SA21-9275-0

File No. S34/S32-36

IBM System/34 and IBM System/32 Scientific Macroinstructions Functions Reference Manual

Program Numbers 5725-SC1
5726-SS1

SA21-9275-0

File No. S34/S32-36

IBM System/34 and IBM System/32 Scientific Macroinstructions Functions Reference Manual

Program Numbers 5725-SC1
5726-SS1

SA21-9275-0

File No. S34/S32-36

IBM System/34 and IBM System/32 Scientific Macroinstructions Functions Reference Manual

Program Numbers 5725-SC1
5726-SS1

First Edition (July 1978)

This publication obsoletes and replaces SA21-9274.

This edition applies to release 2 modification 0, of the IBM System/34 System Support Program Product (Program Number 5726-SS1) and to release 7 modification 0, of the IBM System/32 System Control Program Product (Program Number 5725-SC1) and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, refer to the latest IBM System Bibliography for the editions that are applicable and current; for System/34, see GH30-0231, and for System/32, see GC20-0032.

Preface

This reference manual is for computer programmers, system analysts, system engineers, and other technical people who are interested in the operation and characteristics of the IBM System/34 and System/32 scientific macroinstructions at the machine code level.

Note: This manual follows the convention that *he* means *he* or *she*.

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/34 and IBM System/32 FORTRAN IV Logic Manual,* LY21-0568
- *IBM System/34 System Support Reference Manual,* SC21-5155
- *IBM System/34 Functions Reference Manual,* SA21-9243
- *IBM System/34 Basic Assembler and Macro Processor Reference Manual,* SC21-7705
- *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

Titles and abstracts of other related publications are listed in the *IBM System/34 Bibliography,* GH30-0231 or the *IBM System/32 Bibliography,* GC20-0032.

Contents

CHAPTER 1. INTRODUCTION **1**
Scientific Macroinstruction Statements 2
Data Formats 2
  Binary Format 2
  Floating-Point Format 2
Addressing 2
  Direct Addressing 2
  Indexed Addressing 2
Machine Instruction Format 3
Registers 4

CHAPTER 2. SCIENTIFIC MODE LINKAGE AND SUPPORT MACROINSTRUCTIONS **7**
Scientific Environment and Subroutine Linkage 7
  Scientific Implementation ($CSET) 9
  Execute Scientific Subroutine ($CALL) 9
  Start Scientific Subroutine ($CSUB) 9
  Exit Scientific Subroutine ($CRTN) 9
  Return to System Mode ($INVK) 9
  Return to Scientific Mode ($CRSM) 9
  Enter Scientific Microcode ($CNTR) 10

CHAPTER 3. INSTRUCTIONS **13**
ADDRESS REGISTER INSTRUCTIONS 13
Address Register Load ($ALI, $ALD) 13
  Macroinstruction Format 13
  Machine Instruction Format 13
  Operation 13
  Example (Nonindexed) ($ALD) 13
  Example (Indexed) ($ALI) 13
BINARY REGISTER INSTRUCTIONS 14
  Data Format 14
  Number Representation 14
  Instruction Format 15
Binary Register Add ($HADD)–2 Bytes 15
  Macroinstruction Format 15
  Machine Instruction Format 15
  Operation 15
  Example (Nonindexed) 15
  Example (Indexed) 16
Binary Register Add ($BADD)–4 Bytes
  Macroinstruction Format 16
  Machine Instruction Format 16
  Operation 16
  Example (Nonindexed) 16
  Example (Indexed) 17
Binary Register Compare ($HCMP)–2 Bytes 17
  Macroinstruction Format 17
  Machine Instruction Format 17
  Operation 17
  Resulting Condition Code Register Settings 17
Binary Register Compare ($BCMP)–4 Bytes 18
  Macroinstruction Format 18
  Machine Instruction Format 18
  Operation 18
  Resulting Condition Code Register Settings 18
Binary Register Divide ($HDIV)–2 Bytes 18
  Macroinstruction Format 18
  Machine Instruction Format 18
  Operation 18
  Example (Nonindexed) 18
  Example (Indexed) 19
Binary Register Divide ($BDIV)–4 Bytes
  Macroinstruction Format 19
  Machine Instruction Format 19
  Operation 19
  Example (Nonindexed) 19
  Example (Indexed) 20
Binary Register Load ($HLD)–2 Bytes 20
  Macroinstruction Format 20
  Machine Instruction Format 20
  Operation 20
  Example (Nonindexed) 20
  Example (Indexed) 21
Binary Register Load ($BLD)–4 Bytes 21
  Macroinstruction Format 21
  Machine Instruction Format 21
  Operation 21
  Example (Nonindexed) 21
  Example (Indexed) 22
Binary Register Multiply ($HMLT)–2 Bytes 22
  Macroinstruction Format 22
  Machine Instruction Format 22
  Operation 22
  Example (Nonindexed) 23
  Example (Indexed) 23
Binary Register Multiply ($BMLT)–4 Bytes 23
  Macroinstruction Format 23
  Machine Instruction Format 23
  Operation 23
  Example (Nonindexed) 24
  Example (Indexed) 24
Binary Register Store ($HST)–2 Bytes 24
  Macroinstruction Format 24
  Machine Instruction Format 24
  Operation 24
  Example (Nonindexed) 24
  Example (Indexed) 25
Binary Register Store ($BST)–4 Bytes 25
  Macroinstruction Format 25
  Machine Instruction Format 25
  Operation 25
  Example (Nonindexed) 25
  Example (Indexed) 26

