SYSTEM/32

IBM System/32
Scientific Macroinstructions
Functions Reference Manual

IBM System/32
Scientific Macroinstructions
Programming Information

SA21-9274
File No. S32-36

Program Number
5725-SC1

IBM System/32
Scientific Macroinstructions
Functions Reference Manual

Preface

This reference manual is for computer programmers, systems analysts, system engineers, and other technical people who are interested in the operation and characteristics of the IBM System/32 scientific macroinstructions at the machine code level.

This publication contains:

- Introductory information regarding instruction and data formats, addressing, and registers
- A description of the linkage and support macroinstructions
- A description of the arithmetic macroinstructions

Related Publications

- *IBM System/32 System Control Programming Reference Manual,* GC21-7593
- *IBM System/32 Functions Reference Manual,* GA21-9176
- *IBM System/32 Basic Assembler and Macro Processor Reference Manual,* GC21-7673
- *IBM System/32 Control Storage Logic Manual,* SY21-0533

Titles and abstracts of other related publications are listed in the *IBM System/32 Bibliography,* GC20-0032.

First Edition (May 1977)

This edition applies to Version 06 of the IBM System/32 System Control Program, Program Number 5725-SC1, and to all subsequent versions and modifications until otherwise indicated in new editions or technical newsletters. Changes are continually made to the specifications herein; before using this publication in connection with the operation of IBM systems, refer to the latest *IBM System/32 Bibliography*, GC20-0032, for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A Reader's Comment Form is at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901. Comments become the property of IBM.

Contents

Chapter 1. Introduction

To support FORTRAN on the IBM System/32 there is a scientific instruction set. These instructions are used by FORTRAN and require no programmer action other than writing FORTRAN language statements.

As an added capability you can use a subset of the scientific instruction set with the IBM System/32 Basic Assembler and Macro Processor. To do this you must have Feature 1500, which is the control storage increment and scientific microcode required by FORTRAN, and install the scientific macroinstructions at system generation time.

To use these scientific instructions with the assembler, the programmer must code a series of macroinstructions. These macroinstructions generate the scientific instructions that perform the specific functions necessary for scientific calculations. This expands the assembler user's ability to handle add, subtract, multiply, and divide binary data, and floating-point data.

There are three activities that the programmer must perform to use the scientific instructions with the assembler program.

1. Establish the required environment using the following macroinstructions:
   a. $CSET
   b. $CNTR

2. Establish the interface between the assembler and the scientific microcode. This capability is provided by:
   a. $CALL
   b. $INVK
   c. $CSUB

3. Code the necessary macroinstructions to generate the scientific instructions to solve your particular requirements.

The remainder of this manual describes each of these steps and provides you with the information necessary to use scientific macroinstructions with the IBM System/32 Basic Assembler and Macro Processor. The terms *System/32 mode* and *scientific mode* are used throughout this manual to describe which processor executes a series of instructions or subroutine.

When the system is executing the instructions, described in the *System/32 Functions Reference Manual*, GC21-9176, using the system control program it is in System/32 mode and scientific macroinstructions are invalid. Likewise, when the system is executing the instructions described in this manual using the scientific microcode, it is in scientific mode and System/32 instructions are invalid.

SCIENTIFIC MACROINSTRUCTION STATEMENTS

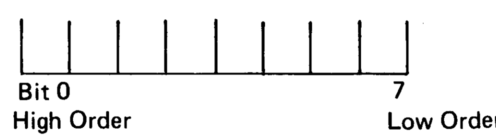
Scientific macroinstructions are symbolic source statements that are expanded into a predetermined sequence of object code by the IBM System/32 Assembler and Macro Processor, then executed by the scientific microcode. The format of a scientific macroinstruction is:

[Label]	$BADD	address[,I]
Name Entry	Operation Code	Operand Address

DATA FORMATS

Data resides in main storage in 8-bit bytes. The instruction the system is executing determines how the data is interpreted.

In any instruction, data is represented as a positive or negative number by the value at bit 0. If bit 0 is 0 the data is positive. If bit 0 is 1 the data is negative.



Binary Format

Binary data is recorded in a 2-byte or 4-byte format. Both formats use bit 0 as a sign bit followed by the integer field. Positive numbers are represented in true binary notation. Negative numbers are represented in twos-complement notation. Twos-complement notation does not include negative zero.

The following is an example of the hexadecimal number 5EB3 written as a positive number in true binary notation:

0101 1110 1011 0011

This is an example of the same hexadecimal number, 5EB3, written as a negative number in twos-complement notation:

1010 0001 0100 1101

Floating-Point Format

Floating-point data is recorded in either single-precision or double-precision format. Both formats use bit 0 as the sign bit of the mantissa followed by the characteristic, in excess 64 notation, in bits 1-7. Single-precision data contains the mantissa in bits 8-31, while double-precision data contains the mantissa in bits 8-63.

ADDRESSING

Main storage is addressed in binary; source programs and program listings customarily use hexadecimal notation to represent these binary addresses. Main storage positions are numbered consecutively from hex 0000 to the upper limit of storage. Storage locations are specified by the address of the leftmost byte in the field.

An address that is used to refer to main storage can be specified by either of two methods: direct addressing or indexed addressing.

Direct Addressing

When direct addressing is used, the effective address (actual storage location of data) is taken from the instruction. The address in the instruction is 2 bytes long.

For example, if you were to code the statement:

```
NAMEA    $BLD    FIELDA
```

If FIELDA equals storage location 0013, then the 4 bytes of data at locations 0013 through 0016 are placed in the binary register.

Indexed Addressing

Addresses in most scientific instructions can be indexed. If an address is indexed, the *effective address* used by the instruction is the sum of the current contents of the scientific index register and the contents of the address portion of the instruction.

For example, if you were to code the statement:

```
NAMEA    $BLD    FIELDA,I
```

If FIELDA equals storage location 0013 and the index register contains 0005, then the 4 bytes of data at locations 0018 through 001B are placed in the binary register.

MACHINE INSTRUCTION FORMAT

All of the System/32 scientific instructions are 3 bytes long. They are composed of a 1-byte op code and either a 2-byte address or a 2-byte data field. Bits 0-6 of the op code specify the instructions, and bit 7 denotes the type of addressing to be used: 0 = direct addressing, 1 = indexed addressing.

Some macroinstructions using this format are named according to the data type, data length, and the operation to be performed. These instructions are:

Instruction	Index Multiplier (M)	Index (X)	Integer*2 (H)	Integer*4 (B)	Real*4 (R)	Real*8 (D)	Address (A)
Load (LD)		$XLD	$HLD	$BLD	$RLD	$DLD	$ALD
Store (ST)	$MST	$XST	$HST	$BST	$RST	$DST	
Add (ADD)		$XADD	$HADD	$BADD	$RADD	$DADD	
Subtract (SUB)			$HSUB	$BSUB	$RSUB	$DSUB	
Multiply (MLT)		$XMLT	$HMLT	$BMLT	$RMLT	$DMLT	
Divide (DIV)			$HDIV	$BDIV	$RDIV	$DDIV	
Compare (CMP)			$HCMP	$BCMP	$RCMP	$DCMP	
Load Immediate (LI)	$MLI	$XLI					$ALI
And (AND)				$BAND			
Or (OR)				$BOR			
Not (NOT)				$BNOT			
Multiply and Add (MTA)		$XMTA					
If (IF)				$BIF	$RIF		

Other macroinstructions are named according to their function. These instructions are:

Instruction	Function
$GOTO	Changes the execution sequence to the instruction at the effective address
$INVK	Changes to System/32 mode execution beginning at the effective address
$LSET	Sets the binary register according to the condition code register contents and the instruction mask
$CALL	Executes the scientific subroutine
$CEQU	Generates labels required to gain access to the scientific communication area
$CSET	Loads the scientific microcode
$CNTR	Enters the scientific microcode
$CSUB	Starts the scientific subroutine
$CRTN	Exits the scientific subroutine

REGISTERS

Scientific mode registers that are directly accessible by the scientific instructions are the index register, the index multiplier register, the binary register, the floating-point register, the address register, and the condition code register. All of these registers are in control storage and can be referenced only through the use of scientific instructions.

The *index register* is used in indexed instructions to compute the effective address. The index register is a 2-byte register that contains the index value for indexed addressing.

The *index multiplier register* is a 2-byte register used in computing the value to be placed in the index register. The $XMTA and $XMLT instructions cause the product of the index multiplier register and the instruction operand to be either added to or placed in the index register.

The *binary register* is a 4-byte register that contains twos-complement binary numbers. It is used for integer arithmetic. For integer*2 (H) operations, the operand is copied to temporary storage and extended on the left with the sign bit to make a 4-byte value; the result is used as the actual operand for the instruction. The exception to this is the HST instruction, which stores the 2 low-order bytes of the register with no consideration for sign or truncation.

The *floating-point register* consists of an 8-byte floating-point value. Associated with the floating-point register are a guard digit during computation and a status indicator for single- or double-precision. Function and resulting status vary according to the operand type (R,D) and the status. All floating-point operations, except load and store, have normalized results; meaning that the high-order hexadecimal digit of the mantissa is nonzero.

The floating-point register status is set to double-precision whenever a single- or double-precision operation is performed (except for RLD) and the prior status was double-precision. The status is set to single-precision by an RLD instruction and remains single-precision as long as only single-precision operations are performed. If the status is double-precision and the operation is single-precision the operand is extended to double-precision and the operation is carried out as double-precision. If characteristic overflow or underflow occurs, the appropriate indicator is set in the scientific communication area.

