SC21-9037-3
File No. S38-01

IBM System/38

IBM System/38
Internal Microprogramming Instructions, Formats, and Functions Reference Manual

SC21-9037-3
File No. S38-01

IBM System/38

IBM System/38
Internal Microprogramming
Instructions, Formats, and Functions
Reference Manual

Fou"th Edition (August 1985)

This major revision makes obsolete SC21-9037-2 and Technical Newsletter SN21-8302. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition. Additions were made to inc!ude the 5382 Model 9 System Unit.

Changes are periodically made to the information herein; these changes will be reported in technical newsletters or in new editions of this publication.

Use this publication only for the purpose stated in *About This Manual*.

Publications are not stocked at the address below. Requests for copies of IBM publications and for technical information abou* the system should be made to your IBM representative or the branch office serving your locality.

This publication could contain technical inaccuracies or typographical errors. Use the Reader's Comment Form at the back of this publication to make comments about this publication. If the form has been removed, address your comments to IBM Corporation, Information Development, Department 245, Rochester, Minnesota 55901. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

This page is intentionally left blank.

About This Manual

PURPOSE OF THIS MANUAL

This reference manual provides a detailed definition of the machine functions performed by the processor but should not be considered tutorial for the IMP (internal microprogramming) instruction set. This manual does not attempt to describe how the VMC (vertical microcode) routines prepare the information or how the HMC (horizontal microcode) attempts to use it.

This manual is to be used by support personnel for the maintenance of System/38.

ORGANIZATION OF THIS MANUAL

The information presented in this manual includes:

Chapter	Content
2	*Internal Microprogramming Structure*-the IMP data types, the instructions, and permanent storage assignments.
3	*Horizontal Microcode Support Functions*-the HMC procedures and the HMC built-in functions.
4	*The processor*-the processor, the processor states, the execution functions, the input/output and asynchronous events.
5	*Tasking*-the function of the IMP, the IMP objects the tasking function uses, the control of tasking, and the intertask communications and synchronization.
6	*Supervisor Linkage and Exception Handling*-the supervisor linkage concepts and the objects it uses, the supervisor linkage control, the exceptions, and the instruction length count and IAR (instruction address register) settings.
7	*Input/Output and Asynchronous Events*-the methods used to communicate with I/O devices and the sources of asynchronous events.
8	*Virtual Storage Addressing*-the storage addressing structure of the IMP.
9	*Machine Support Functions*-the additional services that are available to support IMP instruction processing.
10	*Instructions*-detailed descriptions of IMP instructions.

The glossary in Appendix C includes definitions developed by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). This material is reproduced from the *American National Dictionary for Information Processing*, copyright 1977 by the Computer Business Equipment Manufacturers Association, copies of which may be purchased from the American Standards Institute, 1430 Broadway, New York, New York 10018.

WHAT YOU SHOULD KNOW

The reader should understand computer programming and the concepts used in System/38 before attempting to use the information in this manual.

IF YOU NEED MORE INFORMATION

- *IBM System/38 Bibliography*, GH30-0233

  This publication describes technical publications in support of System/38 machine components, system programming, application programming, and other supplemental information (for example forms and program listings).

- *IBM System/38 Functional Concepts Manual*, GA21-9330

  This publication is designed to provide an overview of the System/38 concepts, a definition and description of structures and objects, and a description of specific System/38 functions.

- *IBM System/38 Functional Reference Manual*, GA21-9331 and GA21-9800

  This publication is designed to describe the System/38 instruction set and contains a detailed description of each instruction. This publication also contains the specifications for objects, events, exceptions, and describes specialized instructions for source/sink devices.

- *IBM System/38 Vertical Microcode Data Areas*, SY21-0892

  This publication is designed to aid service personnel responsible for supporting the IBM System/38 by providing descriptions of the vertical microcode data areas within the system.

- *IBM System/38 Vertical Microcode Logic Overviews and Component Descriptions Manual*, SY21-0889

  This publication is designed to aid service personnel to isolate a malfunction in the System/38 vertical microcode.

- *IBM System/38 Vertical Microcode Logic Listings*, SYB1-0890

  This publication is designed to aid service personnel to isolate a malfunction in the System/38 vertical microcode.

- *IBM System/38 Vertical Microcode Module Descriptions*, SYB1-0891

  This publication is designed to aid service personnel to isolate a malfunction in the System/38 vertical microcode.

- *IBM System/38 Processing Unit Models 3, 4, and 5 Theory—Maintenance*, SY31-0524 and *IBM System/38 Processing Unit Models 6, 7, and 8 Theory—Maintenance*, SY31-0649

  These publications are designed to give service personnel a brief description of some of the unique features of System/38.

- *IBM System/38 Channel Theory—Maintenance*, SY31-0619

  This publication is designed to provide maintenance and theory information that will be used by the service personnel to maintain the System/38 channel.

- *IBM System/38 System Control Adapter Theory—Maintenance*, SY31-0527

  This publication is designed to provide maintenance and theory information that will be used by the service personnel to install and maintain the IBM System/38.

- *IBM System/38 Service Guide*, SY31-0523

  This publication is designed to provide the information needed to use the System/38 maintenance library and service functions. The publication also shows the maintenance overview, the maintenance library organization, the operator/service panel switch settings, how to use the MAPs, and how to select either concurrent or dedicated service functions.

- *IBM System/38 Diagnostic Aids*, SY21-0584

  This publication provides information about the tools, documentation, and procedures needed to aid in problem resolution for programming problems occurring within the System/38 CPF and the VMC of the System/38.

DEFINITIONS OF NOTES

The headings *Notes* and *Programming Notes* are used where additional information is provided on various topics. Notes further explain or clarify text. Programming notes either explain instruction implementation or they suggest additional uses of instructions for support personnel.

TERMINOLOGY

Certain fields or bit combinations in IMP objects are undefined. Some of these may be used by the IMP programmer and some may not. In order to distinguish between them, the following terminology will be used throughout this manual:

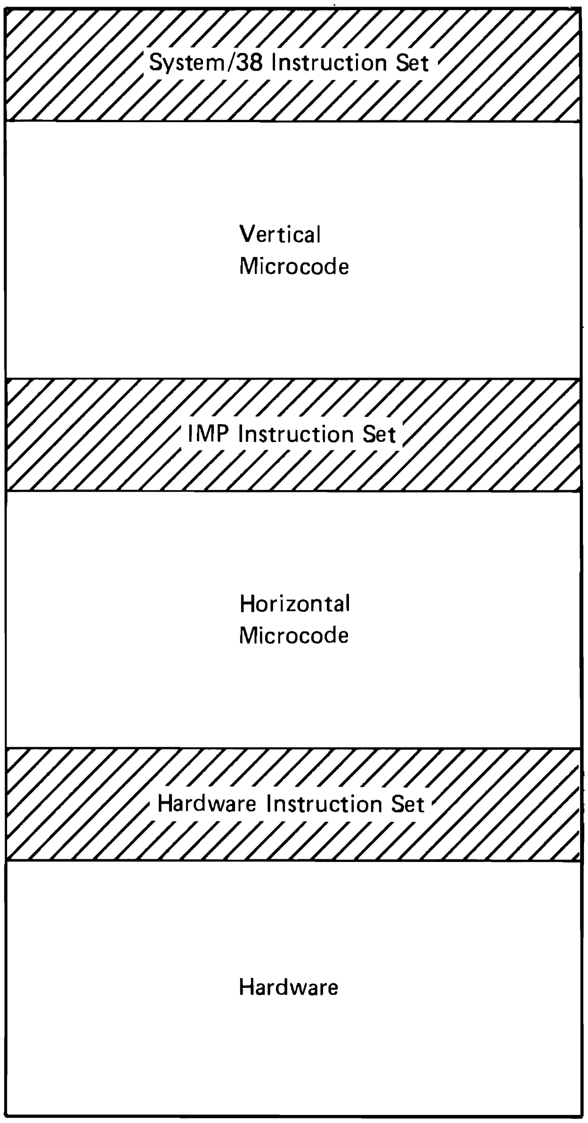
Term	**Meaning**
Not used	The field or bit combination is not interrogated or modified by the processor and may be used by the IMP programmer.
Reserved	The field or bit combination is interrogated or modified by the processor and may not be used by the IMP programmer.
Invalid	The bit combination is checked by the processor and a specification exception or a machine check occurs if an invalid combination is detected.

Chapter 1. Introduction

Machine Product

The IMP (internal microprogramming) instruction set is an internal communications link. The following figure shows the relationship of the instructions to other parts of the system.