Binary Register Subtract ($HSUB)–2 Bytes 26
Macroinstruction Format 26
Machine Instruction Format 26
Operation 26
Example (Nonindexed) 26
Example (Indexed) 27
Binary Register Subtract ($BSUB)–4 Bytes 27
Macroinstruction Format 27
Machine Instruction Format 27
Operation 27
Example (Nonindexed) 27
Example (Indexed) 28
FLOATING-POINT REGISTER INSTRUCTIONS 29
Data Format 29
Single-Precision Floating-Point Number 29
Double-Precision Floating-Point Number 30
Number Representation 30
Normalization 30
Instruction Format 31
Floating-Point Register Add ($RADD)–Single-Precision . . 31
Macroinstruction Format 31
Machine Instruction Format 31
Operation 31
Example (Nonindexed) 32
Example (Indexed) 32
Floating-Point Register Add ($DADD)–Double-Precision . 32
Macroinstruction Format 32
Machine Instruction Format 32
Operation 32
Example (Nonindexed) 33
Example (Indexed) 33
Floating-Point Register Compare ($RCMP)–Single-Precision 33
Macroinstruction Format 33
Machine Instruction Format 33
Operation 33
Resulting Condition Register Settings 33
Floating-Point Register Compare ($DCMP)–Double-Precision 34
Macroinstruction Format 34
Machine Instruction Format 34
Operation 34
Resulting Condition Register Settings 34
Floating-Point Register Divide ($RDIV)–Single-Precision . 34
Macroinstruction Format 34
Machine Instruction Format 34
Operation 34
Example (Nonindexed) 35
Example (Indexed) 35
Floating-Point Register Divide ($DDIV)–Double-Precision . 35
Macroinstruction Format 35
Machine Instruction Format 35
Operation 35
Example (Nonindexed) 36
Example (Indexed) 36
Floating-Point Register Load ($RLD)–Single-Precision . . 36
Macroinstruction Format 36
Machine Instruction Format 36
Operation 36
Example (Nonindexed) 36
Example (Indexed) 37
Floating-Point Register Load ($DLD)–Double-Precision . . 37
Macroinstruction Format 37
Machine Instruction Format 37
Operation 37
Example (Nonindexed) 37
Example (Indexed) 38
Floating-Point Register Multiply ($RMLT)–Single-Precision 38
Macroinstruction Format 38
Machine Instruction Format 38
Operation 38
Example (Nonindexed) 39
Example (Indexed) 39
Floating-Point Register Multiply ($DMLT)–Double-Precision 39
Macroinstruction Format 39
Machine Instruction Format 39
Operation 39
Example (Nonindexed) 40
Example (Indexed) 40
Floating-Point Register Store ($RST)–Single-Precision . . 40
Macroinstruction Format 40
Machine Instruction Format 40
Operation 40
Example (Nonindexed) 40
Example (Indexed) 41
Floating-Point Register Store ($DST)–Double-Precision . . 41
Macroinstruction Format 41
Machine Instruction Format 41
Operation 41
Example (Nonindexed) 41
Example (Indexed) 42
Floating-Point Register Subtract ($RSUB)–Single-Precision 42
Macroinstruction Format 42
Machine Instruction Format 42
Operation 42
Example (Nonindexed) 42
Example (Indexed) 43
Floating-Point Register Subtract ($DSUB)–Double-Precision 43
Macroinstruction Format 43
Machine Instruction Format 43
Operation 43
Example (Nonindexed) 43
Example (Indexed) 44
INDEX MULTIPLIER REGISTER INSTRUCTIONS 45
Index Multiplier Register Load Immediate ($MLI) 45
Macroinstruction Format 45
Machine Instruction Format 45
Operation 45
Example (Nonindexed) 45
Example (Indexed) 45
Index Multiplier Register Store ($MST) 46
Macroinstruction Format 46
Machine Instruction Format 46
Operation 46
Example (Nonindexed) 46
Example (Indexed) 46
INDEX REGISTER INSTRUCTIONS 47
Index Register Add ($XADD) 47
Macroinstruction Format 47
Machine Instruction Format 47
Operation 47
Example (Nonindexed) 47
Example (Indexed) 47
Index Register Load ($XLD) 48
Macroinstruction Format 48
Machine Instruction Format 48
Operation 48
Example (Nonindexed) 48
Example (Indexed) 48

Index Register Load Immediate ($XLI) 49
Macroinstruction Format 49
Machine Instruction Format 49
Operation 49
Example (Nonindexed) 49
Example (Indexed) 49
Index Register Multiply ($XMLT) 50
Macroinstruction Format 50
Machine Instruction Format 50
Operation 50
Example (Nonindexed) 50
Example (Indexed) 50
Index Register Multiply and Add ($XMTA) 51
Macroinstruction Format 51
Machine Instruction Format 51
Operation 51
Example (Nonindexed) 51
Example (Indexed) 51
Index Register Store ($XST) 52
Macroinstruction Format 52
Machine Instruction Format 52
Operation 52
Example (Nonindexed) 52
Example (Indexed) 52
LOGICAL INSTRUCTIONS 53
Binary Register AND ($BAND) 53
Macroinstruction Format 53
Machine Instruction Format 53
Operation 53
Binary Register OR ($BOR) 53
Macroinstruction Format 53
Machine Instruction Format 53
Operation 53
Binary Register NOT ($BNOT) 53
Macroinstruction Format 53
Machine Instruction Format 53
Operation 53
Test Condition ($LSET) 54
Macroinstruction Format 54
Machine Instruction Format 54
Operation 54
Mask Bit Setting 54
BRANCH INSTRUCTIONS 55
Binary Register IF ($BIF) 55
Macroinstruction Format 55
Machine Instruction Format 55
Operation 55
Floating-Point Register IF ($RIF) 55
Macroinstruction Format 55
Machine Instruction Format 55
Operation 55
Branch ($GOTO) 55
Macroinstruction Format 55
Machine Instruction Format 55
Operation 55
CONVERSION INSTRUCTIONS 56
Single-Precision to Binary ($RCNB) 56
Macroinstruction Format 56
Machine Instruction Format 56
Operation 56
Double-Precision to Binary ($DCNB) 56
Macroinstruction Format 56
Machine Instruction Format 56
Operation 56
Binary to Single-Precision ($BCNR) 56
Macroinstruction Format 56
Machine Instruction Format 56
Operation 56
Binary to Double-Precision ($BCND) 56
Macroinstruction Format 56
Machine Instruction Format 56
Operation 56
Sign Change Binary ($BSGN) 57
Macroinstruction Format 57
Machine Instruction Format 57
Operation 57
Sign Change Floating-Point ($RSGN) 57
Macroinstruction Format 57
Machine Instruction Format 57
Operation 57
Binary Absolute ($BABS) 57
Macroinstruction Format 57
Machine Instruction Format 57
Operation 57
Floating-Point Absolute ($RABS) 57
Macroinstruction Format 57
Machine Instruction Format 57
Operation 57

APPENDIX A. HEXADECIMAL-DECIMAL CONVERSION 59

APPENDIX B. SUMMARY OF SYSTEM/34 AND SYSTEM/32 SCIENTIFIC INSTRUCTION SET 61

APPENDIX C. ERROR INFORMATION 67

APPENDIX D. HOW TO SPECIFY THE OLINK PROCEDURE 69

INDEX 71

Chapter 1. Introduction

FORTRAN is supported on System/34 and System/32 through a set of scientific instructions. The scientific instructions consist of a group of instructions and registers that perform functions commonly required in scientific programs. The instructions are used by FORTRAN and require no programming action other than writing FORTRAN language statements.

You can also use a subset of the scientific instructions with the IBM System/34 or System/32 Basic Assembler and Macro Processor. In order to use this subset of scientific instructions with the System/32, however, you must have feature 1500 (this feature consists of the control storage increments, and scientific microcode required by FORTRAN) installed prior to system generation time. You must also have the subset of scientific instructions installed in either system at system generation time.

The subset (defined in *Appendix B. System/34 and System/32 Scientific Instructions Summary*) consists of the set of mnemonic macroinstructions that is compatible with the set of mnemonic scientific instructions. The macroinstructions are symbolic source statements that are expanded into a predetermined sequence of object codes by the Basic Assembler and Macro Processor. That is, the macroinstructions generate the scientific instructions that perform the specific functions necessary for scientific calculations. The macroinstructions expand the assembler user's ability to add, subtract, multiply, and divide binary and floating-point data.

To use the scientific instructions with an assembler program, the programmer must establish the scientific mode environment and create the interface by using the following macroinstructions (see Chapter 2 for a description of each of these macroinstructions):

$CSET
$CALL
$CSUB
$CRTN
$INVK
$CRSM
$CNTR

After the environment has been established and the interface has been created, the programmer must code the macroinstructions required to generate the scientific instructions that are needed to meet your particular requirements.