The *address register* is a 2-byte register used in conjunction with the $INVK (invoke) instruction. Parameters or values used by System/32 mode instructions are addressed via the address register. When the $INVK instruction is executed, the contents of the address register are placed in XR2 (index register 2). XR2 can then be used by the System/32 mode instructions to locate and gain access to the parameters or values in main storage.

The *condition code register* is a 1-byte register that contains the results of a compare operation. The register is set to low, equal, or high by a compare instruction. $LSET (test condition) is the only instruction provided to test the contents of the condition code register.

Chapter 2. Scientific Mode Linkage And Support Macroinstructions

SCIENTIFIC ENVIRONMENT AND SUBROUTINE LINKAGE

Scientific mode linkage and support macroinstructions provide the interface from the System/32 mode routines to scientific routines, between scientific routines, and from scientific routines to System/32 mode routines.

The general environment for scientific mode processing is that System/32 XR1 (index register 1) addresses the scientific communication area and System/32 XR2 (index register 2) addresses the current save area for the executing scientific program.

Note: For detailed information regarding the scientific communication area, see the *IBM System/32 Control Storage Logic Manual*, SY21-0533.

The subroutine linkage in the scientific macro package is implemented via the $CSET, $CNTR, $CALL, $CSUB, and $CRTN macroinstructions. $CSET loads the scientific microcode and establishes the environment for scientific mode processing. $CNTR switches to scientific mode. $CALL generates the linkage to scientific subroutines, and passes required arguments. $CSUB establishes the subroutine environment and makes the received parameters available within the subroutine. $CRTN returns execution control to the calling routine. In the scientific subroutine linkage conventions, called subroutines must be external, separately assembled programs.

Variables passed to this subroutine can be accessed by indexing within the subroutine. The index value (parameter address) is in variables generated by the $CSUB macroinstruction. These variables are named $ARGnn, where nn represents the position of the desired variable within the parameter list. Figure 1 illustrates the subroutine linkage.

EXECUTE SCIENTIFIC SUBROUTINE ($CALL)

[Label] $CALL name(,address . . .)

This instruction causes the external scientific subroutine specified by *name* to be executed using variables at the specified addresses as parameters. When the subroutine completes execution, standard calling discipline resumes execution with the scientific macroinstruction following the $CALL macroinstruction.

RETURN TO SYSTEM/32 MODE ($INVK)

[Label] $INVK address

This instruction transfers the program to System/32 mode and continues execution with the System/32 instruction at the effective address.

LOAD SCIENTIFIC MICROCODE ($CSET)

[Label] $CSET

The $CSET macroinstruction generates code necessary to load the scientific microcode. The expansion includes the scientific communications area and the main save area. The $CSET macroinstruction is used only once in the program.

If you will need to use the data in XR1 or XR2 at a later time, you should save the contents of the registers before issuing the $CSET macroinstruction.

Note: If $CSET is unable to locate and load the microcode, control is passed to $MODERR. $MODERR must be a customer defined error recovery subroutine, failure to do so results in an assembly error.

ENTER SCIENTIFIC MICROCODE ($CNTR)

[Label] $CNTR

The $CNTR macroinstruction generates code necessary to initialize the environment for, and to enter, scientific mode.

If you will need to use the data in XR1 or XR2 at a later time, you should save the contents of the register before issuing the $CNTR macroinstruction.

GENERATE SCIENTIFIC LABELS ($CEQU)

[Label] $CEQU

The $CEQU macroinstruction generates labels necessary to allow access to the scientific communication area.

START SCIENTIFIC SUBROUTINE ($CSUB)

[Label] $CSUB number

This instruction generates code necessary to establish receiving subroutine linkage. The label specifies the entry point name. The number specifies the number of parameters to be received by the subroutine.

EXIT SCIENTIFIC SUBROUTINE ($CRTN)

[Label] $CRTN

This instruction generates code necessary to return control from a scientific subroutine.

Note: A macroinstruction has not been provided that allows the user to issue the $INVK macroinstruction then return to the microcode loaded into the control storage increment without destroying the environment that was established by the $CSET and $CNTR macroinstructions. However, it is possible to issue the $INVK macroinstruction then return to the previously established environment using the XFER instruction described in the *IBM System/32 Functions Reference Manual,* GC21-9176, and the *IBM System/32 Control Storage Logic Manual,* SY21-0533.

```
TESTPG    START    X'0800'
          .
          .
          .
          $CSET             LOAD SCIENTIFIC MICROCODE
          $CNTR             ENTER SCIENTIFIC MICROCODE
          .
          .
          .
          $CALL    SQRTB,X,Y    Y←SQRT(X)
          .
          .
          .
          $INVK    LABELX       LEAVE SCIENTIFIC MODE
LABELX    .
          .
          .
          $EOJ
X         DS       CL4          DEFINE X AREA
Y         DS       CL4          DEFINE Y AREA
          .
          .
          .
          END      TESTPG       END OF ASSEMBLY


SQRTB     $CSUB    2
          $XLD     $ARG1        PICK INDEX TO FIRST ARGUMENT
          $RLD     0,I          PICK UP ARGUMENT VALUE
          .
          .
          .
          $XLD     $ARG2        PICK INDEX TO RESULT VARIABLE
          $RST     0,I          PLACE RESULT IN VARIABLE
          $CRTN                 EXIT SUBROUTINE
WORK      DS       CL8          DEFINE WORK AREAS FOR ROUTINE
          .
          .
          .
          END                   END OF ASSEMBLY
```

Figure 1. Subroutine Linkage Example

Address Register Instructions

The address register is used in conjunction with the invoke ($INVK) macroinstruction to pass parameters and values from scientific subroutines to System/32 subroutines. When a scientific subroutine has been completed and the data required by a System/32 subroutine is ready to be passed to the subroutine, the address of the data (parameter or values) is loaded into the address register. Then, when the $INVK macroinstruction is executed, the contents of the address register are placed in XR2.

ADDRESS REGISTER LOAD ($ALI)

Macroinstruction Format

[Label] $ALI address[,I]

Machine Instruction Format

Byte 1 (Op Code)	**Bytes 2 and 3**
46	Operand address
47	Base address for indexed instruction

Operation

This instruction places the 2-byte effective address in the address register.

Example (Nonindexed)

Instruction

46 00 13

Operand

00000000 00001010
0013 0014

Address Register Before Operation

11001111 10110011
Byte 0 Byte 1

Address Register After Operation

00000000 00010011
Byte 0 Byte 1

Example (Indexed)

Instruction

47 00 13

Operand Before Indexing

00000000 00001010
0013 0014

Index Register

00000000 00000101
Byte 0 Byte 1

Operand After Indexing

00000000 01011100
0018 0019

Address Register Before Operation

11001111 10110011
Byte 0 Byte 1

Address Register After Operation

00000000 00011000
Byte 0 Byte 1

Binary Register Instructions

The binary register instructions perform binary arithmetic on operands serving as fixed-point data, as well as addresses and index quantities. The operands are signed and 32 bits long. Negative quantities are stored in twos-complement form. One operand is always in the binary register; the other operand is in main storage.

Binary register instructions allow loading, adding, subtracting, multiplying, dividing, and storing.

Data Format

Binary numbers appear in a fixed-length format consisting of a sign bit followed by the integer field. When stored in the binary register, a fixed-point quantity has a 31-bit integer field and occupies all 32 bits of the register.

Fixed-Point Number—2 bytes

S	Integer
0	1 … 15

Fixed-Point Number—4 bytes

S	Integer
0	1 … 31

Binary data in main storage appears in a 32-bit format or a 16-bit format, with a binary integer field of 31 or 15 bits, respectively.

A 16-bit operand in main storage is extended to 32 bits by propagating the sign bit as the operand is fetched from storage. Subsequently, the operand is used as a 32-bit operand.

Note: In all discussions of binary numbers in this manual, the expression *4-byte* denotes a 31-bit integer with a sign bit and the expression *2-byte* denotes a 15-bit integer with a sign bit.

Number Representation

All binary operands are treated as signed integers. Positive numbers are represented in true binary notation with the sign bit set to 0. Negative numbers are represented in twos-complement notation with a 1 in the sign bit. The twos complement of a number is obtained by inverting each bit of the number and adding 1 to the result.

This type of number representation is considered the low-order portion of an indefinitely long representation of the number. When the number is positive, all bits to the left of the most significant bit, including the sign bit, are zeros. When the number is negative, all these bits, including the sign bit, are ones. Therefore, when an operand must be extended with the high-order bits, the expansion is achieved by prefixing a field in which each bit is set equal to the high-order bit in the operand.

Twos-complement notation does not include a negative 0. It has a number range in which the set of negative numbers is one larger than the set of positive numbers. The maximum positive number consists of an all-1 integer field with a sign bit of 0, whereas the maximum negative number (the negative number with the greatest absolute value) consists of an all-0 integer field with a sign bit of 1.

The sign bit is the leftmost bit in a number. In an arithmetic operation, a carryout of the integer field changes the sign.

Instruction Format

Binary instructions appear in the following format:

Op code	Operand
0 7	8 24

In this format, bits 0-6 specify the function to be performed by the instruction. Bit 7 indicates if indexing is to be used in addressing the operand. A 0 in bit 7 indicates that bits 8-24 contain the operand location in main storage. If bit 7 is 1 the contents of the index register are added to the operand to form an address designating the storage location of the operand.

The results of binary instructions replace the contents of the binary register; an exception is the *store* instruction, where the register contents replace the data at the main storage location.