System/38 Machine Support



The user/control program interface to the machine product is called the System/38 instruction set. The machine product that supports the System/38 instruction set is composed of hardware and microcode. Microcode is further subdivided into HMC (horizontal microcode) and VMC (vertical microcode). Definitions of these terms are:

- *Hardware*: A combination of silicon, copper, and frames providing a *hardwired* execution instruction set.
- *Microcode*: Instructions providing the basic machine control functions and supporting the System/38 instruction set.
- *HMC*: Microcode that exhibits a high degree of parallelism of execution, controls the detailed state of the hardware, and supports the IMP instruction set. HMC executes the hardware instruction set.
- *VMC*: Microcode that defines logical operations on data, is primarily sequential in execution, and supports the System/38 instruction set. VMC is executed in the IMP instruction set.

System Features

The IMP instruction set provides the fundamental processing capabilities of the machine. It includes decimal operations, with decimal shifting, providing instructions for commercial applications. Floating-point provides an instruction set for scientific computation.

Several of the instructions are executed in VMC. These instructions are indicated as SVL instructions in Appendix B. All other instructions shown in Chapter 10 are executed in HMC.

Storage Descriptions

Storage is composed of more than one technology. Except on performance, the effects of the physical differences between storage types is not observable to the application program.

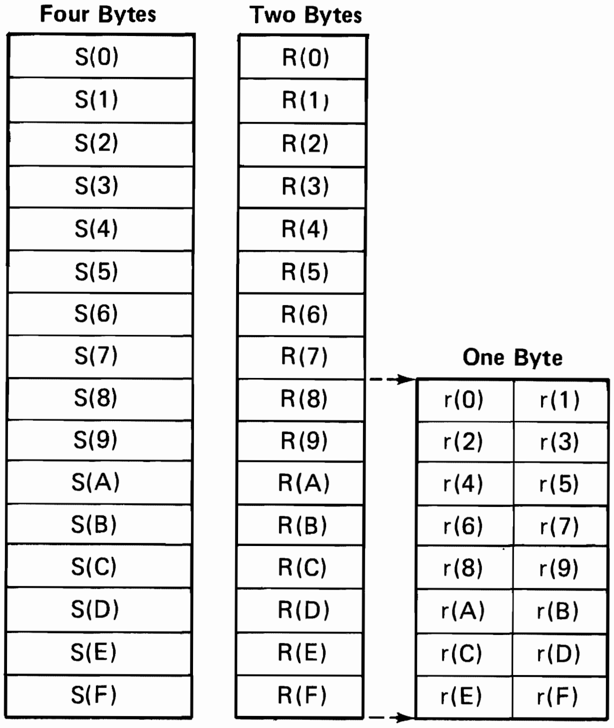
Fetching and storing of main storage data by the processor is temporarily prevented by I/O (input/output) data transfer operations. When concurrent I/O requests for access to a main storage location occur, access is normally granted in a priority sequence.

If the first reference to a storage location changes the contents of that location, any subsequent fetches from that location will obtain the new contents.

Register Descriptions

The hardware registers used with the processor can be used individually or combined to form larger registers. There are 16 SID (segment identifier) registers, 16 R (2-byte) registers, and 16 r (1-byte) registers as shown in the following figure. R registers hex 8-F are divided to form 16 single-byte registers, r(0)-r(F). The S and R registers are combined to form the B or base registers. The 16 base registers can contain addresses during IMP procedure execution. The address contained in base register 0 (B[0]) points to the start of the instruction stream, and all other instruction addressing and branching within a procedure is relative to B0. Base registers can be used to address areas in virtual storage of up to 64 K-bytes each.

Registers—Relative Sizes



Note: The number in parentheses indicates the number of the associated register (S, R, or r).

Data

The basic building block for all IMP formats is the 8-bit byte.

For the purpose of error detection and correction, one or more check bits are transmitted with each byte or group of bytes. The check bits are generated automatically by the hardware and cannot be directly controlled by the program. References in this manual to the sizes of data fields and registers exclude mention of the associated check bits.

The storage capacity is expressed in the number of bytes provided without regard to the storage width (number of bytes fetched or stored in one storage cycle).

The location of any field or group of bytes is specified by the address of its leftmost byte.

The length of a field can be implied by the operation to be performed. When the length is implied, there is no corresponding length field and the field is said to have a fixed length. Fixed-length data can be 1, 2, 4, 6 (in the Load instruction), 8, 12, 16, or 32 bytes long.

When the length of a field is not implied by the instruction operation code but is stated explicitly as a length field in the instruction or as part of the data accessed by the instruction, the information is said to have variable length.

When information is placed in storage, the contents of only those byte locations included in the designated field are replaced, even though the width of the physical path may be wider than the field being stored (fewer bytes may be stored than the processor is capable of storing in one storage cycle).

DATA TYPES

The computational instructions of the IMP operate on five data types: binary, address, character, decimal, and floating point.

Binary Data

An integer can be expressed as a signed or unsigned binary number.

In an unsigned binary number, all bits express the absolute value of the number. When two unsigned binary numbers are added, the shorter number is treated as if extended with high-order zeros. An unsigned binary number can appear as a byte or halfword in registers, or can be of variable length (1 to 256 bytes) in storage.

In signed binary numbers, the twos-complement representation of a negative number is considered the sum of the integer part of the field (taken as a positive number) and the maximum negative number. This is obtained by inverting each bit of the number and adding a one in the low order (units) position.

When an operand must be extended with high-order bits, the expansion is achieved by prefixing the field with bits equal to the sign bit. That is, positive numbers have leading zeros, and negative numbers have leading one bits.

Twos-complement notation does not include a negative zero. The maximum positive number is an all-one integer with a sign bit of zero. The maximum negative number (the negative number with the greatest absolute value) is an all-zero integer with a sign bit of one.

The processor cannot represent the complement of the maximum negative number. When an operation, such as a subtraction of the maximum negative number from zero, attempts to produce the complement of the maximum negative number, a binary overflow exception occurs. An overflow does not result, however, when the maximum negative number is complemented and the final result is within the representable range. An example of this is a subtraction of the maximum negative number from minus one. The product of two maximum negative numbers is represented as a double-length positive number.

In discussions of signed binary numbers in this document, the expression *16-bit signed integer* denotes a 15-bit integer with a sign bit (the maximum value is +32 767 or -32 768), and *32-bit signed integer* denotes a 31-bit integer with a sign bit (the maximum value is +2 147 483 647 or -2 147 483 648).

Address Data

Address data can have either a 6-byte or 2-byte format. The 6-byte format consists of a 4-byte SID (segment identifier) and a 2-byte offset. The SID identifies a 64 K-byte virtual address area called a segment. The offset identifies a 512-byte page within a segment and a single-byte location within the page. Base registers are used to store address data; operations on address data treat the data as unsigned binary values.

Character Data

Character data is stored in EBCDIC (extended binary coded decimal interchange code) with each character occupying an 8-bit byte.

Character strings are variable in length from 1 byte to a maximum of 65 536 bytes. Operations on character strings treat the data as unstructured logical quantities.

Decimal Data

Decimal data operands reside in storage and may be in either zoned or packed format. IMP instructions are provided for adding, subtracting, comparing, multiplying, dividing, editing and shifting decimal data in packed format only. Conversion instructions may be used to convert between packed format and signed binary format, between packed and zoned formats, and between packed or decimal and long or short floating-point formats.

Decimal Data Formats

In the zoned format, the rightmost 4 bits of a byte are called numeric (N) and normally comprise a code representing a decimal digit. The leftmost 4 bits of a byte are called zone (Z), except for the rightmost byte of the field, where these bits are a sign (S) code. In System/38 a zone character is represented as binary 1111 or hex F.

In the packed format, each byte contains two decimal digits (D), except for the rightmost byte, which contains a sign (S) to the right of the decimal digit. The digit and sign codes each comprise 4 bits.

Zoned Decimal Format

Z	N	Z	N	Z	N	Z	N	Z	N	S	N	
0	4	8	12	16	20	24	28	32	36	40	44	Bits

Packed Decimal Format

D	D	D	D	D	D	D	D	D	D	D	S	
0	4	8	12	16	20	24	28	32	36	40	44	Bits

Decimal operands occupy fields in storage that start on a byte boundary and can be variable in length (1 to 16 bytes). Decimal operands can overlap if the rightmost bytes coincide (the Move Packed Shifted instruction allows any overlap).

Decimal Number Representation

All decimal numbers are represented as right-aligned true integers with plus or minus signs.