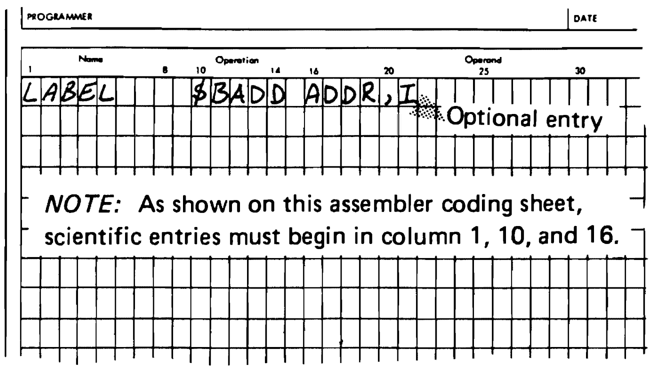
The remainder of this manual provides you with the information necessary to use scientific macroinstructions with the IBM System/34 or System/32 Basic Assembler and Macro Processor. The terms *System/34 and System/32 mode* and *scientific mode* are used throughout this manual to describe which processor executes either a series of instructions or a subroutine.

When the system is using either the system support program (System/34) or the system control program (System/32) to execute the instructions described in the *System/34 and System/32 Functions Reference Manual*, it is in system mode and the scientific macroinstructions are invalid. Likewise, when the system is using the scientific microcode to execute the instructions described in this manual, it is in scientific mode and the system instructions are invalid.

Note: For the remainder of this manual, the term *System(s)* applies to both System/34 and System/32. Also, when manuals containing information about either system are referenced, the titles are not preceded by the appropriate system references. In addition, when the *System/34 and System/32 Functions Reference Manual* is referenced, the title is not preceded by the system reference. That is, the title is simply given as the *Functions Reference Manual.*

SCIENTIFIC MACROINSTRUCTION STATEMENTS

Scientific macroinstructions are symbolic source statements that are expanded into a predetermined sequence of object codes by the Basic Assembler and Macro Processor; this sequence of codes is then executed by the scientific microcode. The format of a scientific macroinstruction is as follows:


PROGRAMMER DATE

Name	Operation	Operand
LABEL	$BADD	ADDR,I

Optional entry

NOTE: As shown on this assembler coding sheet, scientific entries must begin in column 1, 10, and 16.

DATA FORMATS

Data resides in main storage in 8-bit bytes. In any instruction, data is represented as a positive or negative number by the value in bit 0. If bit 0 is 0, the data is positive. If bit 0 is 1, the data is negative.

The instruction the system is executing determines how the data is interpreted.

Binary Format

Binary data is recorded in either a 2-byte or a 4-byte format. Both formats use bit 0 as the sign bit followed by the integer field in bits 1-7. 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

Addresses contained in main storage are 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.

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.

Suppose you were to code the following instruction:

```
NAMEA   $BLD     FIELDA
```

If FIELDA points to 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 equal to the sum of the current contents of the scientific index register and the current contents of the address portion of the instruction.

Suppose you were to code the following instruction:

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

If FIELDA points to 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.1

MACHINE INSTRUCTION FORMAT

All of the 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 instruction, and bit 7 specifies the type of addressing to be used: 0 = direct addressing, and 1 = indexed addressing.

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

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	$RDL	$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 included in the following chart:

Instruction	Function
$GOTO	Changes the execution sequence to the instruction at the effective address.
$LSET	Sets the binary register according to the condition code register contents and the instruction mask.
$CALL	Executes a scientific subroutine.
$CSET	Establishes the scientific environment (System/34). Loads the scientific microcode (System/32).
$CSUB	Starts a scientific subroutine.
$CRTN	Exits a scientific subroutine.
$INVK	Changes to system mode execution beginning at the effective address.
$CRSM	Reenters the scientific environment.
$CNTR	Enters the scientific microcode.
$RCNB	Converts a single-precision, floating-point number to a binary number.
$DCNB	Converts a double-precision, floating-point number to a binary number.
$BCNR	Converts a binary number to a single-precision, floating-point number.
$BCND	Converts a binary number to a double-precision, floating-point number.
$BSGN	Changes the sign of the binary number.
$RSGN	Change the sign of the floating-point number.
$BABS	Provides the absolute binary value.
$RABS	Provides the absolute floating-point value.

REGISTERS

The 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 (multiply and add) and $XMLT (multiply) 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 (store) 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. During computation, a guard digit and a status indicator for single-precision or double-precision numbers are associated with the floating-point register. 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; this means that the high-order hexadecimal digit of the mantissa is nonzero.

The floating-point register status is set to double-precision whenever a single-precision or double-precision operation is performed (except for $RLD) and the prior status was double-precision. The status is set to single-precision by the $RLD (load) instruction; it 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 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 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 the CMP (compare) instruction. The $LSET (test condition) instruction is the only instruction provided to test the content of the condition code register.

Chapter 2. Scientific Mode Linkage And Support Macroinstructions

SCIENTIFIC ENVIRONMENT AND SUBROUTINE LINKAGE

Scientific mode linkage and scientific support macroinstructions provide the interface between system mode routines and scientific mode routines. During scientific mode, the system XR1 (index register 1) addresses the scientific communication area, and the system XR2 (index register 2) addresses the current save area for the executing scientific program. (See *Appendix D. How to Specify the OLINK Procedure* for detailed information regarding the OCL necessary to link edit a module that uses scientific macroinstructions; see the *FORTRAN IV Logic Manual* for detailed information regarding the scientific communication area.)

The following chart gives an overview of the macroinstructions that are used to implement the subroutine linkage in the scientific macroinstruction package for each system. This chart is followed by an example scientific program and a detailed description of each of these macroinstructions.

Macroinstruction	System/34	System/32
$CSET	Used once in the main program to establish the environment for scientific mode processing. The expansion includes the scientific communication area and the main program save area.	Used once in the main program to generate the code necessary to load the scientific microcode. The expansion includes the scientific communication area and the main program save area.
$CALL	Generates the linkage to scientific subroutines and passes required addresses (arguments). Scientific subroutine linkage conventions require that called subroutines must be external, separately assembled programs.	Same as System/34.
$CSUB	Establishes the subroutine linkage and makes the received parameters (arguments) from $CALL available as parameters for the subroutine. The index values (parameter addresses) are variables that are generated by $CSUB. These variables are named $ARGnn; nn represents the position of the desired argument within the parameter list. That is, in a subroutine in which $CALL passes three arguments and $CSUB specifies that three parameters are to be received, the variables generated by $CSUB are $ARG1, $ARG2, and $ARG3.	Same as System/34.
$CRTN	Returns execution control to the calling routine.	Same as System/34.
$INVK	Switches to system mode processing.	Same as System/34.
$CRSM	Allows the user to return to scientific mode without destroying the environment that was established by $CSET. It also allows the user to resume processing.	Same as System/34.
$CNTR	Switches to scientific mode processing.	Same as System/34.

Scientific Implementation ($CSET)

[Label] $CSET

System/34

This instruction must be the first scientific macroinstruction executed. It creates the environment and the instructions necessary to facilitate the first entry into scientific mode. Also, the next sequential instruction must be a scientific instruction. The expansion of the instruction includes the scientific communication area and the main program save area. The $CSET macroinstruction is issued in system mode and allows the program to execute in scientific mode. There are no inputs to the $CSET macroinstruction.