The contents of all registers and storage locations participating in the addressing or execution part of an operation remain unchanged, except for the storing of the final result.

BINARY REGISTER ADD ($HADD)–2 BYTES

Macroinstruction

[Label] $HADD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
26	Operand address
27	Base address for indexed instruction

Operation

This instruction adds the 2 bytes of data starting at the effective address to the contents of the binary register. The 2-byte operand is expanded to 4 bytes before addition by propagating the sign-bit value through the 16 high-order positions. Addition is performed by adding all 32 bits. If the carryout of the sign-bit position and the carryout of the high-order numeric bit position are the same, the sum is satisfactory; if they are not the same an overflow occurs. The sign bit is not changed after an overflow. A positive overflow yields a negative final sum, and a negative overflow results in a positive final sum. An overflow is not flagged, nor does a program interrupt occur.

Example (Nonindexed)

Instruction

26 14 C3

Operand

00001101	10111100
14C3	14C4

Binary Register Before Operation

00000000	00000000	00011000	01100110
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00100110	00100010
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

27 14 C3

Operand Before Indexing

00001101	10111100
14C3	14C4

Index Register

00000000	00111010
Byte 0	Byte 1

Operand After Indexing

00001100	10100011
14FD	14FE

Binary Register Before Operation

00000000	00000000	00011100	01010101
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00101000	11111000
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER ADD ($BADD)–4 BYTES

Macroinstruction Format

[Label] $BADD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
1A	Operand address
1B	Base address for indexed instruction

Operation

This instruction adds the 4 bytes of data starting at the effective address to the contents of the binary register. Addition is performed by adding all 32 bits of both operands. If the carryout of the sign-bit position and the carryout of the high-order numeric bit position are the same, the sum is satisfactory; if they are not the same, an overflow occurs. The sign bit is not changed after an overflow. A positive overflow yields a negative final sum, and a negative overflow results in a positive final sum. An overflow is not flagged, nor does a program interrupt occur.

Example (Nonindexed)

Instruction

1A 0C 14

Operand Before And After Operation

00110001	00101110	00110001	00101110
0C14	0C15	0C16	0C17

Binary Register Before Operation

00111000	10100101	00111000	10100101
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

01101001	11010011	01101001	11010011
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

1B 0C 14

Index Register

00001111	01010000
Byte 0	Byte 1

Operand Before Indexing

01110000	11001100	01110000	00101110
0C14	0C15	0C16	0C17

Operand After Indexing

00000011	10100101	00000011	10100101
1B64	1B65	1B66	1B67

Binary Register Before Operation

00111010	01010101	00111010	01010101
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00111101	11111010	00111101	11111010
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER COMPARE ($HCMP)–2 BYTES

Macroinstruction Format

[Label] $HCMP address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
56	Operand address
57	Base address for indexed instruction

Operation

This instruction compares the contents of the binary register with the 2 bytes of data starting at the effective address. The condition code register is set (low, equal, or high). The 2-byte operand is extended to 4 bytes before the comparison by propagating the sign-bit value through the 16 high-order bit positions. Comparison is algebraic, and both operands are treated as 32-bit signed integers.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Code Register Settings

Bit	Name	Condition Indicated
5	Low	Binary register value is less than the operand value
6	Equal	Values are equal
7	High	Binary register value is greater than the operand value

BINARY REGISTER COMPARE ($BCMP)–4 BYTES

Macroinstruction Format

[Label] $BCMP address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
58	Operand address
59	Base address for indexed instruction

Operation

This instruction compares the contents of the binary register with the 4 bytes of data starting at the effective address. The condition code register is set (low, equal, or high). Comparison is algebraic, and both operands are treated as 32-bit signed integers.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Code Register Settings

Bit	Name	Condition Indicated
5	Low	Binary register value is less than the operand value
6	Equal	Values are equal
7	High	Binary register value is greater than the operand value

BINARY REGISTER DIVIDE ($HDIV)–2 BYTES

Macroinstruction Format

[Label] $HDIV address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
24	Operand address
25	Base address for indexed instruction

Operation

This instruction divides the contents of the binary register by the 2 bytes of data starting at the effective address. The 2-byte operand is extended to 4 bytes before the division by propagating the sign-bit value through the 16 high-order bit positions. Both operands are treated as 32-bit signed integers. The quotient is a 32-bit signed integer and replaces the dividend in the binary register. If both operands have the same sign, the quotient is positive. If they have opposite signs, the quotient is negative. A zero quotient is always positive.

Example (Nonindexed)

Instruction

24 04 9E

Operand

00000000	00000101
049E	049F

Binary Register Before Operation

00000000	00000000	00000000	00110010
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00000000	00001010
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

25 04 9E

Index Register

00000000	00001011
Byte 0	Byte 1

Operand Before Indexing

00000000	00000101
049E	049F

Operand After Indexing

00000000	00001010
04A9	04AA

Binary Register Before Operation

00000000	00000000	00000000	00110010
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00000000	00000101
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER DIVIDE ($BDIV)–4 BYTES

Macroinstruction Format

[Label] $BDIV address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
18	Operand address
19	Base address for indexed instruction

Operation

This instruction divides the contents of the binary register by the 4 bytes of data starting at the effective address. Both operands are treated as 32-bit signed integers. The quotient is a 32-bit signed integer and replaces the dividend in the binary register. If both operands have the same sign, the quotient is positive. If they have opposite signs, the quotient is negative. A zero quotient is always positive.

Example (Nonindexed)

Instruction

18 01 B3

Operand

00000000	00000000	00000000	00001100
01B3	01B4	01B5	01B6

Binary Register Before Operation

00000000	00000000	00100010	00001000
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00000010	11010110
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

19 01 B3

Index Register

00000000	01100111
Byte 0	Byte 1

Operand Before Indexing

00000000	00000000	00000000	00001100
01B3	01B4	01B5	01B6

Operand After Indexing

00000000	00000000	00000000	00000110
021A	021B	021C	021D

Binary Register Before Operation

00000000	00000000	00100010	00001000
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00000101	10101100
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER LOAD ($HLD)–2 BYTES

Macroinstruction Format

[Label] $HLD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
2C	Operand address
2D	Base address for indexed instruction

Operation

This instruction places the 2 bytes of data starting at the effective address in the binary register. The 2-byte operand is extended to 4 bytes during the operation by propagating the sign-bit value through the 16 high-order bit positions.

Example (Nonindexed)

Instruction

2C 02 C1

Operand

01100011	10100011
02C1	02C2

Binary Register Before Operation

00000000	01000001	00000000	00111100
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	01100011	10100011
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

2D 02 C1

Index Register

00000000	00110001
Byte 0	Byte 1

Operand Before Indexing

01100011	10100011
02C1	02C2

Operand After Indexing

10100011	00111010
02F2	02F3

Binary Register Before Operation

00000000	01000001	00000000	00111100
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

11111111	11111111	10100011	00111010
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER LOAD ($BLD)–4 BYTES

Macroinstruction Format

[Label] $BLD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
20	Operand address
21	Base address for indexed instruction

Operation

This instruction places the 4 bytes of data starting at the effective address in the binary register.

Example (Nonindexed)

Instruction

20 01 D4

Operand

00000000	10011101	00110101	11001010
01D4	01D5	01D6	01D7

Binary Register Before Operation

10100011	11000010	00111010	11000001
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	10011101	00110101	11001010
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

21 01 D4

Index Register

00000010	00111000
Byte 0	Byte 1

Operand Before Indexing

00000000	10011101	00110101	11001010
01D4	01D5	01D6	01D7

Operand After Indexing

01100011	10100101	11000110	11110010
040C	040D	040E	040F

Binary Register Before Operation

10100011	11000010	00111010	11000001
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

01100011	10100101	11000110	11110010
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER MULTIPLY ($HMLT)–2 BYTES

Macroinstruction Format

[Label] $HMLT address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
2A	Operand address
2B	Base address for indexed instruction

Operation

This instruction multiplies the contents of the binary register by the 2 bytes of data starting at the effective address. The 2-byte multiplier is extended to 4 bytes before multiplication by propagating the sign-bit value through the 16 high-order bit positions. Both the multiplier and the multiplicand are 32-bit signed integers. The product is always a 32-bit signed integer and replaces the multiplicand in the binary register. The sign of the product is determined by the rules of algebra from the multiplier and multiplicand signs, except that 0 is always positive. An overflow is not flagged, nor does a program interrupt occur.

Programming Note: The significant digits of the product usually occupy 32 bits or less; however, if the product exceeds 32 bits, the high-order bits are shifted out without inspection and are lost.

Example (Nonindexed)

Instruction

2A 10 93

Operand

00000000	00000111
1093	1094

Binary Register Before Operation

00000000	00000000	00000000	01100011
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00000010	10110101
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

2B 10 93

Index Register

00000000	00001100
Byte 0	Byte 1

Operand Before Indexing

00000000	00000111
1093	1094

Operand After Indexing

00000000	00001001
109F	10A0

Binary Register Before Operation

00000000	00000000	00000000	10110101
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	00000110	01011101
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER MULTIPLY ($BMLT)–4 BYTES

Macroinstruction Format

[Label] $BMLT address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
1E	Operand address
1F	Base address for indexed instruction

Operation

This instruction multiplies the contents of the binary register by the 4 bytes of data starting at the effective address. Both the multiplier and the multiplicand are 32-bit signed integers. The product is always a 32-bit signed integer and replaces the multiplicand in the binary register. The sign of the product is determined by the rules of algebra from the multiplier and multiplicand signs, except that 0 is always positive. An overflow is not flagged, nor does a program interrupt occur.