The digits and signs are coded as shown in the following chart:

Binary Code	**Digit Symbol**	**Sign Symbol**
0000	0	Invalid[1]
0001	1	Invalid
0010	2	Invalid
0011	3	Invalid
0100	4	Invalid
0101	5	Invalid
0110	6	Invalid
0111	7	Invalid
1000	8	Invalid
1001	9	Invalid
1010	Invalid[1]	Plus (+)
1011	Invalid	Minus (-)
1100	Invalid	Plus (+)
1101	Invalid	Minus (-)[2]
1110	Invalid	Plus (+)
1111	Invalid	Plus (+)[2]

[1] Invalid means this code is not recognized as valid for this symbol.
[2] The preferred sign code.

Digit and Sign Codes

A data exception occurs with the detection of an invalid code. The operation is terminated when the digit code is invalid, or suppressed when the sign code is invalid (see Chapter 6).

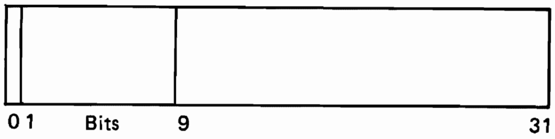
Although alternate encoding of the sign in an operand is accepted, the preferred sign codes are always generated for the results of the decimal arithmetic operations. Exceptions to this rule are permitted only during decimal conversion (CVZP and CVPZ instructions) and editing (EDPD instruction).

Floating-Point Data

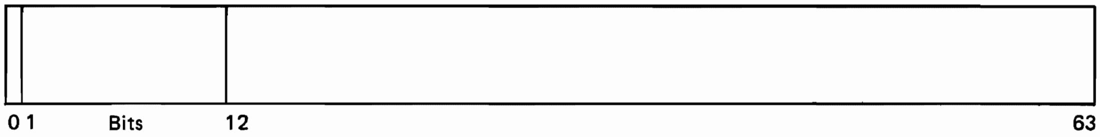
A floating-point number is a bit string characterized by a sign, a signed exponent, and a significand. Its value, if any, is the signed product of its significand and 2 raised to the power of its exponent. The exponent of a floating-point number normally signifies the power to which 2 is raised in determining the value of the represented number. The significand of a floating-point number consists of an implicit leading bit to the left of its binary point and a fraction field to the right of its binary point.

Floating-point data has a fixed length, 4-bytes long (short format) or 8-bytes long (long format). Both formats are designated as operands in storage and must be fullword aligned, or a specification exception occurs and the operation is suppressed. The formats are as follows:

Short Format



Long Format



Component	Short Format	Long Format
Sign Bit	Bit 0	Bit 0
Exponent	Bits 1-8	Bits 1-11
Fraction	Bits 9-31	Bits 12-63

Floating-Point Number Representation

The floating-point number values that can be represented by the short and long floating-point data formats include both real and symbolic numbers.

Real numbers can be represented in either normalized or denormalized format. In normalized format, the significand for the floating-point number is formed by assuming an implicit 1 bit to the left of the binary point and concatenating the fraction to the right of the binary point. As previously stated, the binary point in either format is assumed to be to the immediate left of the leftmost bit of the fraction; the fraction is expressed in binary digits (bits).

The significand is multiplied by a power of 2; the exponent indicates this power. The exponent field can contain a value that can range from 0 through 255 for the short format and 0 through 2047 for the long format. The minimum bias value 0 identifies plus or minus 0 and denormalized numbers, all of which are real numbers. The maximum exponent values (255 and 2047) identify symbolic numbers. The biased exponent when adjusted by the appropriate bias (-127 for the short format and -1023 for the long format) yields a signed (unbiased) exponent. This signed exponent specifies the power of 2 which is to be multiplied with the significand to produce the magnitude of the floating-point number. The sign of the floating-point number is either positive or negative, depending on whether the sign bit is 0 or 1 respectively.

In addition to real numbers, the symbolic entities of plus and minus infinity and a concept of not-a-number (NaN) can be represented.

Infinity is represented by the maximum exponent value (255 for short format and 2047 for long format) and a fraction of all 0 bits. Infinity is either positive or negative, depending on whether the sign bit is 0 or 1 respectively.

Not-a-number is represented by the maximum exponent value and a fraction that contains one or more 1 bits. There are two types of NaNs, masked and unmasked, with the high-order bit of the fraction indicating the type through a value of 1 or 0 respectively. The fraction component of a NaN can have any value other than all zeros. These values have no meaning, except that the fraction value of a leading 1 bit followed by all zeros is the value returned when a masked floating-point invalid operand occurs and neither operand is an unmasked NaN. Unmasked NaNs, when encountered in a floating-point operation, force the detection of the floating-point invalid operand condition. Masked NaNs, when encountered in a floating-point operation, are propagated into the result field, but do not force detection of the floating-point invalid operand condition. A potential usage of these NaN values is to set them into uninitialized floating-point fields. This allows the detection of a reference to a floating-point field that has not been set with a value by the time it is accessed.

The following information provides a summary of the values that can be represented by floating-point data. In the following formulas, S = the sign, E = the biased exponent or reserved value, and F = the fraction components of a floating-point field as previously described. Additionally, the ** characters denote exponentiation, and the ¬ character denotes a logical not.

The values that can be represented in the short format are:

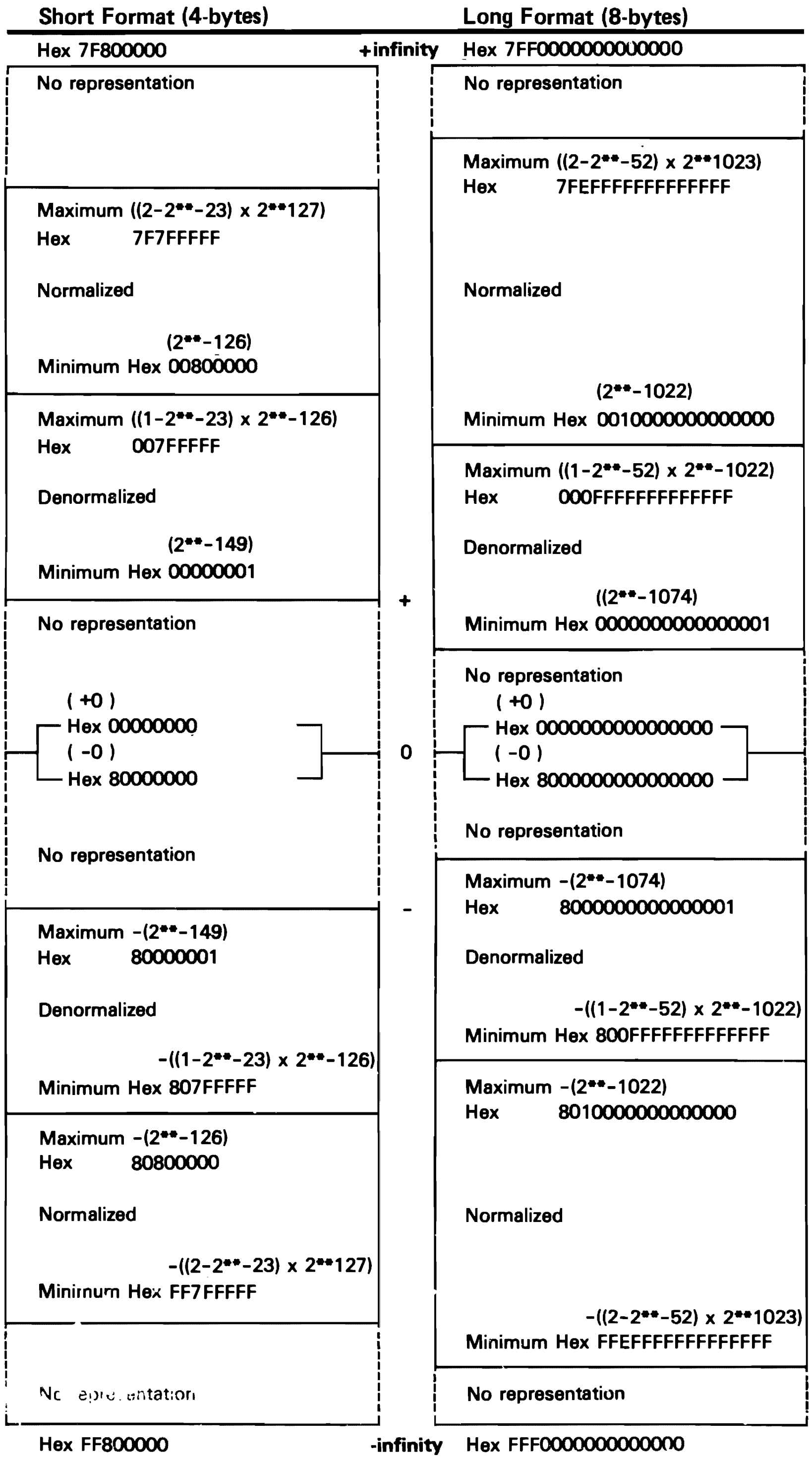
- Normalized number
  (For 0<E<255,
  value = (-1)**S x 2**(E-127) x 1.F)
- Denormalized number
  (For E=0 & F¬=0,
  value = (-1)**S x 2**(-126) x 0.F)
- Signed zero
  (For E=0 & F=0,
  value = (-1)**S x 0)
- Signed infinity
  (For E=255 & F=0,
  value = (-1)**S x infinity)
- Not-a-number (NaN)
  (For S=0 or 1, E=255, F¬=0 and with:
  - high-order fraction bit =1;
    value = masked NaN
  - high-order fraction bit = 0;
    value = unmasked NaN)