System/32

This instruction is used only once in the program. It is used to generate the code necessary to load the scientific microcode. The expansion includes the scientific communication area and the main program save area. It saves XR1 and XR2 if they contained data before $CSET used them. If the macroinstruction is unable to locate and load the scientific instruction set microcode, control is passed to $MODERR ($MODERR must be a user defined error recovery subroutine). Failure to define $MODERR results in an assembly error.

Execute Scientific Subroutine ($CALL)

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

This instruction causes the specified external scientific subroutine to be executed by using the variables at the specified addresses as parameters. When the external scientific subroutine completes execution, control is returned and execution resumes with the next scientific macroinstruction. The $CALL macroinstruction builds a call block for the specified subroutine; the call block contains the list of addresses specified as arguments to the macroinstruction. (For additional information on call blocks, see *Linkage Conventions* in the *FORTRAN IV Logic Manual*). The $CALL macroinstruction is issued in scientific mode.

Start Scientific Subroutine ($CSUB)

[Label] $CSUB number

This instruction generates the code that is necessary to establish receiving subroutine linkage. The label specifies the entry point name, and the number specifies the number of parameters to be received by the subroutine. The macroinstruction builds save areas and parameter lists for the subroutine. (Further information on the save areas and parameter lists can be found in *Linkage Conventions* in the *FORTRAN IV Logic Manual*). The $CSUB macroinstruction is issued in scientific mode.

Exit Scientific Subroutine ($CRTN)

[Label] $CRTN

This instruction generates the code that is necessary to return execution control to a calling routine from a scientific subroutine. The $CRTN macroinstruction is issued in scientific mode.

Return to System Mode ($INVK)

[Label] $INVK address

This instruction transfers the program to system mode and continues execution with the next program instruction at the effective address. The operand of $INVK cannot be indexed. The $INVK macroinstruction is issued in scientific mode.

Return to Scientific Mode ($CRSM)

[Label] $CRSM

This macroinstruction allows the user to leave system mode and return to the instruction following the $INVK macroinstruction in the environment that was established by the $CSET macroinstruction. The control storage portion of the scientific communications area cannot be accessed by the user's program by any means other than the scientific macroinstructions.

Enter Scientific Microcode ($CNTR)

[Label] $CNTR

The $CNTR macroinstruction generates the code that is 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.

Following is an example System/32 scientific program using the various macroinstructions. The first portion of the program is the separately assembled subroutine. This program is also applicable for System/34; the System/34 scientific macroinstructions, however, are loaded at hexadecimal location 0000 rather than at hexadecimal location 0800, which is used for System/32.

IBM

IBM System/32 Basic Assembler Coding Form

GX21-9245
Printed in U.S.A.

PROGRAM			PAGE OF
PROGRAMMER	DATE		

```
EXTPGM   START Ø
ARR      EQU   8
PGMEXT   $CSUB 2                       SUBROUTINE WITH 2 PARMS
         $XLD  $ARG2                   GET SECOND PARM ADDRESS
         $BLD  Ø,I                     GET BINARY NUMBER
         $HDIV TWO                     DIVIDE BY TWO
         $XLD  $ARG1                   FIRST PARM ADDRESS
         $BST  Ø,I                     ANSWER TO CALL VARIABLE
         $CRTN                         RETURN TO CALLER
TWO      EQU   *
         DC    AL2(2)
         END
PROG32   START X'8ØØ'
         EXTRN PGMEXT
         $CSET                         LOAD SCIENTIFIC MICROCODE
* ENTER SCIENTIFIC CODE
         $CNTR                         ENTER SCIENTIFIC CODE
         $CALL PGMEXT,WRKAR,SEVENT     GO TO SUBROUTINE - WILL RETURN TO NEXT INST
         $BLD  WRKAR                   RESULTS TO BINARY REGISTER
         $BSUB THREE3                  SUBTRACT 3 FROM RESULTS
         $BIF  BAD,GOOD,BAD            TEST RESULT FOR ZERO
BAD      $HLD  STATUS                  GET STATUS
         $HADD BIT1                    SET ERROR BIT ONE ON
         $HST  STATUS                  STATUS TO STORAGE
GOOD     $INVK TERMINAT                RETURN TO S/32 MODE
* ENTER S/32 MODE
TERMINAT $EOJ
STATUS   EQU   *                       STATUS OF PROG
         DC    XL2'ØØØØ'
BIT1     EQU   *                       FIRST BIT
         DC    XL2'8ØØØ'
WRKAR    EQU   *                       WORK AREA
         DC    XL8'Ø'
SEVEN7   EQU   *                       CONSTANT 7
         DC    XL4'7'
THREE3   EQU   *                       CONSTANT 3
         DC    XL4'3'
         END   PROG32
```

Chapter 3. Instructions

Address Register Instructions

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

ADDRESS REGISTER LOAD ($ALI, $ALD)

Macroinstruction Format

```
[Label] $ALI address ,I
[Label] $ALD 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 effective 2-byte address in the address register.

Example (Nonindexed) ($ALD)

Instruction

```
460013
```

Address Register Before Operation

```
11001111  10110011
Byte 0    Byte 1
```

Address Register After Operation

```
00000000  00010011
Byte 0    Byte 1
```

Example (Indexed) ($ALI)

Instruction

```
470013
```

Index Register

```
00000000  00000101
Byte 0    Byte 1
```

Address Register Before Operation

```
11001111  10110011
Byte 0    Byte 1
```

Address Register After Operation

```
00000000  00011000
Byte 0    Byte 1
```

Binary Register Instructions

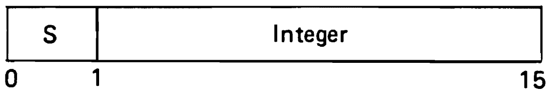
The binary register instructions perform binary arithmetic on operands serving as fixed-point data, base addresses and index quantities. The operands are 32 bits long including the sign bit. An operand value is always in the binary register and the operand value specified in the instruction is always in main storage. After the instruction has been completed, the binary register contains the result.

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

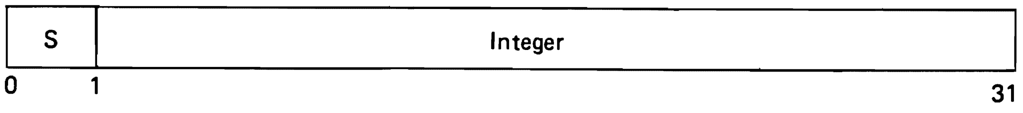
Data Format

Binary numbers appear in a fixed-length format consisting of the 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



Fixed-Point Number—4 Bytes



Binary data in main storage appears in a 32-bit format or a 16-bit format; each format has 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 positive or negative numbers. Positive numbers are represented in true binary notation with a sign bit of 0. Negative numbers are represented in twos-complement notation with a sign bit of 1. (The twos complement of a number is obtained by inverting each bit of the number and adding 1 to the result.)

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 propagating the sign bit.

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.

Note: 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. If bit 7 is 0, 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 operand location in main storage.

The results of binary instructions replace the contents of the binary register; an exception is the store instruction, which replaces the contents of the main storage location with the contents of the register.

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 results.

BINARY REGISTER ADD ($HADD)–2 BYTES

Macroinstruction Format

[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 yields 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
```

Operand After Indexing

```
00001100  10100011
14FD      14FE
```

Index Register

```
00000000  00111010
Byte 0    Byte 1
```

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. 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 yields a positive final sum. An overflow is not flagged, nor does a program interrupt occur.