Programming Note: The significant digits of the product usually occupy 32 bits or less; however, if the product exceeds 32 bits, the high-order bits are shifted out without inspection and are lost.

Example (Nonindexed)

Instruction

1E 01 C4

Operand

00000000	00000000	10100001	00101001
01C4	01C5	01C6	01C7

Binary Register Before Operation

00000000	00000000	00000000	11000110
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	01111100	10100101	10110110
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

1F01 C4

Index Register

00000000	10100011
Byte 0	Byte 1

Operand Before Indexing

00000000	00000000	10100001	00101001
01C4	01C5	01C6	01C7

Operand After Indexing

00000000	00000000	00000001	00110110
0267	0268	0269	026A

Binary Register Before Operation

00000000	00000000	00000000	11000110
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00000000	11101111	11000100
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER STORE ($HST)–2 BYTES

Macroinstruction Format

[Label] $HST address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
22	Operand address
23	Base address for indexed instruction

Operation

This instruction places the contents of the 2 low-order bytes of the binary register in the 2-byte area starting at the effective address.

Example (Nonindexed)

Instruction

22 04 36

Binary Register

00111000	01100110	10100011	11001001
Byte 0	Byte 1	Byte 2	Byte 3

Operand Before Operation

00011000	11110111
0436	0437

Operand After Operation

10100011	11001001
0436	0437

Example (Indexed)

Instruction

23 04 36

Index Register

00000000	10010011
Byte 0	Byte 1

Binary Register

00000000	11000011	10100101	00111100
Byte 0	Byte 1	Byte 2	Byte 3

Operand Before Indexing

00011000	11110111
0436	0437

Operand Before Operation (after indexing)

10011001	01100110
04C9	04CA

Operand After Operation

10100101	00111100
04C9	04CA

BINARY REGISTER STORE ($BST)–4 BYTES

Macroinstruction Format

[Label] $BST address[,I]

Machine Instruction Format

Byte 1 (Op Code)	**Bytes 2 and 3**
16	Operand address
17	Base address for indexed instruction

Operation

This instruction places the contents of the binary register in the 4-byte area starting at the effective address.

Example (Nonindexed)

Instruction

16 0C 19

Binary Register

00000000	01001101	00111010	11000101
Byte 0	Byte 1	Byte 2	Byte 3

Operand Before Operation

00111100	01011100	01101001	00111100
0C19	0C1A	0C1B	0C1C

Operand After Operation

00000000	01001101	00111010	11000101
0C19	0C1A	0C1B	0C1C

Example (Indexed)

Instruction

17 0C 19

Index Register

00000000	00111010
Byte 0	Byte 1

Binary Register

00000000	01001101	00111010	11000101
Byte 0	Byte 1	Byte 2	Byte 3

Operand Before Indexing

00111100	01011100	01101001	00111100
0C19	0C1A	0C1B	0C1C

Operand Before Operation (after indexing)

01100011	11000111	10101010	01010111
0C53	0C54	0C55	0C56

Operand After Operation

00000000	01001101	00111010	11000101
0C53	0C54	0C55	0C56

BINARY REGISTER SUBTRACT ($HSUB)–2 BYTES

Macroinstruction Format

[Label] $HSUB address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
28	Operand address
29	Base address for indexed instruction

Operation

This instruction subtracts the 2 bytes of data starting at the effective address from the contents of the binary register. The 2-byte operand is extended to 4 bytes before the subtraction by propagating the sign-bit value through the 16 high-order bit positions. All 32 bits of both operands are used, as in *Binary Register Add ($HADD)*.

Example (Nonindexed)

Instruction

28 03 19

Operand

00001011	01101100
0319	031A

Binary Register Before Operation

00000000	00001100	00111100	11000111
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00001100	00110001	01011011
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

29 03 19

Index Register

00000000	11010100
Byte 0	Byte 1

Operand Before Indexing

00001011	01101100
0319	031A

Operand After Indexing

00000100	01110111
03ED	03EE

Binary Register Before Operation

00000000	00001100	00111100	11000111
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00001100	00111000	01010000
Byte 0	Byte 1	Byte 2	Byte 3

BINARY REGISTER SUBTRACT ($BSUB)–4 BYTES

Macroinstruction Format

[Label] $BSUB address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
1C	Operand address
1D	Base address for indexed instruction

Operation

This instruction subtracts the 4 bytes of data starting at the effective address from the contents of the binary register. All 32 bits of both operands are used, as in *Binary Register Add ($BADD)*.

Example (Nonindexed)

Instruction

1C 00 7B

Operand

00000000	11001100	11110100	01000011
007B	007C	007D	007E

Binary Register Before Operation

00000000	11111100	11001100	10001111
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	00101111	11011000	01001100
Byte 0	Byte 1	Byte 2	Byte 3

Example (Indexed)

Instruction

1D 00 7B

Index Register

00000000	01100011
Byte 0	Byte 1

Operand Before Indexing

00000000	11001100	11110100	01000011
007B	007C	007D	007E

Operand After Indexing

00000000	00001100	01001000	10000011
00DE	00DF	00E0	00E1

Binary Register Before Operation

00000000	11111100	11001100	10001111
Byte 0	Byte 1	Byte 2	Byte 3

Binary Register After Operation

00000000	11110000	10000100	00001100
Byte 0	Byte 1	Byte 2	Byte 3

Floating-Point Register Instructions

The floating-point instructions perform calculations on operands with a wide range of magnitude and yield scaled results to preserve precision.

A floating-point number consists of a signed characteristic and a signed mantissa. The quantity expressed by this number is the product of the mantissa and the number 16 raised to the power of the characteristic. The characteristic is expressed in excess-64 notation; the mantissa is expressed as a hexadecimal number having a radix point (see *Number Representation*, later in this chapter) to the left of the high-order digit.

The floating-point instructions provide loading, adding, subtracting, multiplying, dividing, and storing. Short operands provide faster processing and require less storage than long operands. Long operands provide greater precision in computation.

Maximum precision is preserved in addition, subtraction, multiplication, and division by producing normalized results (see *Normalization*, in this chapter). Normalized operands are used in any floating-point operation.

Data Format

Floating-point data appears in a fixed-length format, which may be either a single-precision format or a double-precision format. Both formats can be used in main storage and in the floating-point register.

Single-Precision Floating-Point Number

S	Characteristic	Mantissa
0	1 – 7	8 – 31

Double-Precision Floating-Point Number

S	Characteristic	Mantissa
0	1 – 7	8 – 63

The first bit in either format is the sign bit (S). The subsequent 7 bit positions are occupied by the characteristic. The mantissa can have either 6 or 14 hexadecimal digits.

The entire set of floating-point instructions is available for both single- and double-precision operands. When single-precision is specified, all operands and results are 32-bit floating-point values. The rightmost 32 bits of the floating-point register are not used in the operations and remain unchanged. When double-precision is specified, all operands and results are 64-bit floating-point values.

Final results have six mantissa digits in single-precision and 14 mantissa digits in double-precision.

Number Representation

The mantissa of a floating-point number is expressed in hexadecimal digits. The radix point of the mantissa is assumed to be immediately to the left of the high-order mantissa digit. To provide the proper magnitude for the floating-point number, the mantissa is considered to be multiplied by the power of 16. The characteristic portion, bits 1-7 of both floating-point formats, indicates this power. The bits within the characteristic field can represent numbers from 0 through 127. To accommodate large and small magnitudes, the characteristic is formed by adding 64 to the actual number. The range of the characteristic is thus -64 through +63. This technique produces a characteristic in excess 64 notation.

Both positive and negative quantities have a true mantissa, the difference in sign being indicated by the sign bit. The number is positive or negative accordingly as the sign bit is 0 or 1.

The range covered by the magnitude (M) of a normalized floating-point number is:

$16^{-65} \leq M \leq (1 - 16^{-6})\ 16^{63}$ in single precision
$16^{-65} \leq M \leq (1 - 16^{-14})\ 16^{63}$ in double precision
or approximately
$5.4\ 10^{-79} \leq M \leq 7.2\ 10^{75}$ in either precision

Normalization

A quantity can be represented with the greatest precision by a floating-point number of given mantissa length when that number is *normalized*. A normalized floating-point number has a nonzero, high-order, hexadecimal mantissa digit.

If one or more high-order mantissa digits are 0, the number is said to be unnormalized. The process of normalization consists of shifting the mantissa to the left until the high-order hexadecimal digit is nonzero and reducing the characteristic by the number of hexadecimal digits shifted. A 0 fraction is considered to be normalized.

Normalization usually takes place when the intermediate arithmetic result is changed to the final result. This function is called *postnormalization.*

Programming Note: Since normalization applies to hexadecimal digits, the 3 high-order bits of a normalized mantissa may be 0.

Instruction Format

Floating-point instructions appear in the following format:

Op code	Operand
0 7	8 24

In this format, bits 0-6 specify the function to be performed by the instruction. Bit 7 indicates if indexing is to be used in addressing the operand. A 0 in bit 7 indicates that bits 8-24 contain the operand location in main storage. If bit 7 is 1 the contents of the index register are added to the operand to form an address designating the storage location of the operand.

FLOATING-POINT REGISTER ADD ($RADD)–SINGLE PRECISION

Macroinstruction Format

[Label] $RADD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
32	Operand address
33	Base address for indexed instruction

Operation

This instruction adds the 4 bytes of data starting at the effective address to the contents of the floating-point register. The 4 low-order bytes of the floating-point register are ignored and remain unchanged.