The values that can be represented in the long format are:

- Normalized number
  (For 0<E<2047,
  value = (-1)**S x 2**(E-1023) x 1.F)
- Denormalized number
  (For E=0 & F¬=0,
  value = (-1)**S x 2**(-1022) x 0.F)
- Signed zero
  (For E=0 & F=0,
  value = (-1)**S x 0)
- Signed infinity
  (For E=2047 & F=0,
  value = (-1)**S x infinity)
- Not-a-number (NaN)
  (For S=0 or 1, E=2047, F¬ = 0 and with:
  - high-order fraction bit = 1;
    result = masked NaN
  - high-order fraction bit = 0;
    result = unmasked NaN)

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

- In the short format:
  - Normalized
    2**-126 ≤ M ≤ (2-2**-23) x 2**127
  - Denormalized
    2**-149 ≤ M ≤ (1-2**-23) x 2**-126
- In the long format:
  - Normalized
    2**-1022 ≤ M ≤ (2-2**-52) x 2**1023
  - Denormalized
    2**-1074 ≤ M ≤ (1-2**-52) x 2**-1022



Short Format (4-bytes)

Hex 7F800000

No representation

Maximum ((2-2**-23) x 2**127)
Hex 7F7FFFFF

Normalized

(2**-126)
Minimum Hex 00800000

Maximum ((1-2**-23) x 2**-126)
Hex 007FFFFF

Denormalized

(2**-149)
Minimum Hex 00000001

No representation

(+0)
Hex 00000000
(-0)
Hex 80000000

No representation

Maximum -(2**-149)
Hex 80000001

Denormalized

-((1-2**-23) x 2**-126)
Minimum Hex 807FFFFF

Maximum -(2**-126)
Hex 80800000

Normalized

-((2-2**-23) x 2**127)
Minimum Hex FF7FFFFF

No representation

Hex FF800000

+infinity

+

0

-

-infinity

Long Format (8-bytes)

Hex 7FF0000000000000

No representation

Maximum ((2-2**-52) x 2**1023)
Hex 7FEFFFFFFFFFFFFF

Normalized

(2**-1022)
Minimum Hex 0010000000000000

Maximum ((1-2**-52) x 2**-1022)
Hex 000FFFFFFFFFFFFF

Denormalized

((2**-1074)
Minimum Hex 0000000000000001

No representation
(+0)
Hex 0000000000000000
(-0)
Hex 8000000000000000

No representation

Maximum -(2**-1074)
Hex 8000000000000001

Denormalized

-((1-2**-52) x 2**-1022)
Minimum Hex 800FFFFFFFFFFFFF

Maximum -(2**-1022)
Hex 8010000000000000

Normalized

-((2-2**-52) x 2**1023)
Minimum Hex FFEFFFFFFFFFFFFF

No representation

Hex FFF0000000000000

Hex 7FC00000	—	Masked NaN minimum	—	Hex 7FF8000000000000
Hex 7FFFFFFF	—	Masked NaN maximum	—	Hex 7FFFFFFFFFFFFFFF
Hex 7F80001.	—	Unmasked NaN minimum	—	Hex 7FF0000000000001
Hex 7FBFFFF.	—	Unmasked NaN maximum	—	Hex 7FF7FFFFFFFFFFFF

Note: Use of sign field bit value of 0 is arbitrary.

Normalization

Normalization is performed on intermediate results prior to assigning their value to the result field. If the number is nonzero, its significand bit becomes 1; the exponent is regarded as if its range is unlimited. This produces normalized floating-point data for which an implicit 1 bit is assumed to be to the immediate left of the binary point. If the significant is 0, the number becomes 0 with the sign being set as described under *Sign Bit* and *Signed Zero*. Normalizing a number does not change its sign.

If a normalized floating-point number has an exponent value that is outside the range supported for normalized numbers in the destination format, one of the following conditions is recognized:

- A floating-point overflow condition is recognized if the exponent is greater than the maximum (127 for short and 1023 for long).

- A floating-point underflow condition is recognized if the exponent is less than the minimum (-126 for short and -1022 for long) and either the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled.

Floating-point operations for which the floating-point underflow condition is detected and masked at the time of detection produce denormalized floating-point data in the result field. Denormalization is performed on the normalized intermediate result by shifting the significand right while incrementing the exponent until the exponent attains the formats assumed value for denormalized numbers (-126 for short and -1022 for long). The intermediate denormalized floating-point is then represented in the result field by setting the result exponent to the minimum value of 0 and the result signed fraction to the value from the significand of the intermediate denormalized floating-point number. Rounding is performed according to the current rounding mode on assignment of the significand value to the result fraction. This produces denormalized floating-point data for which an implicit 0 bit is assumed to be to the immediate left of the binary point and for which an unbiased exponent value of -126 for short and -1022 for long is to be assumed. The exponent value of 0, which correlates with unbiased exponent values of -126 for short and -1022 for long, serves as an identifier for denormalized floating-point data and is not used to form the true signed exponent of the floating-point number represented. The underflow exception is signaled only if the result is not exact.

Floating-point fields can only contain real numbers in the normalized or denormalized formats. The concept of an unnormalized number (one which would allow for a variable exponent in conjunction with one or more leading 0 bits prior to the first significant 1 bit) does not exist and cannot be represented.

Rounding

All floating-point operations are performed as if to infinite precision and then, if necessary, rounded to fit in the destination's format. Four mutually exclusive rounding modes are supported: round to nearest, round toward zero, round toward positive infinity, and round toward negative infinity. The rounding mode bits are kept in the TDE (task dispatching element). If y is the infinitely precise number that is to be rounded, x is the number with the largest representable significand less than y, and z is the number with the smallest representable significand greater than y (where the exponents for x and z may be out of range); then, if y is not representable in the destination format (needs rounding), the rounding modes change y as follows:

- Round to nearest: y is changed to the closer of x or z. If they are equally close, the even one (the one whose least significant bit is a 0) is chosen.

- Round toward zero: y is changed to the smaller (in magnitude) of x or z.

- Round toward positive infinity: y is changed to z, unless z is negative and its exponent overflows the destination's format. In this case, y becomes the format's largest (in magnitude) negative real number.

- Round toward negative infinity: y is changed to x, unless x is positive and its exponent overflows the destination's format. In this case, y becomes the format's largest positive real number.

Infinity

In infinity arithmetic, infinities compare equal regardless of sign, and compare unordered with anything else. Arithmetic operations on infinity are always exact.

Sign Bit and Signed 0

The sign of a product or a quotient is the exclusive OR of the signs of the operands. The sign of a sum or a difference differs from, at most, one of the signs of the operands following the normal rules of algebra. These rules apply even when operands or results are 0 or infinite. The only exception is when the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly 0, the sign of that sum (or difference) depends on the current rounding mode of the process. For round toward negative infinity, the sign is minus; for all other rounding modes, the sign is plus.

Exceptions

The following floating-point exception conditions can be detected during floating-point operations:

- Floating-point inexact result

- Floating-point invalid operand

- Floating-point overflow

- Floating-point underflow

- Floating-point zero divide

- Invalid floating-point conversion

Associated with these exceptions (except the invalid floating-point conversion) is a set of mask and occurrence bits in the TDE.

The mask bit controls the signaling of the exception. If the mask bit is 0, the exception is not signaled; if the mask bit is 1, the exception is signaled. The mask bit is only interrogated for its value. It must be set manually.