Example (Nonindexed)

Instruction

```
1A   0C   14
```

Operand

```
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

```
1B0C14
```

Operand Before Indexing

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

Operand After Indexing

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

Index Register

```
00001111 01010000
Byte 0   Byte 1
```

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 to 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 positive or negative numbers.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Code Register Settings

Bit	Name	Condition Indicated
5	Low	The binary register value is less than the operand value.
6	Equal	The values are equal.
7	High	The 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 to low, equal, or high. Comparison is algebraic, and both operands are treated as 32-bit positive or negative numbers.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Code Register Settings

Bit	Name	Condition Indicated
5	Low	The binary register value is less than the operand value.
6	Equal	The values are equal.
7	High	The 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 positive or negative numbers. The quotient is a 32-bit positive or negative number; it replaces the dividend in the binary register. (Dividing by zero always yields zero.) 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
```

Operand Before Indexing

```
00000000 00000101
049E     049F
```

Operand After Indexing

```
00000000 00001010
04A9     04AA
```

Index Register

```
00000000 00001011
Byte 0   Byte 1
```

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 positive or negative numbers. The quotient is a 32-bit positive or negative number; it replaces the dividend in the binary register. (Dividing by zero always yields zero.) 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
```

Operand Before Indexing

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

Operand After Indexing

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

Index Register

```
00000000 01100111
Byte 0   Byte 1
```

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

```
 -D 02 C1
```

Operand Before Indexing

```
 01100011  10100011
 02C1      02C2
```

Operand After Indexing

```
 10100011  00111010
 02F2      02F3
```

Index Register

```
 00000000  00110001
 Byte 0    Byte 1
```

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

Operand Before Indexing

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

Operand After Indexing

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

Index Register

00000010	00111000
Byte 0	Byte 1

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 positive or negative numbers. The product is always a 32-bit positive or negative number; it replaces the multiplicand in the binary register. The sign of the product is determined by the signs of the multiplier and multiplicand; 0, however, 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 deleted and are lost.

Example (Nonindexed)

Instruction

2A1093

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

2B1093

Operand Before Indexing

00000000	00000111
1093	1094

Operand After Indexing

00000000	00001001
109F	10A0

Index Register

00000000	00001100
Byte 0	Byte 1

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 positive or negative numbers. The product is always a 32-bit positive or negative number; it replaces the multiplicand in the binary register. The sign of the product is determined by the signs of the multiplier and multiplicand; 0, however, 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 deleted and are lost.

Example (Nonindexed)

Instruction

```
1E01C4
```

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

```
1F01C4
```

Operand Before Indexing

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

Operand After Indexing

```
00000000 00000000 00000001 00110110
0267     0268     0269     026A
```

Index Register

```
00000000 10100011
Byte 0   Byte 1
```

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

```
220436
```

Operand Before Operation

```
00011000 11110111
0436     0437
```

Operand After Operation

```
10100011 11001001
0436     0437
```

Binary Register Before and After Operation

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

Example (Indexed)

Instruction

23 04 36

Operand Before Indexing

00011000	11110111
0436	0437

Operand After Indexing (before operation)

10011001	01100110
04C9	04CA

Operand After Operation

10100101	00111100
04C9	04CA

Index Register

00000000	10010011
Byte 0	Byte 1

Binary Register Before and After Operation

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

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 0C19

Operand Before Operation

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

Operand After Operation

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

Binary Register Before and After Operation

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

Example (Indexed)

Instruction

```
17 0C19

Byte 0    Byte 1    Byte 2    Byte 3
```

Operand Before Indexing

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

Operand After Indexing (before operation)

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

Operand After Operation

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

Index Register

```
00000000  00111010
Byte 0    Byte 1
```

Binary Register Before and After Operation

```
00000000  01001101  00111010  11000101
```

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

Operand Before Indexing

00001011	01101100
0319	031A

Operand After Indexing

00000100	01110111
03ED	03EE

Index Register

00000000	11010100
Byte 0	Byte 1

Binary Register Before Operation

00000000	00001100	00111100	11000111
Bvte 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
```

Operand Before Indexing

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

Operand After Indexing

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

Index Register

```
00000000  01100011
Byte 0    Byte 1
```

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 large numbers while preserving computational precision.

A floating-point number consists of a positive or negative characteristic and a positive or negative mantissa. The quantity expressed by this number is equal to 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 to the left of the high-order digit (see *Number Representation* later in this chapter).

The floating-point instructions allow 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*, later in this chapter). Normalized operands are used in any floating-point operation.

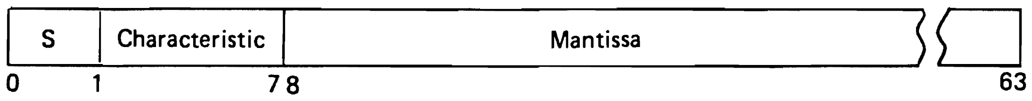
Data Format

Floating-point data appears in a fixed-length format, that is either single-precision or double-precision. 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



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 format and 14 mantissa digits in double-precision format.

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 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, 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. That is, the number is positive or negative according to the sign bit (0 or 1).

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

$16^{-65} \leq M \leq (1 - 16^{-6})\ 16^{63}$ in single precision

$16^{-65} \leq M \leq (1 - 16^{-14})\ 16^{63}$ in double precision

$5.4\ 10^{-79}\ ^{M\ 7.2\ \ 1075}$ in either precision.

Normalization

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

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. If a number has no fraction, it is considered to be *normalized*. However, if one or more high-order mantissa digits are 0, the number is said to be *unnormalized*.

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

Programming Note: Because 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. A 1 in bit 7 indicates that 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.

Before two floating-point numbers (operands) can be added, the characteristics of the two operands must be compared. The mantissa with the smaller characteristic is shifted right; the operand with the smaller characteristic is increased by 1 for each hexadecimal digit that has been shifted 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 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

```
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
```

Operand Before Indexing

```
40   10   24   00
1432 1433 1434 1435
```

Operand After Indexing

```
40   11   93   01
1566 1567 1568 1569
```

Index Register

```
01     34
Byte 0 Byte 1
```

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.

Before two floating-point numbers (operands) can be added, the characteristics of the two operands must be compared. The mantissa with the smaller characteristic is shifted right; the operand with the smaller characteristic is increased by 1 for each hexadecimal digit that has been shifted 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

```
3E0347
```

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

```
3F0347
```

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
```

Index Register

```
01       10
Byte 0   Byte 1
```

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 to low, equal, or high. The 4 low-order bytes of the floating-point register are ignored. During the comparison, the sign, characteristic, and mantissa of each number are taken into account.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Register Settings

Bit	Name	Condition Indicated
5	Low	The binary register value is less than the operand value.
6	Equal	The values are equal.
7	High	The 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 to low, equal, or high. During the comparison, the sign, characteristic, and mantissa of each number are taken into account.

Programming Note: Neither operand is altered by the instruction.

Resulting Condition Register Settings

Bit	Name	Condition Indicated
5	Low	The binary register value is less than the operand value.
6	Equal	The values are equal.
7	High	The 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. The sign of the quotient is determined by the rules of algebra. If the data is 0, the divide check indicator is set in the scientific communication area, and the dividend is left unchanged in the floating-point register.

Before two floating-point numbers can be divided, the difference between the dividend characteristic and the divisor characteristic plus 64 must be obtained. The number is used as an intermediate characteristic.

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
```

Operand Before Indexing

```
37   E0   00   00
1631 1632 1633 1634
```

Operand After Indexing

```
35   E0   00   00
1665 1666 1667 1668
```

Index Register

```
00      34
Byte 0  Byte 1
```

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 Format

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. The sign of the quotient is determined by the rules of algebra. If the data is 0, the divide check indicator is set in the scientific communication area, and the dividend is left unmodified in the floating-point register.