Addition of two floating-point numbers consists of a characteristic comparison and a mantissa addition. The characteristics of the two operands are compared, and the mantissa with the smaller characteristic is shifted right; its characteristic is increased by 1 for each hexadecimal digit of shift until the two characteristics agree. The mantissas are then added algebraically to form an intermediate sum. The intermediate sum consists of seven hexadecimal digits and a possible carry.

The low-order digit is a guard digit obtained from the mantissa that is shifted right. Only one guard digit position is used in the addition. The guard digit is 0 if no shift occurs. After the addition, the intermediate sum is shifted left as necessary to form a normalized fraction; vacated low-order positions are filled with zeros; and, the characteristic is reduced by the amount of shift. The sign of the sum is derived by the rules of algebra. The sign of a sum with a 0 mantissa is always positive.

Example (Nonindexed)

Instruction

32	14	32

Operand

40	10	24	00
1432	1433	1434	1435

Floating-Point Register Before Operation

40	21	34	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	31	58	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

33	14	32

Index Register

01	34
Byte 0	Byte 1

Operand Before Indexing

40	10	24	00
1432	1433	1434	1435

Operand After Indexing

40	11	93	01
1566	1567	1568	1569

Floating-Point Register Before Operation

40	21	00	00	72	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	32	93	01	72	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER ADD ($DADD)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DADD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
3E	Operand address
3F	Base address for indexed instruction

Operation

This instruction adds the 8 bytes of data starting at the effective address to the contents of the floating-point register.

Addition of two floating-point numbers consists of a characteristic comparison and a mantissa addition. The characteristics of the two operands are compared, and the mantissa with the smaller characteristic is shifted right; its characteristic is increased by 1 for each hexadecimal digit of shift until the two characteristics agree. The mantissas are then added algebraically to form an intermediate sum. The intermediate sum consists of 15 hexadecimal digits and a possible carry.

The low-order digit is a guard digit obtained from the mantissa that is shifted right. Only one guard digit position is used in the mantissa addition. The guard digit is 0 if no shift occurs.

After the addition, the intermediate sum is shifted left as necessary to form a normalized mantissa; vacated low-order positions are filled with zeros; and the characteristic is reduced by the amount of shift. The sign of the sum is derived by the rules of algebra. The sign of a sum with a 0 mantissa is always positive.

Example (Nonindexed)

Instruction

3E 03 47

Operand

36	12	04	00	00	00	00	00
0347	0348	0349	034A	034B	034C	034D	034E

Floating-Point Register Before Operation

35	10	60	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

36	13	0A	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

3F 03 47

Index Register

01	01
Byte 0	Byte 1

Operand Before Indexing

40	00	00	00	00	00	12	04
0347	0348	0349	034A	034B	034C	034D	034E

Operand After Indexing

37	29	71	00	00	00	00	00
0457	0458	0459	045A	045B	045C	045D	045E

Floating-Point Register Before Operation

37	13	0A	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

37	3C	7B	00	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER COMPARE ($RCMP)–SINGLE PRECISION

Macroinstruction Format

[Label] $RCMP address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
5A	Operand address
5B	Base address for indexed instruction

Operation

This instruction compares the contents of the floating-point register with the 4 bytes of data starting at the effective address. The condition code register is set (low, equal, or high). The 4 low-order bytes of the floating-point register are ignored. Comparison is algebraic and takes into account the sign, characteristic, and mantissa of each number.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Register Settings

Bit	Name	Condition Indicated
5	Low	Binary register value is less than the operand value
6	Equal	Values are equal
7	High	Binary register value is greater than the operand value

FLOATING-POINT REGISTER COMPARE ($DCMP)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DCMP address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
5C	Operand address
5D	Base address for indexed instruction

Operation

This instruction compares the contents of the floating-point register with the 8 bytes of data starting at the effective address. The condition code register is set (low, equal, or high). Comparison is algebraic and takes into account the sign, characteristic, and mantissa of each number.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Register Settings

Bit	Name	Condition Indicated
5	Low	Binary register value is less than the operand value
6	Equal	Values are equal
7	High	Binary register value is greater than the operand value

FLOATING-POINT REGISTER DIVIDE ($RDIV)–SINGLE PRECISION

Macroinstruction Format

[Label] $RDIV address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
30	Operand address
31	Base address for indexed instruction

Operation

This instruction divides the contents of the floating-point register by the 4 bytes of data starting at the effective address. If the data is 0, the divide check indicator is set in the scientific communication area and the floating-point register remains unchanged.

A floating-point division consists of a characteristic subtraction and a fraction division. The difference between the dividend characteristic and the divisor characteristic plus 64 is used as an intermediate characteristic. The sign of the quotient is determined by the rules of algebra.

All dividend fraction digits participate in forming the quotient, even if the normalized dividend fraction is larger than the normalized divisor fraction. The quotient fraction is normalized, if necessary.

Example (Nonindexed)

Instruction

30 16 31

Operand

37	E0	00	00
1631	1632	1633	1634

Floating-Point Register Before Operation

35	A8	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

3E	C0	00	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

31 16 31

Index Register

00	34
Byte 0	Byte 1

Operand Before Indexing

37	E0	00	00
1631	1632	1633	1634

Operand After Indexing

35	E0	00	00
1665	1666	1667	1668

Floating-Point Register Before Operation

35	A8	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	C0	00	00	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER DIVIDE ($DDIV)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DDIV address[,I]

Machine Instruction Fromat

Byte 1 (Op Code)	Bytes 2 and 3
3C	Operand address
3D	Base address for indexed instruction

Operation

This instruction divides the contents of the floating-point register by the 8 bytes of data starting at the effective address. If the data is 0, the divide check indicator is set in the scientific communication area.

A floating-point division consists of a characteristic subtraction and a fraction division. The difference between the dividend characteristic and the divisor characteristic plus 64 is used as an intermediate characteristic. The sign of the quotient is determined by the rules of algebra.

All dividend fraction digits participate in forming the quotient, even if the normalized dividend fraction is larger than the normalized divisor fraction. The quotient fraction is normalized, if necessary

Example (Nonindexed)

Instruction

3C 17 43

Operand

33	E0	00	00	00	00	00	00
1743	1744	1745	1746	1747	1748	1749	174A

Floating-Point Register Before Operation

34	B6	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

41	D0	00	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

3D 17 43

Index Register

01	01
Byte 0	Byte 1

Operand Before Indexing

33	E0	00	00	00	00	00	00
1743	1744	1745	1746	1747	1748	1749	174A

Operand After Indexing

33	B0	00	00	00	00	00	00
1844	1845	1846	1847	1848	1849	184A	184B

Floating-Point Register Before Operation

34	6E	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

41	A0	00	00	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER LOAD ($RLD)–SINGLE PRECISION

Macroinstruction Format

[Label] $RLD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
38	Operand address
39	Base address for indexed instruction

Operation

This instruction places the 4 bytes of data starting at the effective address in the floating-point register and sets the floating-point register status to single-precision.

Example (Nonindexed)

Instruction

38 04 62

Operand

40	36	93	02
0462	0463	0464	0465

Floating-Point Register Before Operation

39	08	67	30	01	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	36	93	02	01	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

39 04 62

Index Register

00	12
Byte 0	Byte 1

Operand Before Indexing

40	36	93	02
0462	0463	0464	0465

Operand After Indexing

41	27	08	00
0474	0475	0476	0477

Floating-Point Register Before Operation

39	08	67	30	01	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

41	27	08	00	01	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER LOAD ($DLD)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DLD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
44	Operand address
45	Base address for indexed instruction

Operation

This instruction places the 8 bytes of data starting at the effective address in the floating-point register and sets the floating-point register status to double-precision.

Example (Nonindexed)

Instruction

44 81 94

Operand

36	14	30	00	00	00	00	00
8194	8195	8196	8197	8198	8199	819A	819B

Floating-Point Register Before Operation

40	18	96	40	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

36	14	30	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

45 81 94

Index Register

00	20
Byte 0	Byte 1

Operand Before Indexing

36	14	30	00	00	00	00	00
8194	8195	8196	8197	8198	8199	819A	819B

Operand After Indexing

34	26	19	00	00	00	00	00
81B4	81B5	81B6	81B7	81B8	81B9	81BA	81BB

Floating-Point Register Before Operation

40	18	96	40	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

34	26	19	00	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER MULTIPLY ($RMLT)–SINGLE PRECISION

Macroinstruction Format

[Label] $RMLT address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
36	Operand address
37	Base address for indexed instruction

Operation

This instruction multiplies the contents of the floating-point register by the 4 bytes of data starting at the effective address.

The multiplication of two floating-point numbers consists of a characteristic addition and a fraction multiplication. The sum of the characteristics minus 64 is used as the characteristic of the product. The sign of the product is determined by the rules of algebra. The product fraction is normalized, if necessary. The product characteristic is reduced by the number of left shifts. The product fraction is truncated to 6 digits after normalization. When the product fraction is zero, the product sign and characteristic are made zeros, yielding a true zero result.

Example (Nonindexed)

Instruction

36 06 42

Operand

41	F0	00	00
0642	0643	0644	0645

Floating-Point Register Before Operation

34	A0	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

35	96	00	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

36 06 42

Index Register

00	43
Byte 0	Byte 1

Operand Before Indexing

41	F0	00	00
0642	0643	0644	0645

Operand After Indexing

39	B0	00	00
0685	0686	0687	0688

Floating-Point Register Before Operation

34	A0	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

2D	6E	00	00	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER MULTIPLY ($DMLT)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DMLT address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
42	Operand address
43	Base address for indexed instruction

Operation

This instruction multiplies the contents of the floating-point register by the 8 bytes of data starting at the effective address.