The occurrence bit records the detection of the exception condition whether or not the exception is masked at the time of detection (1 = occurred; 0 = has not occurred). The occurrence bit must be manually reset.

The definition of these exception conditions and what the result of the operation will be when they are detected is contained under *Exceptions* in Chapter 6.

Internal Microprogramming Objects

IMP objects are separately addressable units (or collections of data) that have associated attributes as well as operational characteristics based on these attributes. The IMP objects support the tasking and I/O structures of the IMP. These objects are used by certain IMP instructions (such as Enqueue Message) and must begin on fullword main storage boundaries without crossing page boundaries (see *Data Alignment*, in this chapter). The IMP objects are:

- Task dispatching queue (TDQ)
- Task dispatching element (TDE)
- Send/receive queue (SRQ)
- Send/receive message (SRM)
- Send/receive counter (SRC)

A characteristic common to all IMP objects is the use of the descriptor. The descriptor provides type and control information about an IMP object. It is used during execution of any IMP instruction that operates on an IMP object, to ensure that the operand is valid for the operation and to provide additional information about an IMP object.

The descriptor is 2 bytes long. Byte 0 identifies the type of IMP object being described and contains additional information, including status information (see the following figure).

Byte 1 of the descriptor is used by the processor to monitor accesses to the SRM, SRQ, and TDE IMP objects. If the processor detects byte 1 ≠ hex 00 while executing an operation that accesses one or more of these objects, a descriptor access exception occurs.

Byte 1 of the TDQ descriptor is ignored.

When an IMP object is initially created, bytes 0 and 1 must be appropriately initialized or unpredictable results will be encountered. (For more information see Chapter 5, *Tasking*.)

Descriptor Type–Byte 0, Bit Assignment

Bits[1]				
0	**1**	**2**	**Mnemonic**	**Name**
0	0	0	TDQ	Task Dispatching Queue
0	0	1	TDE	Task Dispatching Element
0	1	0	SRQ	Send/Receive Queue
0	1	1	SRM	Send/Receive Message
1	0	0	SRC	Send/Receive Counter
1	0	1	–	Not valid
1	1	0	–	Not valid
1	1	1	–	Not valid

[1] Bits 3 through 7 are object dependent and are described in Chapter 5.

DATA ALIGNMENT

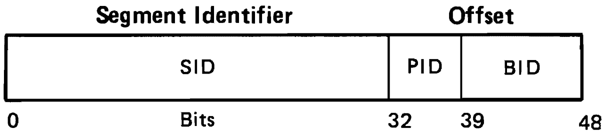
Data alignment must meet the following requirements:

- Instructions must begin on halfword boundaries.
- Halfword storage operands must begin on halfword boundaries.
- Fullword storage operands must begin on fullword boundaries.
- Full addresses (6 bytes) in storage must begin on halfword boundaries.
- Intermediate address fields for certain instructions (BALL, CLCL, and MVCL) must begin on fullword boundaries and for CLCL and MVCL cannot cross page boundaries.
- Floating-point data (long or short format) must begin on fullword boundaries.
- IMP objects must begin on fullword boundaries and cannot cross page boundaries (see the note under *Send/Receive Message* in Chapter 5 for an exception).
- Doubleword alignment is required for certain I/O objects (see *I/O Addressing Restrictions* in Chapter 7), some addresses in the control address table (see *Assigned Virtual Storage Locations* in this chapter), and the repetitive interval timer doubleword (see *Interval Timers* in Chapter 9).
- Space pointers must be quadword aligned.
- The hash table and primary directory must be aligned on an SID (segment identifier) boundary.

If the alignment requirements are not met, a specification exception occurs. If the system is attempting to recover from some malfunction and the system encounters an object not in proper alignment, a machine check occurs.

ADDRESSING

All addresses used by the processor in executing instructions or fetching data are virtual addresses. The complete virtual address of any byte of storage is 48 bits containing an SID (segment identifier) and an offset. The offset contains a PID (page identifier) and a BID (byte identifier).



The SID uniquely identifies a 64 K-byte virtual address area called a segment. The entire virtual storage of the VMC can be considered a collection of nonoverlapping segments.

The offset identifies a 512-byte page within a segment and a single-byte location within the page. Therefore, it provides for relative addressing of up to 65 535 bytes beyond the location designated by the SID.

Storage operand addressing is achieved by adding a displacement to a base register identified by the instruction. The displacement is a 12-bit field also found in the instruction.

Address translation uses the VAT (virtual address translator) facilities described in Chapter 8. These facilities include:

- *Hash Table*–a list of entries used to index the primary directory.
- *Primary Directory*–a table of the virtual addresses of a page. The table also provides status information about the page.
- *Lookaside Buffer*–a high-speed buffer storage that contains some of the information specified in the PD (primary directory). The translation process is shortened if the virtual address referred to is currently listed in the LB (lookaside buffer).

The following virtual addresses are called virtual = real addresses and do not use the VAT facilities. Virtual = real addresses are invalid if they exceed the amount of real storage configured for the processor.

Virtual = Real Address

Model	**Segment Identifier (SID) Values**	**Segments of Real Storage Addressed**
3, 4, and 5	0000 0100 through 0000 011F	First 32
6, 7, and 8	0000 0100 through 0000 01FF	First 256

Eight system control instructions are used to verify a virtual address and maintain the VAT facilities. These instructions are HVVA, IPDE, RRCRR, LPDEA, LHTEA, LPDEAR, EPDE, and RPDE.

Virtual address overflow protection is only on segment boundaries. When an offset attempts to overflow into the next SID, either the SID is incremented by 1 or an effective address overflow exception occurs. Which of these two events occurs is determined by the instruction involved, the model, the level of horizontal microcode, and the hardware. If a carry out of bit 24 occurs when adding 1 to the high-order 32 bits of a storage address, an effective address overflow occurs.

Instructions

Each instruction consists of two major parts: an op (operation) code and one or more operands.

- The operation code specifies the operation to be performed.
- The operands designate the data or address of data for the operation.

In addition, certain instructions may contain operand lengths, masks, or other control information needed to perform the specified operation.

OPERATION CODES

The operation code for an IMP instruction consists of an 8-bit code that is unique to either one instruction or to a set of instructions that use a unique operation code extender. The operation code occupies the first byte of the instruction. Appendix C shows the operation code assignments.

Operation Code	Instruction Length	Format
000x xxxx	2 bytes	RR
001x xxxx	2 bytes	RR
010x xxxx	4 bytes	RI, RS, or SI
011x xxxx	4 bytes	RI, RS, or SI
100x xxxx	4 bytes	RI, RS, or SI
101x xxxx	6 bytes	SI or SS
110x xxxx	6 bytes	SI or SS
111x xxxx	6 bytes	SI or SS

OPERANDS

Operands can be grouped in three classes and can be either explicitly or implicitly designated. The classes are:

- S (storage operands)–located in real storage
- R (register operands)–located in registers (internal storage)
- I (immediate operands)–located in the instruction itself

The length of an operand in storage can either be implied by the operation code, be specified by a bit mask, be explicitly provided by a register, or be specified by a 4-, 8-, or 16-bit L (length) field contained in the instruction or operand.

For explicitly stated variable length operands, the length code in the L field specifies the number of additional bytes to the right of the byte designated by the storage operand address. Therefore, the length in bytes is one more than the value of the L field.

The addresses of operands in storage are specified by means of a format that uses the contents of a B (base) register as part of the address. This makes it possible to:

- Specify a complete address by using an abbreviated notation.
- Perform address manipulation using instructions that use base registers for operands.
- Modify addresses by program means without alteration of the instruction stream.
- Operate independently of the locations of data areas by directly using addresses received from other programs.

The address used to refer to storage is contained in a register designated by the B field in the instruction, or is calculated from a base address and displacement designated by the B and D (displacement) fields in the instruction.

Register operands are located in registers identified in a 4-bit field in the instruction.

Immediate operands are contained within the instruction in a half-byte, byte, or halfword I (immediate) field.

To describe the execution of instructions, storage operands are designated as first and second (and in some cases, third) operands.

In general, two operands participate in the execution of an instruction. The result replaces the first operand. Except for storing the final result, the contents of all registers and storage locations participating in the addressing or execution of an operation for most instructions remain unchanged. A few instructions (such as TRT) also modify operands other than the final result.

Operand referencing is summarized in Figure 2-1. This figure shows the use of storage, immediate, and register operands.

