exponent underflow in MDRN and MERN differs from the result produced by MDR and MER. An exponent-overflow exception is also recognized during the rounding operation if a carry would cause the characteristic of the final result to exceed 127.

The $FR_1$ and $FR_2$ fields must designate floating-point registers 0, 2, 4, or 6, and bits 0-29 of general register 0 must contain zeros. Otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

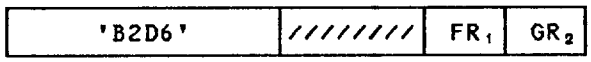
- Exponent overflow
- Exponent underflow
- Operation (if the high-accuracy-arithmetic facility is not installed)
- Specification

Programming Note

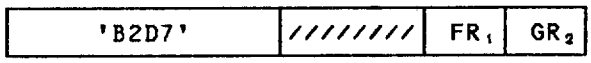
MULTIPLY WITH ROUNDING (MERN) differs from MULTIPLY (MER) in that the result is a rounded product in the short format instead of an exact product in the long format. When MERN is used while the rounding mode is round to zero, the result in the left half of the target register is the same as when MER is used on the same operands, but the right half remains unchanged, as for all results in the short format.

## ROUND FROM ACCUMULATOR

RACD $FR_1,GR_2$ [RRE, Long Operands]




RACE $FR_1,GR_2$ [RRE, Short Operands]




The contents of the accumulator at the second-operand location are converted to a normalized and rounded floating-point number, which is placed in the first-operand location.

The $FR_1$ field designates the floating-point register which receives the result. The $GR_2$ field designates a general register which contains the storage address of the accumulator. The value of the accumulator contents remains unchanged.

The $FR_1$ field must designate floating-point registers 0, 2, 4, or 6; the accumulator must be designated on a 256-byte boundary; bits 0-29 of general register 0 must contain zeros; and the left and right bounds in the status area of the accumulator must satisfy the bound restrictions. Otherwise, a specification exception is recognized.

If the accumulator contents are zero, a true zero is placed in the first-operand location. If the accumulator contents are nonzero and the value can be represented exactly as a normalized floating-point number in the long (RACD) or short (RACE) format, that value is placed in the first-operand location. Otherwise, the accumulator contents are rounded to the long or short format by choosing one of the two normalized floating-point numbers in that format which are nearest in value to the accumulator contents. Which of the two neighboring values is

chosen depends on the rounding mode specified by general register 0.

An exponent-overflow exception is recognized when the characteristic of the normalized and rounded result would exceed 127. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for exponent overflow occurs. The result sign and fraction are correct.

An exponent-underflow exception is recognized when the characteristic of the normalized and rounded nonzero result would be less than zero. If the exponent-underflow mask bit is one, the operation is completed by making the result characteristic 128 greater than the correct value, and a program interruption for exponent underflow takes place; the result sign and fraction are correct. If the exponent-underflow mask bit is zero, a program interruption does not occur; instead, the operation is completed by placing the default result for the rounding mode in the first-operand location.

Resulting Condition Code:

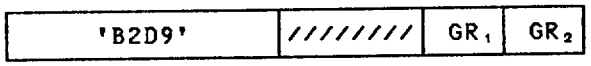
0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 --

Program Exceptions:

Access (fetch and store, operand 2)
Exponent overflow
Exponent underflow
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

## SUBTRACT ACCUMULATOR FROM ACCUMULATOR

SACAC $GR_1,GR_2$ [RRE]




The contents of the accumulator at the second-operand location are subtracted from the contents of the accumulator at the first-operand location, and the difference is placed in the first-operand location.

The operation is performed the same as for ADD ACCUMULATOR TO ACCUMULATOR, except that the two's-complement of the contents of the accumulator specified by the second operand is added.

If either accumulator is not designated on a 256-byte boundary, or if the left and right bounds in the status area of either accumulator do not satisfy the bound restrictions, a specification exception is recognized.

Resulting Condition Code:

0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 Accumulator overflow

Program Exceptions:

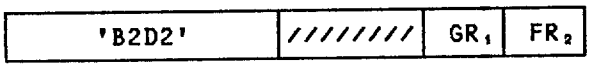
Access (fetch and store, operands 1 and 2)
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

Programming Note

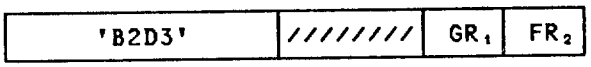
When the $GR_1$ and $GR_2$ fields are the same, the result is zero.

## SUBTRACT FROM ACCUMULATOR

SACDR $GR_1,FR_2$ [RRE, Long Operands]




SACER $GR_1,FR_2$ [RRE, Short Operands]




The second operand is a floating-point number which is subtracted from the accumulator at the first-operand location.

The operation is performed the same as for ADD TO ACCUMULATOR, except that the second operand participates in the operation with its sign bit inverted.

The $FR_2$ field must designate floating-point registers 0, 2, 4, or 6; the accumulator must be designated on a 256-byte boundary; and the left and right bounds in the status area of the accumulator must satisfy the bound restrictions. Otherwise, a specification exception is recognized.

Resulting Condition Code:

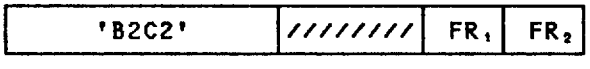
0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 Accumulator overflow

Program Exceptions:

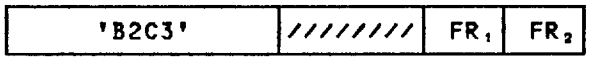
Access (fetch and store, operand 1)
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

SUBTRACT WITH ROUNDING

SDRN $FR_1,FR_2$ [RRE, Long Operands]




SERN $FR_1,FR_2$ [RRE, Short Operands]




The second operand is subtracted from the first operand, and the normalized and rounded difference is placed in the first-operand location.

The operation is performed the same as for ADD WITH ROUNDING, except that the second operand participates in the operation with its sign bit inverted.

The $FR_1$ and $FR_2$ fields must designate floating-point registers 0, 2, 4, or 6, and bits 0-29 of general register 0 must contain zeros. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Result fraction is zero
1 Result is less than zero
2 Result is greater than zero
3 --

Program Exceptions:

Exponent overflow
Exponent underflow
Operation (if the high-accuracy-arithmetic facility is not installed)
Specification

This page is intentionally left blank.

IBM System/370 RPQ High Accuracy Arithmetic

Order No. SA22-7093-0

**READER'S COMMENT FORM**

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

**Note:** *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

*Note:* Staples can cause problems with automated mail sorting equipment. Please use pressure-sensitive or other gummed tape to seal this form.

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the front cover or title page.)

SA22-7093-0

## Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department B98
P.O. Box 390
Poughkeepsie, New York 12602

Fold and tape | Please Do Not Staple | Fold and tape

IBM

Cut or Fold Along Line

IBM System/370 RPQ High Accuracy Arithmetic (File No. S370-01) Printed in U.S.A. SA22-7093-0