The multiplication of two floating-point numbers consists of a characteristic addition and a fraction multiplication. The sum of the characteristics minus 64 is used as the characteristic of the product. The sign of the product is determined by the rules of algebra. The product fraction is normalized, if necessary. The product characteristic is reduced by the number of left shifts. The product fraction is truncated to 14 digits after normalization. When the product fraction is zero, the product sign and characteristic are made zeros, yielding a true zero result.

Example (Nonindexed)

Instruction

42 08 13

Operand

40	90	00	00	00	00	00	00
0813	0814	0815	0816	0817	0818	0819	081A

Floating-Point Register Before Operation

3F	D0	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	75	00	00	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

43 08 13

Index Register

01	00
Byte 0	Byte 1

Operand Before Indexing

41	09	00	00	00	00	00	00
0813	0814	0815	0816	0817	0818	0819	081A

Operand After Indexing

40	70	00	00	00	00	00	00
0913	0914	0915	0916	0917	0918	0919	091A

Floating-Point Register Before Operation

40	E0	00	00	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	62	00	00	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER STORE ($RST)–SINGLE PRECISION

Macroinstruction Format

[Label] $RST address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
2E	Operand address
2F	Base address for indexed instruction

Operation

This instruction places the single-precision portion (high-order bytes) of the floating-point register in the 4-byte area starting at the effective address.

Example (Nonindexed)

Instruction

2E	01	23

Floating-Point Register

39	08	42	60	19	08	00	00
Byte 0							Byte 7

Operand Before Operation

41	92	36	08
0123	0124	0125	0126

Operand After Operation

39	08	42	60
0123	0124	0125	0126

Example (Indexed)

Instruction

2F	01	23

Index Register

00	08
Byte 0	Byte 1

Floating-Point Register

40	18	09	63	00	00	00	00
Byte 0							Byte 7

Operand Before Indexing

41	92	36	08
0123	0124	0125	0126

Operand Before Operation (after indexing)

39	10	83	62
012B	012C	012D	012E

Operand After Operation

40	18	09	63
012B	012C	012D	012E

FLOATING-POINT REGISTER STORE ($DST)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DST address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
3A	Operand address
3B	Base address for indexed instruction

Operation

This instruction places the contents of the floating-point register in the 8 byte area starting at the effective address.

Example (Nonindexed)

Instruction

3A 48 03

Floating-Point Register

49	80	14	30	00	00	00	00
Byte 0							Byte 7

Operand Before Operation

36	00	91	87	40	00	00	00
4803	4804	4805	4806	4807	4808	4809	480A

Operand After Operation

49	80	14	30	00	00	00	00
4803	4804	4805	4806	4807	4808	4809	480A

Example (Indexed)

Instruction

3B 48 03

Index Register

03	8A
Byte 0	Byte 1

Floating-Point Register

38	10	83	47	62	10	00	00
Byte 0							Byte 7

Operand Before Indexing

36	00	91	87	40	00	00	00
4803	4804	4805	4806	4807	4808	4809	480A

Operand Before Operation (after indexing)

31	68	79	53	00	00	00	00
4B8D	4B8E	4B8F	4B90	4B91	4B92	4B93	4B94

Operand After Operation

38	10	83	47	62	10	00	00
4B8D	4B8E	4B8F	4B90	4B91	4B92	4B93	4B94

FLOATING-POINT REGISTER SUBTRACT ($RSUB)–SINGLE PRECISION

Macroinstruction Format

[Label] $RSUB address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
34	Operand address
35	Base address for indexed instruction

Operation

This instruction subtracts the 4 bytes of data starting at the effective address from the contents of the floating-point register. The low-order half of the floating-point register is ignored and remains unchanged. This instruction is similar to *Floating-Point Register Add ($RADD)*, except that the sign of the operand is inverted before addition. The sign of the difference is determined by the rules of algebra. The sign of the difference with a 0 result fraction is always positive.

Example (Nonindexed)

Instruction

34 03 45

Operand

39	18	43	00
0345	0346	0347	0348

Floating-Point Register Before Operation

40	91	96	50	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	90	12	20	00	00	00	00
Byte 0							Byte 7

Example (Indexed)

Instruction

35 03 45

Index Register

00	10
Byte 0	Byte 1

Operand Before Indexing

39	18	43	00
0345	0346	0347	0348

Operand After Indexing

40	80	14	30
0355	0356	0357	0358

Floating-Point Register Before Operation

40	91	96	50	00	00	00	00
Byte 0							Byte 7

Floating-Point Register After Operation

40	11	82	20	00	00	00	00
Byte 0							Byte 7

FLOATING-POINT REGISTER SUBTRACT ($DSUB)–DOUBLE PRECISION

Macroinstruction Format

[Label] $DSUB address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
40	Operand address
41	Base address for indexed instruction

Operation

This instruction subtracts the 8 bytes of data starting at the effective address from the contents of the floating-point register. This instruction is similar to *Floating-Point Register Add ($DADD)*, except that the sign of the operand is inverted before addition. The sign of the difference is determined by the rules of algebra. The sign of the difference with a 0 result fraction is always positive.

Example (Nonindexed)

Instruction

40	10	F2

Operand

36	21	02	69	52	01	11	00
10F2	10F3	10F4	10F5	10F6	10F7	10F8	10F9

Floating-Point Register Before Operation

36	38	14	79	63	55	21	00
Byte 0							Byte 7

Floating-Point Register After Operation

36	17	12	10	11	54	10	00
Byte 0							Byte 7

Example (Indexed)

Instruction

41	10	F2

Index Register

04	00
Byte 0	Byte 1

Operand Before Indexing

36	21	02	69	52	01	11	00
10F2	10F3	10F4	10F5	10F6	10F7	10F8	10F9

Operand After Indexing

36	16	03	58	61	43	11	00
14F2	14F3	14F4	14F5	14F6	14F7	14F8	14F9

Floating-Point Register Before Operation

36	38	14	79	63	55	21	00
Byte 0							Byte 7

Floating-Point Register After Operation

36	22	11	21	02	12	10	00
Byte 0							Byte 7

Index Multiplier Register Instructions

INDEX MULTIPLIER REGISTER LOAD IMMEDIATE ($MLI)

Macroinstruction Format

[Label] $MLI DATA[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
12	Data
13	Base data

Operation

This instruction places 2 bytes of data from the instruction in the index multiplier register. If indexing is used, the sum of the instruction data added to the contents of the index register is placed in the index multiplier register.

Example (Nonindexed)

Instruction

12 04 CA

Index Multiplier Register Before Operation

00000011	10101011
Byte 0	Byte 1

Index Multiplier Register After Operation

00000100	11001010
Byte 0	Byte 1

Example (Indexed)

Instruction

13 04 CA

Operand Before Indexing

04CA

Index Register

00000000	00001011
Byte 0	Byte 1

Operand After Indexing

04D5

Index Multiplier Register Before Operation

00000011	10101011
Byte 0	Byte 1

Index Multiplier Register After Operation

00000100	11010101
Byte 0	Byte 1

INDEX MULTIPLIER REGISTER STORE ($MST)

Macroinstruction Format

[Label] $MST address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
14	Operand address
15	Base address for indexed instruction

Operation

This instruction places the contents of the index multiplier register in the 2-byte area starting at the effective address.

Example (Nonindexed)

Instruction

14 03 96

Index Multiplier Register

00000000	00000111
Byte 0	Byte 1

Operand Before Operation

00111000	01011101
0396	0397

Operand After Operation

00000000	00000111
0396	0397

Example (Indexed)

Instruction

15 03 96

Index Register

00000000	00001100
Byte 0	Byte 1

Index Multiplier Register

00000000	01000101
Byte 0	Byte 1

Operand Before Indexing

00111000	01011101
0396	0397

Operand Before Operation (after indexing)

00000100	10011001
03A2	03A3

Operand After Operation

00000000	01000101
03A2	03A3

Index Register Instructions

INDEX REGISTER ADD ($XADD)

Macroinstruction Format

[Label] $XADD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
08	Operand address
09	Base address for indexed instruction

Operation

This instruction adds the 2 bytes of data starting at the effective address to the contents of the index register.

Example (Nonindexed)

Instruction

08 08 14

Operand

00000001 10001001
0814 0815

Index Register Before Operation

00000111 00101100
Byte 0 Byte 1

Index Register After Operation

00001000 10110101
Byte 0 Byte 1

Example (Indexed)

Instruction

09 08 14

Index Register

00000101 10100011
Byte 0 Byte 1

Operand Before Indexing

00000001 10001001
0814 0815

Operand After Indexing

00000000 11011011
0DB7 0DB8

Index Register Before Operation

00000101 10100011
Byte 0 Byte 1

Index Register After Operation

00000110 01111110
Byte 0 Byte 1

INDEX REGISTER LOAD ($XLD)

Macroinstruction Format

[Label] $XLD address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
06	Operand address
07	Base address for indexed instruction

Operation

This instruction places the 2 bytes of data starting at the effective address in the index register.