- A Operation Code
- B Number of a halfword register (hex value), R(0)–R(F).
- C Number of a 1-byte register (hex value), r(0)–r(F); or a 4-bit operation code extension field, E
- D Immediate operand
- E Number of a base register (hex value), B(0)–B(F) B(n) = S(n) concatenated to R(n), where n is a value of hex 0 to F
- F A 12-bit displacement added to the base register
- G Storage operand addressed by B + D
- H Length of the storage operand in bytes, minus one

Note: The format used here does not represent an actual instruction. It does, however, illustrate the use of actual fields.

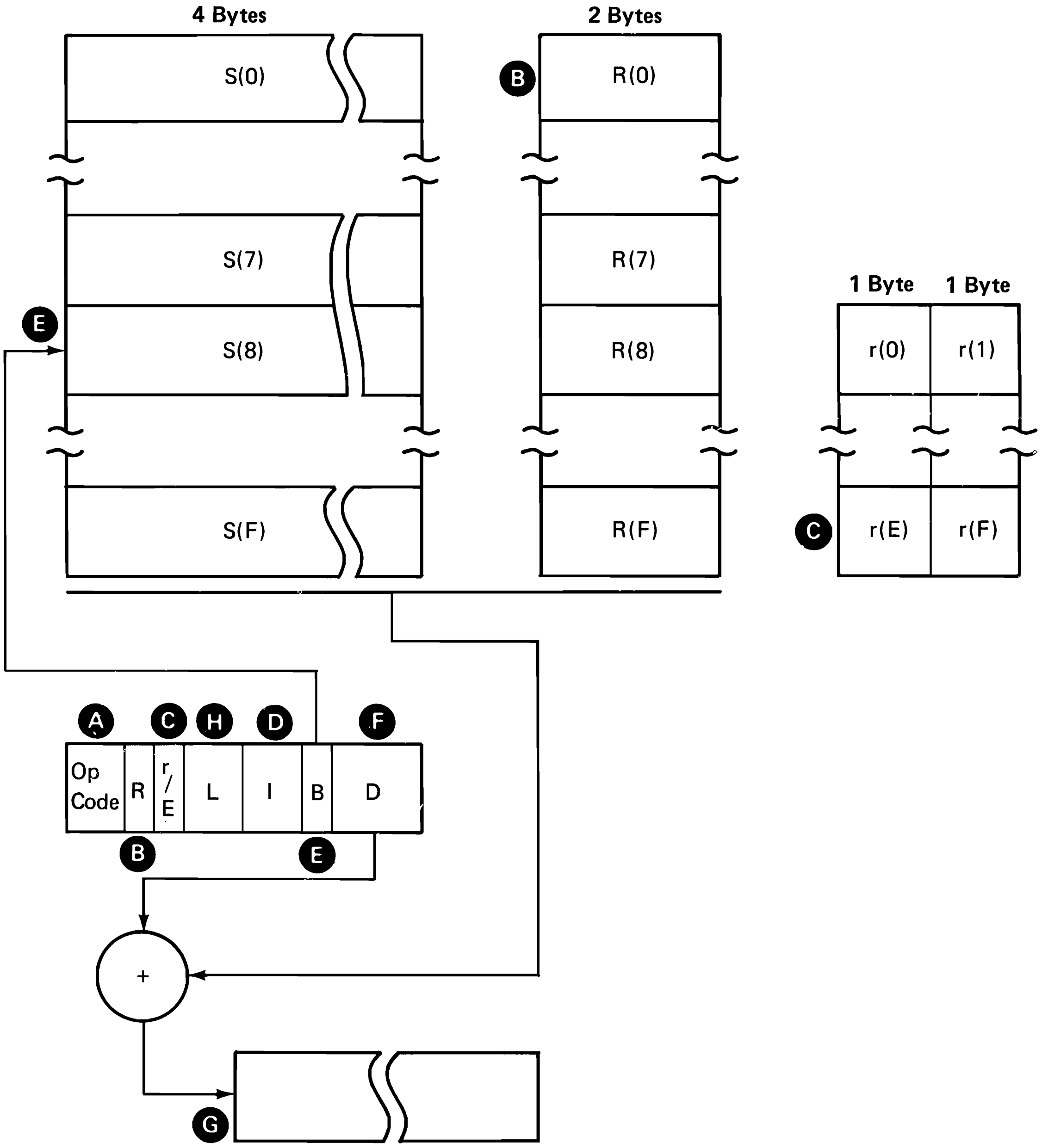


Figure 2-1. IMP Operand Reference

FORMATS AND EXAMPLES

An instruction is 1, 2, or 3 halfwords in length. Each instruction must be aligned on a halfword storage boundary and cannot cross a segment boundary. The basic instruction formats are shown in the following figure. The format of an instruction is dictated by the type of operation to be performed. In the figure, the bytes in each format are labeled with letters that indicate the use for each byte. The use of the bits within a given format can vary from instruction to instruction.

Basic IMP Formats

All IMP instructions fall within one of the following categories. Within each category, some instructions differ slightly from the basic format shown:

RR (register to register) — 2 Bytes

Operation Code	R_1	R_2/E

RI (register and immediate) — 4 Bytes

Operation Code	R_1	E	I_2

RS (register to storage) — 4 Bytes

Operation Code	R_1	E	B_2	D_2

SI (storage and immediate) — 4 Bytes

Operation Code	← I_2 → |←E→|	B_1	D_1

SI (storage and immediate) — 6 Bytes

Operation Code	← L → |←E→|	B_1	D_1	I_2

SS (storage to storage) — 6 Bytes

Operation Code	L_1	L_2/E	B_1	D_1	B_2	D_2

Legend[2]	
B	Base register
D	Displacement
E	Operation code extension field
I	Immediate
J	Jump displacement[1]
L	Length
M	Mask[1]
R	Halfword register
r	One-byte register[1]

[1] Symbols not shown in the examples above are used in the formats in Chapter 10.

[2] Subscript numbers that appear with these letters designate the operand number.

Note: A field left blank in the instruction format diagram may contain random values that are not important to the execution of the instruction; the same field is represented in the instruction example with the placeholder 0.

ADDRESS GENERATION

The storage address can be contained in a register designated by the B (base register) field in the instruction or calculated from a B and a D (displacement) field in the instruction.

The base address is a 48-bit number contained in a base register specified by the 4-bit B field of the instruction. A base address can be used as a means of independently addressing each program and data area. In array-type calculations, it can specify the location of an array. In record processing, it can identify the record.

In forming the storage address, the 16-bit offset (page and byte identifiers) portion of the base register and the 12-bit displacement field of the instruction are added as unsigned binary integers. The sum is always 16 bits long and is logically appended on the right to the high-order 32 bits of the base address. When an overflow occurs, either the high-order 32 bits of the storage address are incremented by one or an effective address overflow exception occurs. Which of these events occurs depends upon the particular instruction involved, the model and the engineering level of the horizontal microcode, and the engineering level of the hardware. If, in adding 1 to the high-order 32 bits of the storage address, a carry occurs from bit 24, an effective address overflow exception occurs.

A zero on the R (2-byte register), B (base register), or D (displacement) fields has no special significance except to denote the use of register zero or a zero displacement.

An instruction can designate the same base register both for address computation and location of an operand. Address computation is completed prior to the execution of the operation.

Unless otherwise indicated in the individual instruction definition, the computed operand address designates an operand in storage. When a storage operand is designated, the address points to the leftmost byte of the operand.

To find the effective address of a storage operand, first use the B field of the instruction to locate the base register; then add the contents of the base register and the contents of the D field of the instruction (see Figure 2-1) as follows:

Effective Address = Contents of Base Register + 12-bit displacement

EXECUTION

The IAR (instruction address register) contains a 2-byte offset into the segment identified by the SID (segment identifier) contained in register S(0). In program execution, the next instruction is fetched from the location designated by the IAR. The instruction address is then increased by the number of bytes in the instruction in order to address the next instruction in sequence. The instruction is then executed, and the same steps are repeated using the new value of the instruction address.

The normal sequential execution of instructions can be changed by:

- The use of branching instructions to perform subroutine linkage, decision making, and loop control.

- Conditions arising during program execution that cause linkage to an exception-handling routine.

- Conditions arising external to the currently executing program. Such conditions can cause interruption of processing, the storing of information describing the current program, and the invocation of another program that is part of the task whose condition caused the interruption.

Conceptually, the processor processes one instruction at a time, executes instructions sequentially, executes the instruction specified by the branch address following the successful execution of the branch, and allows interrupts to take place between the execution of instructions. Physical storage width and overlap of instruction execution with storage accessing may cause actual processing to differ from this concept. Each operation is performed sequentially with the next instruction being prefetched before the current operation is completed. Modification of succeeding instructions while using prefetch will produce unpredictable results.