Before two floating-point numbers can be divided, the difference between the dividend characteristic and the divisor characteristic plus 64 must be obtained. This number is used as an intermediate characteristic.

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

```
3C1743
```

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

```
3D1743
```

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
```

Index Register

```
01        01
Byte 0    Byte 1
```

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 to single precision.

Example (Nonindexed)

Instruction

```
380462
```

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
```

Operand Before Indexing

```
40   36   93   02
0462 0463 0464 0465
```

Operand After Indexing

```
41   27   08   00
0474 0475 0476 0477
```

Index Register

```
00      12
Byte 0  Byte 1
```

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 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
```

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
```

Index Register

```
00      20
Byte 0  Byte 1
```

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 sign of the product is determined by the rules of algebra.

Before two floating-point numbers can be multiplied, the sum of the characteristics minus 64 must be obtained. This number is used as the characteristic of the product. The product characteristic is reduced by the number of left shifts.

The product fraction is normalized, if necessary. The product fraction is truncated to 6 digits after normalization. When the product fraction is zero, the product sign and characteristic are made zeros, thereby 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
```

Operand Before Indexing

```
41   F0   00   00
0642 0643 0644 0645
```

Operand After Indexing

```
39   B0   00   00
0685 0686 0687 0688
```

Index Register

```
00      43
Byte 0  Byte 1
```

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 sign of the product is determined by the rules of algebra.

Before two floating-point numbers can be multiplied, the sum of the characteristics minus 64 must be obtained. This number is used as the characteristic of the product. The product characteristic is reduced by the number of left shifts.

The product fraction is determined by the rules of algebra. The product fraction is truncated to 14 digits after normalization. When the product fraction is zero, the product sign and characteristic are made zeros, thereby 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
```

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
```

Index Register

```
01       00
Byte 0   Byte 1
```

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 8-byte area starting at the effective address.

Example (Nonindexed)

Instruction

```
2E 01 23
```

Operand Before Operation

```
41   92   36   08
0123 0124 0125 0126
```

Operand After Operation

```
39   08   42   60
0123 0124 0125 0126
```

Floating-Point Register Before and After Operation

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

Example (Indexed)

Instruction

```
2F 01 23
```

Operand Before Indexing

```
41   92   36   08
0123 0124 0125 0126
```

Operand After Indexing (before operation)

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

Operand After Operation

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

Index Register

```
00      08
Byte 0  Byte 1
```

Floating-Point Register Before and After Operation

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

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
```

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
```

Floating-Point Register Before and After Operation

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

Example (Indexed)

Instruction

```
3B48 03
```

Operand Before Indexing

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

Operand After Indexing (before operation)

```
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
```

Index Register

```
03      8A
Byte 0  Byte 1
```

Floating-Point Register Before and After Operation

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

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 ($RADD),* except that the sign of the operand is changed before addition when the ($RADD) instruction is used. The sign of the difference is determined by the rules of algebra. The sign for any difference that equals 0 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
```

Operand Before Indexing

```
39   18   43   00
0345 0346 0347 0348
```

Operand After Indexing

```
40   80   14   30
0355 0356 0357 0358
```

Index Register

```
00     10
Byte 0 Byte 1
```

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 ($DADD)*, except that the sign of the operand is changed before addition when the ($DADD) instruction is used. The sign of the difference is determined by the rules of algebra. The sign for any difference that equals 0 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
```

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
```

Index Register

```
04      00
Byte 0  Byte 1
```

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

Operand After Indexing

04D5

Index Register

00000000	00001011
Byte 0	Byte 1

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

```
140396
```

Operand Before Operation

```
00111000   01011101
0396       0397
```

Operand After Operation

```
00000000 00000111
0396     0397
```

Index Multiplier Register Before and After Operation

```
00000000 00000111
Byte 0   Byte 1
```

Example (Indexed)

Instruction

```
150396
```

Operand Before Operation (after indexing)

```
00000100 10011001
03A2     03A3
```

Operand After Indexing (before operation)

```
00111000 01011101
0396     0397
```

Operand After Operation

```
00000000 01000101
03A2     03A3
```

Index Register

```
00000000 00001100
Byte 0   Byte 1
```

Index Multiplier Register Before and After Operation

```
00000000 01000101
Byte 0   Byte 1
```

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

    080814

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

    090814

Operand Before Indexing

    00000001 10001001
    0814   0815

Operand After Indexing

    00000000 11011011
    0DB7   0DB8

Index Register

    00000101 10100011
    Byte 0   Byte 1

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
```

Operand Before Indexing

```
00000000  01111101
0738      0739
```

Operand After Indexing

```
00000000  11011111
07B1      07B2
```

Index Register

```
00000000  01111001
Byte 0    Byte 1
```

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

[Labe'] $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

0B01 8C

Operand Before Indexing

018C

Operand After Indexing

01FB

Index Register

00000000	01101111
Byte 0	Byte 1

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

```
0E06C4
```

Operand

```
00000000 00001101
06C4     06C5
```

Index Multiplier Register Before and After Operation

```
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

```
0F06C4
```

Operand Before Indexing

```
00000000 00001101
06C4     06C5
```

Operand After Indexing

```
00000000 00011000
06D0     06D1
```

Index Register

```
00000000 00001100
Byte 0   Byte 1
```

Index Multiplier Register Before and After Operation

```
00000000 00000011
Byte 0   Byte 1
```

Index Register Before Operation

```
00000000 01000101
```

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  000110
0DC2      0DC3
```

Index Multiplier Register Before and After Operation

```
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
```

Operand Before Indexing

```
00000000  000110
0DC2      0DC3
```

Operand After Indexing

```
00000000  000010
0DEC      0DED
```

Index Register

```
00000000  00101010
Byte 0    Byte 1
```

Index Multiplier Register Before and After Operation

```
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
```

Operand Before Operation

```
00001001 10011001
0A12     0A13
```

Operand After Operation

```
00000000 00110011
0A12     0A13
```

Index Register Before and After Operation

```
00000000 00110011
Byte 0   Byte 1
```

Example (Indexed)

Instruction

```
0D 0A 12
```

Operand Before Indexing

```
00001001 10011001
0A12     0A13
```

Operand After Indexing (before operation)

```
00110110 01100110
0A45     0A46
```

Operand After Operation

```
00000000 00110011
0A45     0A46
```

Index Register Before and After Operation