Example (Nonindexed)

Instruction

06 07 38

Operand

00000000 01111101
0738 0739

Index Register Before Operation

00001100 10100011
Byte 0 Byte 1

Index Register After Operation

00000000 01111101
Byte 0 Byte 1

Example (Indexed)

Instruction

07 07 38

Index Register

00000000 01111001
Byte 0 Byte 1

Operand Before Indexing

00000000 01111101
0738 0739

Operand After Indexing

00000000 11011111
07B1 07B2

Index Register Before Operation

00000000 01111001
Byte 0 Byte 1

Index Register After Operation

00000000 11011111
Byte 0 Byte 1

INDEX REGISTER LOAD IMMEDIATE ($XLI)

Macroinstruction Format

[Label] $XLI data[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
0A	Data
0B	Data for indexed instruction

Operation

This instruction places the 2 bytes of data from the instruction in the index register. If indexing is used, the instruction data is added to the contents of the index register. The result is placed in the index register.

Example (Nonindexed)

Instruction

0A 01 8C

Operand

018C

Index Register Before Operation

00000011 10011110
Byte 0 Byte 1

Index Register After Operation

00000001 10001100
Byte 0 Byte 1

Example (Indexed)

Instruction

0B 01 8C

Operand Before Indexing

018C

Index Register

00000000 01101111
Byte 0 Byte 1

Operand After Indexing

01FB

Index Register Before Operation

00000000 01101111
Byte 0 Byte 1

Index Register After Operation

00000001 11111011
Byte 0 Byte 1

INDEX REGISTER MULTIPLY ($XMLT)

Macroinstruction Format

[Label] $XMLT address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
0E	Operand address
0F	Base address for indexed instruction

Operation

This instruction multiplies the contents of the index multiplier register by the 2 bytes of data starting at the effective address and places the product in the index register.

Example (Nonindexed)

Instruction

0E 06 C4

Operand

00000000	00001101
06C4	06C5

Index Multiplier Register

00000000	00000011
Byte 0	Byte 1

Index Register Before Operation

00000010	10001010
Byte 0	Byte 1

Index Register After Operation

00000000	00100111
Byte 0	Byte 1

Example (Indexed)

Instruction

0F 06 C4

Index Register

00000000	00001100
Byte 0	Byte 1

Operand Before Indexing

00000000	00001101
06C4	06C5

Operand After Indexing

00000000	00011000
06D0	06D1

Index Multiplier Register

00000000	00000011
Byte 0	Byte 1

Index Register After Operation

00000000	01001000
Byte 0	Byte 1

INDEX REGISTER MULTIPLY AND ADD ($XMTA)

Macroinstruction Format

[Label] $XMTA address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
10	Operand address
11	Base address for indexed instruction

Operation

This instruction adds the product of the index multiplier register and the 2 bytes of data starting at the effective address to the contents of the index register.

Example (Nonindexed)

Instruction

10 0D C2

Operand

00000000 00000110
0DC2 0DC3

Index Multiplier Register

00000000 00000101
Byte 0 Byte 1

Index Register Before Operation

00000000 00101010
Byte 0 Byte 1

Index Register After Operation

00000000 01001000
Byte 0 Byte 1

Example (Indexed)

Instruction

11 0D C2

Index Register

00000000 00101010
Byte 0 Byte 1

Operand Before Indexing

00000000 00000110
0DC2 0DC3

Operand After Indexing

00000000 00000010
0DEC 0DED

Index Multiplier Register

00000000 00000101
Byte 0 Byte 1

Index Register Before Operation

00000000 00101010
Byte 0 Byte 1

Index Register After Operation

00000000 00110100
Byte 0 Byte 1

INDEX REGISTER STORE ($XST)

Macroinstruction Format

[Label] $XST address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
0C	Operand address
0D	Base address for indexed instruction

Operation

This instruction places the contents of the index register in the 2 byte area starting at the effective address.

Example (Nonindexed)

Instruction

0C 0A 12

Index Register

00000000	00110011
Byte 0	Byte 1

Operand Before Operation

00001001	10011001
0A12	0A13

Operand After Operation

00000000	00110011
0A12	0A13

Example (Indexed)

Instruction

0D 0A 12

Index Register

00000000	00110011
Byte 0	Byte 1

Operand Before Indexing

00001001	10011001
0A12	0A13

Operand Before Operation (after indexing)

00110110	01100110
0A45	0A46

Operand After Operation

00000000	00110011
0A45	0A46

Logical Instructions

BINARY REGISTER AND ($BAND)

Macroinstruction Format

[Label] $BAND address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
60	Operand address
61	Base address for indexed instruction

Operation

This instruction interrogates the contents of the binary register and the 4 bytes of data starting at the effective address. If both values are nonzero, the binary register is set to X'00000001'. If either value is 0, the binary register is set to X'00000000'.

BINARY REGISTER OR ($BOR)

Macroinstruction Format

[Label] $BOR address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
62	Operand address
63	Base address for indexed instruction

Operation

This instruction interrogates the contents of the binary register and the 4 bytes of data starting at the effective address. If both values are 0, the binary register remains unchanged. If either value is nonzero, the binary register is set to X'00000001'.

BINARY REGISTER NOT ($BNOT)

Macroinstruction Format

[Label] $BNOT

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
64	Not used

Operation

This instruction interrogates the contents of the binary register. If the binary register contains 0, it is set to X'00000001'. If the binary register is nonzero, it is set to X'00000000'.

TEST CONDITION ($LSET)

Macroinstruction Format

[Label] $LSET mask

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
5E	Mask

Operation

This instruction tests the contents of the condition code register. If the condition code register value (less than, equal, or greater than) satisfies the $LSET mask, the binary register is set to X'00000001'; otherwise, the binary register is set to X'00000000'.

Mask Bit Setting

Code (Hex)	Name
0004	Low
0006	Low, Equal
0002	Equal
0005	Not Equal
0003	Equal, High
0001	High

Branch Instructions

BINARY REGISTER IF ($BIF)

Macroinstruction Format

[Label] $BIF address1,address2,address3

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
04	Address of the IF block

Operation

The next instruction to be executed is located at the corresponding address if the binary register value is negative (address1), zero (address2), or positive (address3).

FLOATING-POINT REGISTER IF ($RIF)

Macroinstruction Format

[Label] $RIF address1,address2,address3

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
04	Address of the IF block

Operation

The instruction to be executed next is located at the corresponding address if the floating-point register value is negative (address1), zero (address2), or positive (address3).

BRANCH ($GOTO)

Macroinstruction Format

[Label] $GOTO address[,I]

Machine Instruction Format

Byte 1 (Op Code)	Bytes 2 and 3
02	Operand address
03	Base address for indexed instruction

Operation

The next instruction to be executed is at the effective address.

Appendix A. System/32 Scientific Instruction Set Summary

System/32 scientific programs are executed under the control of an interpreter resident in the control storage increment. The object program language, processed by the interpreter, is called the scientific instruction set. The major component of the scientific instruction set is the scientific instruction. A 3-byte scientific instruction is generated for each executable statement in the processed source string. Byte 0 contains the operation code (bits 0 through 6) and the index bit (bit 7). Bytes 1 and 2 contain a 16-bit System/32 address. The effective address for a scientific instruction is the address part of the instruction plus the scientific instruction set XR (index register) if the index bit is 1. Scientific instruction addresses consistently refer to the leftmost byte of entries in the symbol table.

The principal scientific instruction set registers are:

1. XR. Index register: A 2-byte value used in indexing for effective address.
2. XMR. Index multiplier register: 2 bytes, used for temporary storage in computing index values.
3. BR. Binary register: 4-byte two's complement register, used for integer arithmetic.
4. FR. Floating-point register: Holds short or long precision floating-point hexadecimal value in System/360 format.
5. Scientific IAR. Instruction address register: Contains 2 bytes which hold the address for the next scientific instruction to be executed.
6. AR. Address register: Holds addresses for certain scientific operands.
7. CR. Condition code register: 1 byte containing the result of a compare operation.

When control is passed to the load module for execution, the first instruction in the program entry record is a branch to the interpreter code. The interpreter locates the first scientific instruction following the branch and before decoding and executing it, sets the scientific IAR to point to the next instruction. This continues until all scientific instructions are executed. In executing the various instructions, other interpreter modules or sections of code may be used.

The following table describes the scientific instructions and operations:

Hex Value	Scientific Instruction Mnemonic	Scientific Macroinstruction Mnemonic	Functional Description
X'00'	CGO[1]	–	Sequence control for computed GOTO
X'02'	GO	$GOTO	Sequence control for GO branch
X'04'	IFGO	$BIF or $RIF	Sequence control for arithmetic IF
X'06'	XL	$XLD	Indexed register load
X'08'	XA	$XADD	Index add
X'0A'	XLI	$XLI	Index register load immediate
X'0C'	XST	$XST	Index register store
X'0E'	XM	$XMLT	Index multiply
X'10'	XMA	$XMTA	Index multiply and add
X'12'	XMLI	$MLI	Index multiplier register load immediate
X'14'	XMST	$MST	Index multiplier register store
X'16'	BST	$BST	Binary register store
X'18'	BD	$BDIV	Binary register divide
X'1A'	BA	$BADD	Binary register add
X'1C'	BS	$BSUB	Binary register subtract
X'1E'	BM	$BMLT	Binary register multiply
X'20'	BL	$BLD	Binary register load
X'22'	HST	$HST	Binary register half store
X'24'	HD	$HDIV	Binary register half divide
X'26'	HA	$HADD	Binary register half add
X'28'	HS	$HSUB	Binary register half subtract
X'2A'	HM	$HMLT	Binary register half multiply
X'2C'	HL	$HLD	Binary register half load
X'2E'	RST	$RST	Floating-point register store
X'30'	RD	$RDIV	Floating-point register divide
X'32'	RA	$RADD	Floating-point register add
X'34'	RS	$RSUB	Floating-point register subtract
X'36'	RM	$RMLT	Floating-point register multiply
X'38'	RL	$RLD	Floating-point register load
X'3A'	DST	$DST	Floating-point register double-precision store
X'3C'	DD	$DDIV	Floating-point register double-precision divide
X'3E'	DA	$DADD	Floating-point register double add
X'40'	DS	$DSUB	Floating-point register double-precision subtract
X'42'	DM	$DMLT	Floating-point register double-precision multiply
X'44'	DL	$DLD	Floating-point register double-precision load
X'46'	ADR	$ALI	Addressing operations
X'48'	INV	$INVK	Invoke branch
X'4A'	DOBGN[1]	–	DO loop initialization
X'4C'	DOEND[1]	–	DO loop variable control