It can be assumed that the execution of each instruction occurs as an indivisible event. However, in actual operation, the execution of an instruction can consist of a series of discrete steps. Depending on the instruction, operands can be fetched and stored in a piecemeal fashion, and some delay can occur between fetching and storing a result.

Programming Note: Because of a hardware restriction the last fullword of a segment on Models 3, 4, and 5 should not contain an instruction.

BRANCHING

A branch instruction (ALHBL, BC, BCN, BCNX, BCT, BU, TMBIBO, and TMBIBZ) is used for branching within the instruction stream that contains the branch instruction. The halfword displacement in the instruction (or pointed to by the instruction) is added to the contents of register R(0), and the result replaces the IAR (instruction address register).

The address in registers S(0) and R(0) must always point to the start of the instruction stream because all branching is done relative to this address.

The Branch Internal (BI) instruction is used for branching within the current segment group.

A jump instruction (JBN, JBF, and JC) works relative to the IAR. A 1-byte displacement is added to the 2-byte IAR to form the address of the next instruction.

A linkage instruction (BAL, BR, BALL, BRL, and CALLI) provides a mechanism to do a branch and link and a return. BAL (Branch and Link) and BR (Branch Register) instructions provide linkage to instructions in the same segment. BALL (Branch and Link Long) and BRL (Branch Register Long) provide linkage to instructions in a different segment.

Programming Note: The extended mnemonics used by the IMP instruction assembler for the BC, BU, and JC instructions are listed with their respective instruction descriptions.

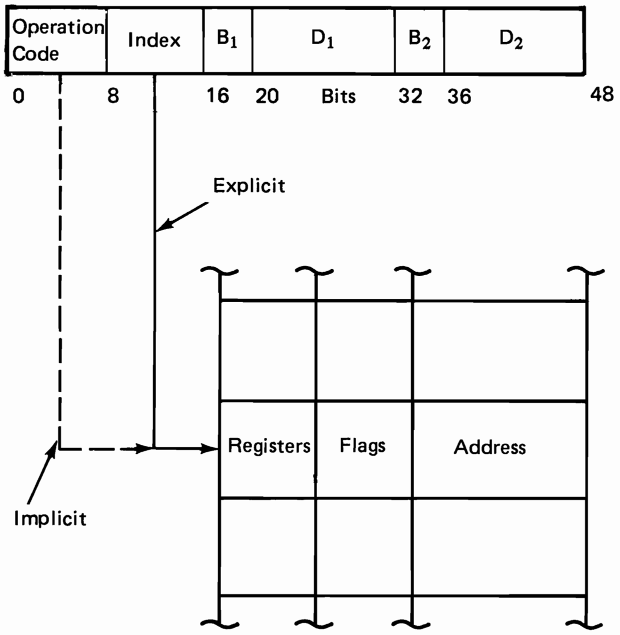
CONDITION CODES

Facilities for decision making are provided by the branch instructions. A 4-bit condition code reflects the results of most of the arithmetic, logical, and other manipulation and control instructions. Each of these operations can set (and reset) bits of the condition code and the branching instructions can specify (by masking) any selection of the bits as the criterion for branching. (See Chapter 10, *Instruction Descriptions* for the specific condition code settings.)

SUPERVISOR LINKAGE

The normal sequential execution of instructions can be changed by conditions arising during program execution. The IMP SVL (supervisor linkage) provides a trapping mechanism to handle these interruptions.

The SVL instructions have the following format:



For explicit SVLs, the second byte of the SVL instruction is used as an index into a main storage area called an SVL table. For implicit SVLs, the operation code acts as an index into the SVL table.

Each entry in the SVL table contains the number of registers to be stored, the address of the procedure to which control is passed, and other descriptive and control information.

Whenever the processor passes control via an SVL it automatically saves certain designated machine facilities such as the IAR (instruction address register), exception code, and condition code values. In addition, it optionally saves base registers. These facilities are saved in a special list element known as the CRE (call/return element).

See Chapter 6 for a description of the SVL facilities.

PROGRAM EXCEPTIONS AND INSTRUCTION LENGTH COUNT SETTINGS

Exceptions that result from the execution of instructions are called program exceptions. The SVL (supervisor linkage) mechanism is used to indicate exceptions. The first entry of the SVL table is the implicit index value associated with program exceptions.

For a more detailed explanation of exceptions and exception codes, see *Call/Return Element* and *Exceptions* in Chapter 6.

Concurrent Exceptions and Causes

Exception	Causes	Instruction Is
Soft address compare	Main store address compare when in address compare mode.	Completed (Note 1)
Task interval timer	Task interval timer expired during a timed task.	Nullified
Monitored ACQ (available call/return element queue)	An exception SVL detected a monitored ACQ (header byte 1 ≠ hex 00) during an implicit CRE receive.	Note 2
Monitored CRE (call/return element)	A monitored CRE (byte 1 ≠ hex 00) was detected, due to an implicit receive by an exception SVL.	Note 2
Monitored TDE (task dispatching element)	No available CREs exist for an implicit receive by an exception SVL, and the current TDE is monitored (byte 1 ≠ hex 00).	Note 2

Note 1: A soft address compare exception during an instruction stream fetch nullifies the instruction.

Note 2: The instruction termination state is determined by the concurrent program exception condition.

Program Exceptions and Causes

Hex Code	Exceptions	Causes	Instruction Is
00	No Exception		
02	Invalid Descriptor	Invalid field encountered during operation on IMP object.	Terminated
04	Busy	1. SRQ (send/receive queue) busy. 2. Hold/Free Chain busy.	Nullified Nullified
06	Reserved		
08	Allocate Page Frame	OU task requests page frame to be allocated and cleared in main storage.	Nullified
OA	Monitored Descriptor SRQ	SRQ (send/receive queue) access attempted when its byte 1 is nonzero.	Suspended
OC	Monitored Descriptor SRM	SRM (send/receive message) access attempted when its byte 1 is nonzero.	Suspended
OE	Monitored Descriptor TDE	TDE (task dispatching element) access attempted when its byte 1 is nonzero.	Suspended
10	Send/Receive Counter Overflow	A carry from the high-order position of the count field occurred during a send operation.	Terminated
12	Address Translation	Unable to translate a virtual to a real address by using VAT. For GHRF, GHR, FHRF, and FHR instructions, the instruction is completed and condition code 3 is set if exception occurs on hold record chain.	Nullified
14	Programming Event	An instruction is executed in a defined address range. 1. If not masked (bit 8 of TDE exception mask field is set) 2. If masked (bit 8 of TDE exception mask field is reset)	 Nullified Completed
16	Execute	Subject of EX instruction is another EX instruction	Suppressed
18	Specification (see note)	1. Improper alignment 2. Other conditions (see *Specification Exception* in Chapter 6).	Suppressed Suppressed
1A	Addressing	1. Invalid virtual = real instruction address. 2. Invalid virtual = real operand address.	Suppressed Terminated
1C	Effective Address Overflow	1. Offset overflow during effective address calculation. 2. Storage operand crossed segment boundary.	Suppressed Suppressed
1E	Data	1. Invalid decimal sign code. 2. Invalid decimal digit code. 3. Insufficient left zeros in multiplicand (MP).	Suppressed Terminated Terminated

Note: All instructions are tested for this exception.

Program Exceptions and Causes (continued)

Hex Code	Exceptions	Causes	Instruction Is
20	Binary Overflow	1. Carry from sign bit and carry from high-order numeric bit disagree. 2. Result exceeds 31 bits (CVPB). 3. Significant bits are lost (SLA).	Completed Completed Completed
22	Binary Divide	Quotient exceeds the size of the result field or an attempt to divide by zero.	Suppressed
24	Decimal Overflow	Destination field is too small for the result.	Completed
26	Decimal Zero Divide	An attempt to divide by zero.	Suppressed
28	Floating-Point Overflow	Resultant exponent is too large.	Completed
2A	Floating-Point Underflow	Resultant exponent is too small.	Completed
2C	Floating-Point Inexact Result	Rounded result is not exact.	Completed
2E	Floating-Point Zero Divide	An attempt to divide by a number with a zero fraction.	Suppressed if not masked; Completed if masked
30	Operation (see note)	Invalid operation code	Suppressed
32	Stack	1. Stack entry to be removed during unstack has flag bit 15 (first entry in segment) set. 2. Stack operation adds entry that extends beyond stack limit value.	Suppressed Suppressed
34	Verify	A verify exception occurs when an LVT, AHSPOI, AHSPO, or AFSPO instruction detects an invalid operand.	Suppressed
36	Chain Conflict	1. Conflict on an object hold operation. 2. Object free operation attempted to free a monitored hold.	Nullified Nullified
38	End-of-Chain	1. Empty chain on free operation. 2. End of available chain on hold operation. 3. No matching hold on free operation.	Nullified Nullified Nullified
3A	Edit Digit Count	1. End-of-source field was reached and there are more control characters corresponding to digits in edit-mask field than in source field. 2. End-of-edit-mask field was reached and there are more digit positions in the source field.	Terminated Terminated

Note: All instructions are tested for this exception.