```
00000000 00110011
Byte 0   Byte 1
```

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 checks 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 checks 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 checks the contents of the binary register. If the binary register is 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, 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 (address 1), zero (address 2), or positive (address 3).

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.

Conversion Instructions

SINGLE-PRECISION TO BINARY ($RCNB)

Macroinstruction Format

[Label] $RCNB

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 01

Operation

This instruction converts the single-precision real number in the floating-point register to an integer value in the binary register.

DOUBLE-PRECISION TO BINARY ($DCNB)

Macroinstruction Format

[Label] $DCNB

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 02

Operation

This instruction converts the double-precision real number in the floating-point register to an integer value in the binary register.

BINARY TO SINGLE-PRECISION ($BCNR)

Macroinstruction Format

[Label] $BCNR

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 03

Operation

This instruction converts the integer number in the binary register to a single-precision real value in the floating-point register.

BINARY TO DOUBLE-PRECISION ($BCND)

Macroinstruction Format

[Label] $BCND

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 04

Operation

This instruction converts the integer number in the binary register to a double-precision value in the floating-point register.

SIGN CHANGE BINARY ($BSGN)

Macroinstruction Format

[Label] $BSGN

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 05

Operation

This instruction changes the sign of the number in the binary register. That is; a negative sign is changed to positive, and a positive sign is changed to negative. The integer value remains unchanged.

SIGN CHANGE FLOATING-POINT ($RSGN)

Macroinstruction Format

[Label] $RSGN

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 06

Operation

This instruction changes the sign of the number in the floating-point register. That is, a negative sign is changed to positive, and a positive sign is changed to negative. The characteristic and mantissa remain unchanged.

BINARY ABSOLUTE ($BABS)

Macroinstruction Format

[Label] $BABS

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 07

Operation

This instruction provides the absolute value of the number in the binary register. If the sign of the number is negative, it is changed to positive. If the sign of the number is positive, it remains positive. The integer value remains unchanged.

FLOATING-POINT ABSOLUTE ($RABS)

Macroinstruction Format

[Label] $RABS

Machine Instruction Format

Byte 1 Op Code	Bytes 2 and 3
66	00 08

Operation

This instruction provides the absolute value of the number in the floating-point register. If the sign of the number is negative, it is changed to positive. If the sign of the number is positive, it remains positive. The characteristic and mantissa remain unchanged.

Appendix A. 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    |        3380₁₀=D34₁₆
```

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 ————————————————↑
```

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

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

$16^3 = 4096$

$$4096 \overline{)2215.000000} = .540771$$

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

Appendix B. Summary of System/34 and System/32 Scientific Instruction Set

System/34 and System/32 scientific programs are executed under the control of an interpreter that resides in control storage. The object program language that is processed by the interpreter is called the scientific instruction set. The major component of the scientific instruction set is the 3-byte scientific instruction. 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 address. The effective address for a scientific instruction is computed by adding the contents of bytes 1 and 2 to the contents of XR (index register) if the index bit is 1. Scientific instruction addresses consistently refer to the leftmost byte of data entries in main storage.

The principal scientific instruction set registers are as follows:

1. *XR: Index Register*—contains 2-byte value used in indexing the effective address.

2. *XMR: Index Multiplier Register*—contains 2 bytes used for temporary storage in computing index values.

3. *BR: Binary Register*—contains 2 4-byte, twos-complement register used for integer arithmetic.

4. *FR: Floating-Point Register*—contains short or long precision floating-point hexadecimal value in System/360 format.

5. *Scientific IAR: Instruction Address Register*—contains 2 bytes that hold the address for the next scientific instruction to be executed.

6. *AR: Address Register*—contains addresses for certain scientific operands.

7. *CR: Condition Code Register*—contains 1 byte that holds the result of a compare operation.

When control is passed to the load module for execution, the first instruction in the program 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	*	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	Index register load
X'08'	XA	$XADD	Index register add
X'0A'	XLI	$XLI	Index register load immediate
X'0C'	XST	$XST	Index register store
X'0E	XM	$XMLT	Index register multiply
X'10'	XMA	$XMTA	Index register mulitply and add
X'12'	XMLI	$MLI	Index multiplier register load immediate
X'14'	XMST	$XMST	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

Hex Value	Scientific Instruction Mnemonic	Scientific Macroinstruction Mnemonic	Functional Description
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 divided
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

Hex Value	Scientific Instruction Mnemonic	Scientific Macroinstruction Mnemonic	Functional Description
X'44'	DL	$DLD	Floating-point register double-precision load
X'46'	ADR	$ALI or $ALD	Address register load
X'48'	INV	$INVK	Invoke system mode
X'4A'	DOBGN	*	DO loop initialization
X'4C'	DOEND	*	DO loop end
X'4E'	CALL	$CALL	Subprogram call
X'50'	IO	*	Input/output control
X'52'	DED	*	Data element descriptor
X'54'	DODED	*	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
X'66'	MFUNC (01)	$RCNB	Single precision to binary conversion

Hex Value	Scientific Instruction Mnemonic	Scientific Macroinstruction Mnemonic	Functional Description
X'66'	MFUNC (02)	DONB	Double precision to binary conversion
X'66'	MFUNC (03)	$BCNR	Binary to single precision floating-point conversion
X'66'	MFUNC (04)	$BCND	Binary to double precision conversion
X'66'	MFUNC (05)	$BSGN	Sign change binary conversion
X'66'	MFUNC (06)	$RSGN	Sign change floating-point conversion
X'66'	MFUNC (07)	$BABS	Binary absolute conversion
X'66'	MFUNC (08)	$RABS	Floating-point absolute conversion

*These scientific instructions (asterisk in the Scientific Macroinstruction column) do not have macroinstruction equivalents and cannot be used by the assembler programmer.

**The hex entry in parentheses is the select byte for the MFUNC mnemonic.

Appendix D. How to Specify the OLINK Procedure

The OLINK procedure resides in the system library (#LIBRARY). It can be used to call the overlay linkage editor to create a load module. Following is the format of the procedure and a chart showing what each parameter is used for in the procedure:

```
OLINK module name, [object library] , [load module name]
                   [#LIBRARY      ]   [module name     ]

                  ,[load module library] , [attribute1] , [attribute2]
                   [#LIBRARY           ]

                  ,[mrtmax value] ,[user subrlib1,user subrlib2]
                   [     0      ]  [#LIBRARY                   ]