Hex Value	Scientific Instruction Mnemonic	Scientific Macroinstruction Mnemonic	Functional Description
X'4E'	CALL	$CALL	Subprogram call
X'50'	IO[1]	–	Input/output control
X'52'	DED[1]	–	Data element descriptor
X'54'	DODED[1]	–	DO control variable DED
X'56'	HC	$HCMP	Binary register compare (integer*2)
X'58'	BC	$BCMP	Binary register compare (integer*4)
X'5A'	RC	$RCMP	Floating-point register compare (real*4)
X'5C'	DC	$DCMP	Floating-point register compare (real*8)
X'5E'	LSET	$LSET	Test condition code register
X'60'	AND	$BAND	Logical AND
X'62'	OR	$BOR	Logical OR
X'64'	NOT	$BNOT	Logical NOT

[1]These scientific instructions do not have macroinstruction equivalents and cannot be used by the assembler programmer.

Appendix B. Error Information

Any errors made in coding macroinstructions are flagged in the $ASMINPT file by placing an error code and an error message immediately after the macroinstruction. The error code and message are then printed on the assembly listing when the source program is assembled.

The following listing shows the error codes that may be caused by errors in macroinstructions. Other error codes may be generated by the macro processor and are caused by errors in the macroinstruction definitions. These error codes are explained in the *Basic Assembler and Macro Processor Reference Manual.*

MIC	Message
2660	MISSING FIRST ADDRESS–NSI ASSUMED
2661	MISSING SECOND ADDRESS–NSI ASSUMED
2662	MISSING THIRD ADDRESS–NSI ASSUMED
2663	SUBROUTINE ADDRESS NOT SPECIFIED
2664	NUMBER OF SUBROUTINE PARAMETERS NOT NUMERIC
2665	ADDRESS OR IMMEDIATE DATA MISSING
2666	INVALID LSET MASK
2667	INVALID INDEX SPECIFICATION

Appendix C. Hexadecimal-Decimal Conversion

Hexadecimal and Decimal Integer Conversion Table

HALFWORD								HALFWORD							
BYTE				BYTE				BYTE				BYTE			
BITS: 0123		4567		0123		4567		0123		4567		0123		4567	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
8		7		6		5		4		3		2		1	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of decimal numbers corresponding to the leftmost digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

EXAMPLE

Conversion of Hexadecimal Value	D34
1. D	3328
2. 3	48
3. 4	4
4. Decimal	3380

TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

EXAMPLE

Conversion of Decimal Value	3380
1. D	-3328
	52
2. 3	-48
	4
3. 4	-4
4. Hexadecimal	D34

To convert integer numbers greater than the capacity of table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

```
D =  13
     x16
     208
3 =  + 3
     211
     x16
    3376
4 =   +4
    3380
```

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order.

Example: $3380_{10} = X_{16}$

```
16 | 3380      remainder
16 | 211    →  4
16 | 13     →  3
            →  D
```

$3380_{10} = D34_{16}$

Hexadecimal and Decimal Fraction Conversion Table

HALFWORD															
BYTE					BYTE										
BITS 0123		4567			0123				4567						
Hex	Decimal	Hex	Decimal		Hex	Decimal			Hex	Decimal Equivalent					
.0	.0000	.00	.0000	0000	.000	.0000	0000	0000	.0000	.0000	0000	0000	0000		
.1	.0625	.01	.0039	0625	.001	.0002	4414	0625	.0001	.0000	1525	8789	0625		
.2	.1250	.02	.0078	1250	.002	.0004	8828	1250	.0002	.0000	3051	7578	1250		
.3	.1875	.03	.0117	1875	.003	.0007	3242	1875	.0003	.0000	4577	6367	1875		
.4	.2500	.04	.0156	2500	.004	.0009	7656	2500	.0004	.0000	6103	5156	2500		
.5	.3125	.05	.0195	3125	.005	.0012	2070	3125	.0005	.0000	7629	3945	3125		
.6	.3750	.06	.0234	3750	.006	.0014	6484	3750	.0006	.0000	9155	2734	3750		
.7	.4375	.07	.0273	4375	.007	.0017	0898	4375	.0007	.0001	0681	1523	4375		
.8	.5000	.08	.0312	5000	.008	.0019	5312	5000	.0008	.0001	2207	0312	5000		
.9	.5625	.09	.0351	5625	.009	.0021	9726	5625	.0009	.0001	3732	9101	5625		
.A	.6250	.0A	.0390	6250	.00A	.0024	4140	6250	.000A	.0001	5258	7890	6250		
.B	.6875	.0B	.0429	6875	.00B	.0026	8554	6875	.000B	.0001	6784	6679	6875		
.C	.7500	.0C	.0468	7500	.00C	.0029	2968	7500	.000C	.0001	8310	5468	7500		
.D	.8125	.0D	.0507	8125	.00D	.0031	7382	8125	.000D	.0001	9836	4257	8125		
.E	.8750	.0E	.0546	8750	.00E	.0034	1796	8750	.000E	.0002	1362	3046	8750		
.F	.9375	.0F	.0585	9375	.00F	.0036	6210	9375	.000F	.0002	2888	1835	9375		
1		2			3				4						

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250

Find .0B in position 2 .0429 6875

Find .00C in position 3 .0029 2968 7500

.ABC Hex is equal to .6708 9843 7500

TO CONVERT .13 DECIMAL TO HEXADECIMAL

1. **Find .1250 next lowest to** .1300
   subtract -.1250 = .2 Hex
2. **Find .0039 0625 next lowest to** .0050 0000
   -.0039 0625 = .01
3. **Find .0009 7656 2500** .0010 9375 0000
   -.0009 7656 2500 = .004
4. **Find .0001 0681 1523 4375** .0001 1718 7500 0000
   -.0001 0681 1523 4375 = .0007
   .0000 1037 5976 5625 = .2147 Hex
5. **.13 Decimal is approximately equal to** → .2147 Hex

To convert fractions beyond the capacity of table, use techniques below:

HEXADECIMAL FRACTION TO DECIMAL

Convert the hexadecimal fraction to its decimal equivalent using the same technique as for integer numbers. Divide the results by 16^n (n is the number of fraction positions).

Example: $.8A7 = .540771_{10}$

$8A7_{16} = 2215_{10}$

$16^3 = 4096$

$2215.000000 \div 4096 = .540771$

DECIMAL FRACTION TO HEXADECIMAL

Collect the integer parts of the product in the order of calculation.

Example: $.5408_{10} = .8A7_{16}$

```
           .5408
             x16
 8  <-  [8].6528
             x16
 A  <-  [10].4448
             x16
 7  <-  [7].1168
```

POWERS OF 16 TABLE

Example: $268,435,456_{10} = (2.68435456 \times 10^8)_{10} = 1000\ 0000_{16} = (10^7)_{16}$

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10 = A
17 592 186 044 416	11 = B
281 474 976 710 656	12 = C
4 503 599 627 370 496	13 = D
72 057 594 037 927 936	14 = E
1 152 921 504 606 846 976	15 = F

Decimal Values

Index

2-byte binary register add 13
2-byte binary register compare 15
2-byte binary register divide 16
2-byte binary register load 18
2-byte binary register multiply 20
2-byte binary register store 22
2-byte binary register subtract 24

READER'S COMMENT FORM

Please use this form only to identify publication errors or request changes to publications. Technical questions about IBM systems, changes in IBM programming support, requests for additional publications, etc, should be directed to your IBM representative or to the IBM branch office nearest your location.

Error in publication (typographical, illustration, and so on). **No reply.**

Page Number *Error*

Inaccurate or misleading information in this publication. Please tell us about it by using this postage-paid form. We will correct or clarify the publication, or tell you why a change is not being made, provided you include your name and address.

Page Number *Comment*

☐ Check if reply is requested.

Name ____________________

Address ____________________

____

Note: All comments and suggestions become the property of IBM.

- No postage necessary if mailed in the U.S.A.

SA21-9274-0

Fold

Fold

Cut Along Line

IBM System/32 Scientific Macroinstructions Functions Reference Manual (File No. S32-36) Printed in USA SA21-9274-0

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

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Development Laboratory
Publications, Dept. 245
Rochester, Minnesota 55901

Fold

Fold

IBM®

International Business Machines Corporation

General Systems Division
5775D Glenridge Drive N. E.
P.O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
U.S.A.
(International)

IBM

International Business Machines Corporation

General Systems Division
5775D Glenridge Drive N. E.
P.O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
U.S.A.
(International)

IBM System/32 Scientific Macroinstructions Functions Reference Manual (File No. S32-36) Printed in USA SA21-9274-0

SA21-9274-0