                  ,[YES]
                   [NO ]
```

Note: If the required module name is not entered, a prompt screen will appear. Each parameter and its respective defaults appear on the prompt screen. Any or all parameters may be keyed.

Parameter	Required/ Optional	Type	Specify	Default
1	Required	Name of object module	Name of object module	None
2	Optional	Library name with object module	Library name to be searched for object module	#LIBRARY
3	Optional	Name of load module	Name to be put on load module	Object name
4	Optional	Library in which load module is to be placed	Library in which load module is to be placed	#LIBRARY
5 and 6	Optional	Attribute	Enter one of the following per parameter (maximum of two):	
COM (Common)			DED (Dedicated) NEP (Never-ending program) NEX (Not executable) NIQ (Noninquirable) NSW (Nonswapable) LSC (Load only from system console) SIS (Scientific mode) SRQ (Source required) USC (Utility control statements)	Null
7	Optional	MRTMAX	Number of terminals available to be allocated (0 to 255)	0
8 and 9	Optional	User subroutine library	Where to find user subroutine members (maximum of 2 parameters)	
10	Optional	Whether to place on input job queue	Whether to place on input job queue	No

Examples:

1. An object module , PROGA, resides in the user library called USERLIB. The user wants an executable load module from this. The load module name is LOADA, and it will be placed in the USERLIB.

   OLINK PROGA,USERLIB,LOADA,USERLIB

2. An object module, SAMPL, resides in the system library called #LIBRARY. The user wants an executable load module having the same name in the same library.

   Note: By specifying only the object name, the defaults for the other parameters determine that the system library is to be searched, the load module name is to be the same as the object module name, and the load module is to be placed in the system library.

   OLINK SAMPL

Index

$ALD 13
$ALI 13
$BABS 57
$BADD 16
$BAND 53
$BCMP 18
$BCND 56
$BCNR 56
$BDIV 19
$BIF 55
$BLD 21
$BMLT 23
$BNOT 53
$BOR 53
$BSGN 57
$BST 25
$BSUB 27
$CALL 8, 9
$CNTR 8, 10
$CRSM 8, 9
$CRTN 8, 9
$CSET 8, 9
$CSUB 8, 9
$DADD 32
$DCMP 34
$DCNB 56
$DDIV 35
$DLD 37
$DMLT 39
$DST 41
$DSUB 43
$GOTO 55
$HADD 15
$HCMP 17
$HDIV 18
$HLD 20
$HMLT 22
$HST 24
$HSUB 26
$INVK 8, 9
$LSET 54
$MLI 45
$MST 46
$RABS 57
$RADD 31
$RCMP 33
$RCNB 56
$RDIV 34
$RIF 55
$RLD 36
$RMLT 38
$RSGN 57
$RST 40
$RSUB 42
$XADD 47
$XLD 48
$XLI 49
$XMLT 50
$XMTA 51
$XST 52

absolute binary 57
absolute floating-point 57
add, binary register-2 bytes 15
add, binary register-4 bytes 16
add, floating-point register, double-precision 32
add, floating-point register, single-precision 31
add, index register 47
address register 5
address register instructions 13
address register load 13
address, effective 2
addressing
  base-displacement 2
  direct 2
  indexed 2
AND, binary register 53

base-displacement addressing 2
binary absolute 57
binary conversions
  to floating-point double-precision 56
  to floating-point single-precision 56
binary data format 2, 14
binary instruction format 15
binary number representation 14
binary register 4

normalization, floating-point numbers 30
NOT, binary register 53
notation
  excess 64 29
  twos complement 14
number normalization, floating-point 30
number representation
  binary 14
  floating-point 30

OLINK procedure 69
OR, binary register 53

procedure, OLINK 61

register add, binary-2 bytes 14
register add, binary-4 bytes 16
register add, floating-point, double-precision 32
register add, floating-point, single-precision 31
register add, index 47
register AND, binary 53
register compare, binary-2 bytes 17
register compare, binary-4 bytes 18
register compare, floating-point, double-precision 34
register compare, floating-point, single-precision 33
register divide, binary-2 bytes 18
register divide, binary-4 bytes 19
register divide, floating-point, double-precision 35
register divide, floating-point, single-precision 34
register IF, binary 55
register IF, floating-point 55
register instructions
  address 13
  binary 14
  floating-point 29
  index 47
  index multiplier 45
register load immediate, index 49
register load immediate, index multiplier 45
register load, binary-2 bytes 20
register load, binary-4 bytes 21
register load, floating-point, double-precision 37
register load, floating-point, single-precision 36
register load, index 48
register multiply and add, index 51
register multiply, binary-2 bytes 22
register multiply, binary-4 bytes 23
register multiply, floating-point, double-precision 39
register multiply, floating-point, single-precision 38
register multiply, index 50
register NOT, binary 53
register OR, binary 53
register store, binary-2 bytes 24
register store, binary-4 bytes 25
register store, floating-point, double-precision 41
register store, floating-point, single-precision 40
register store, index 52
register store, index multiplier 46
register subtract, binary-2 bytes 26
register subtract, binary-4 bytes 27
register subtract, floating-point, double-precision 43
register subtract, floating-point, single-precision 42
registers 4
  address 5
  binary 4
  condition code 5
  floating-point 4
  index 4
  index multiplier 45
registers, size 4, 5
representation
  binary numbers 14
  floating-point numbers 30
return to scientific mode 8, 9
return to System/34 mode 8, 9

scientific environment 7
scientific implementation 9
scientific macroinstruction statements 2
scientific mode 7
  return to 9
scientific subroutine
  execute 9
  exit 9
  start 9
sign change to binary 57
sign change to floating-point 57
single-precision floating-point register add 31
single-precision floating-point register compare 33
single-precision floating-point register divide 34
single-precision floating-point register load 36
single-precision floating-point register multiply 38
single-precision floating-point register store 40
single-precision floating-point register subtract 42
single-precision floating-point to binary conversion 57
size, register 4
start scientific subroutine 8, 9
statements, scientific macroinstructions 1
store, binary register-2 bytes 24
store, binary register-4 bytes 25
store, floating-point register, double-precision 41

store, floating-point register, single-precision 40
store, index multiplier register 46
store, index register 52
subroutine
  execute scientific 8, 9
  exit scientific 8, 9
  start scientific 8, 9
subroutine linkage 7
subtract, binary register-2 bytes 26
subtract, binary register-4 bytes 27
subtract, floating-point, double-precision 43
subtract, floating-point register, single-precision 42
summary of scientific instruction set 61
System/34 mode
  return to 8, 9

test condition 54

2-byte binary register add 15
2-byte binary register compare 17
2-byte binary register divide 18
2-byte binary register load 20
2-byte binary register multiply 22
2-byte binary register store 24
2-byte binary register subtract 26
2-byte binary register add 15
2-byte binary register compare 17
2-byte binary register divide 18
2-byte binary register load 20
2-byte binary register multiply 22
2-byte binary register store 24
2-byte binary register subtract 26
4-byte binary register add 16
4-byte binary register compare 18
4-byte binary register divide 19
4-byte binary register load 21
4-byte binary register multiply 23
4-byte binary register store 25
4-byte binary register subtract 27

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*

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

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

Name ______________________________

Address ______________________________

____

IBM System/34 and IBM System/32
Scientific Macroinstructions
Functions Reference Manual

SA21-9275-0

SA21-9275-0

Fold

Fold

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
4111 Northside Parkway N.W.
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)

Cut Along Line

IBM System/34 and IBM System/32 Scientific Macroinstructions Functions Reference Manual (File No. S34/S32-36) Printed in U.S.A. SA21-9275-0

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*

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

Name ______________________________

Address ______________________________

____

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

IBM System/34 and IBM System/32
Scientific Macroinstructions
Functions Reference Manual

SA21-9275-0

SA21-9275-0

Cut Along Line

IBM System/34 and IBM System/32 Scientific Macroinstructions Functions Reference Manual (File No. S34/S32-36) Printed in U.S.A. SA21-9275-0

Fold Fold

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
4111 Northside Parkway N.W.
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/34 and IBM System/32 Scientific Macroinstructions Functions Reference Manual (File No. S34/S32-36) Printed in U.S.A. SA21-9275-0

SA21-9275-0