Program Exceptions and Causes (continued)

Hex Code	Exceptions	Causes	Instruction Is
3C	Length Conformance	1. More character positions in result than in edit-mask field (EDPD). 2. More character positions in edit-mask field than in result field (EDPD). 3. Incorrect number of hex B2's following a hex B1 (floating string) field in the edit mask (EDPD). 4. The converted form of the source record is larger than the result record length (CVTMC).	Terminated Terminated Terminated Terminated
3E	Edit Mask Syntax	1. Invalid control characters in edit-mask field. 2. End-of-string character field termination missing.	Terminated Terminated
40	Invalid Segment Group Address	1. Leftmost 3 bytes of virtual address are invalid. 2. Address below lower boundary address. 3. Overflow generated in calculation of 3-byte address.	Suppressed Suppressed Suppressed
42	Floating-Point Invalid Operand	An operand or operation is invalid.	Suppressed if not masked; Completed if masked
44	Reserved		
46	Second Chain Search	A Grand Hold or Free Hold instruction has determined that a secondary chain must be searched.	Nullified
47	Reserved		
48	Conversion	1. Data length in string control byte is zero for CVTMC or CVTSC instruction. 2. The end of source is encountered before the end of a compression string in CVTSC. 3. A compression string describes a character string which would cross a record boundary in the receiver CVTSC.	Terminated Terminated Terminated
4A	Invalid Floating-Point Conversion	When overflow, infinity, or not-a-number precludes accurate representation in binary format.	Suppressed
4C-7F	Reserved		
80	Invalid Segment	1. Operand addresses are not within the same segment group. 2. Segment or segment group specified by first operand does not exist.	Suppressed Suppressed

Program Exceptions and Causes (continued)

Hex Code	Exceptions	Causes	Instruction Is
81	Invalid Page	Segment group size was less than 16 megabytes and a reference was made to an address that would have been valid if segment had been larger (PPR).	Suppressed
82	Page Read Error	Permanent I/O error while reading page from auxiliary storage.	Terminated
83	Invalid Pool State	Too many pages pinned to perform bring or clear with pin (PPR).	Suppressed
84	Invalid Pin Request	1. Attempted pin was 256th pin for same page. 2. Unpin attempted on unpinned page.	Suspended
85	Invalid Write Request	Write requested to a pinned page.	Suppressed
86-8F	Bad Main Storage Page Frame	Changed data in main storage could not be accessed due to a memory failure.	Terminated

The cause of the exception is identified in the exception code field of a CRE (call/return element). See Chapter 6 for the bit assignment of this field.

The ILC (instruction length count) is a 3-bit code that provides the length of the last instruction executed. The ILC permits identifying the instruction causing the exception when the IAR (instruction address register) designates the next sequential instruction. The value of the ILC indicates the number of bytes that the IAR has been incremented. The status field of a CRE or a TDE (task dispatching element) contains the ILC after an exception has occurred.

Program exceptions are treated according to the cause. The instruction being executed at the time of the exception is handled in one of the following ways:

- *Completed*–the instruction is allowed to continue to completion with predictable results and the IAR is advanced to the next instruction address. The ILC indicates the length of the completed instruction.

- *Terminated*–the instruction is terminated at the point of the exception with unpredictable results and the IAR is advanced to the next instruction address. The ILC indicates the length of the terminated instruction.

- *Suppressed*–the instruction is not allowed to continue and the IAR is advanced to the next instruction address. The result fields are not changed. The ILC indicates the length of the suppressed instruction.

- *Nullified*–the instruction is stopped with the IAR not advanced to the next instruction address. The ILC is set to zero.

- *Suspended*–the instruction is stopped at the point of the exception and checkpoint data is stored in a reserved area. So that the operation can be resumed at the point of the exception, the IAR is not advanced to the next instruction address. The ILC is set to zero.

Permanent Storage Assignments

CONTROL ADDRESS TABLE

To execute the IMP tasks, the location of certain control information must be known to the processor. This information includes all of the system-known queue headers, the addresses of the system exception handling routines, and the storage management parameters including the storage page tables. This control information is located in the segment at hex 0000 0100, beginning at offset hex 0000.

ASSIGNED VIRTUAL STORAGE LOCATIONS

The control address table entries are shown in Figure 2-2. Those entries that have not already been described are covered in subsequent chapters.

The addresses are right aligned on doubleword boundaries for ease of indexing by the processor. The objects pointed to by the addresses in the table (except for the Function Routine Address Table or the first available hold record, neither of which references resident data) must be resident in main storage and properly aligned; otherwise a machine check occurs. An improper alignment of that object causes a program specification exception.

The leftmost 2 bytes of each table entry are reserved, and except where specified, must be set to zero.

Byte (Hex)	Use Code[1]	Virtual = Real	Address Of
0	d	Yes	Main store defective frame table (If an alternate IMPL is performed, the length of the overlay area [in bytes] is placed in the high-order [leftmost] 2 bytes for use by Service Monitor 1; otherwise, the high-order 2 bytes are unused.)
8	a	Yes	HMC overlay area (If an alternate IMPL is performed, the length of the overlay area (in bytes) is placed in the high-order (leftmost) 2 bytes for use by Service Monitor 1; otherwise, the high-order 2 bytes are unused.)
10	a	Yes	Hash table address and the number of entries-1 in the leftmost 2 bytes[2]
18	a	Yes	Primary directory address and the number of entries-1 in the leftmost 2 bytes
20	a	No	I/O event stack (must be pinned and V=V)
28	a/b[3]	Yes	I/O register table
30			Reserved
38	b	Yes	Machine check log buffer
40	b	Yes	Machine check handler
48	c	Yes	Current TDE (task dispatching element) or previous TDE when the processor is in the wait state
50	a	No	Prime TDQ (task dispatching queue)
58			Reserved
60	b	No	SVL (supervisory linkage) table
68	b	No	ACQ (available CRE queue)
70	b	No	Repetitive interval timer doubleword
78	b	No	SRC (send/receive counter) for interval timer
80	b	No	SRC for clock comparator
88	a	Yes	Hash table for hold/free instructions
90	c	No	First available hold record for hold/free instructions
98	b	No	Task switch trace table (must be pinned if V=V)
A0	b	No	FRAT (function routine address table)
A8	b	No	Instruction address sampler control block

[1] Use codes:

a. Loaded into the processor at IPL (initial program load) or IMPL (initial microprocessor program load) time.
b. Referenced by the processor whenever an address is needed by HMC (horizontal microcode).
c. Altered by the processor as required.
d. Loaded and used at IMPL time.

[2] Programming note: The number of entries in the hash table must be a power of two.

[3] The use code is *a* for Models A and C and *b* for all other models.

Figure 2-2. Assigned Virtual Storage Locations

The continuation of the control address table (MCA, machine communications area) is described in the *Vertical Microcode Data Areas* manual.

Chapter 3. Horizontal Microcode Support Functions

Horizontal Microcode Procedures

An HMC (horizontal microcode) procedure has many of the characteristics of an IMP (internal microprogramming) procedure in that it uses the IMP facilities and operates on operands in storage. It is different in that the horizontal microcode instructions execute directly on the processor hardware and are addressed by the CSAR (control store address register) rather than the IAR (instruction address register).

HMC procedures, the primary communications device in the OU (operational unit) tasks (see Chapter 7), perform built-in processor functions that support the IMP. These procedures provide a highly developed yet controlled operation to enhance the performance of the processor. HMC procedure functions are distinguished in this manual from the processor built-in functions as follows:

- HMC procedures can incur page faults and other IMP exceptions; processor built-in functions cannot incur page faults and other IMP exceptions.

- HMC procedures compete with IMP procedures and other HMC procedures for system resources via the task dispatching structure; processor built-in functions execute immediately (on the next IMP instruction boundary or interruptible point if the instruction is interruptible) when invoked.

An HMC procedure can use the same processor built-in functions that are invoked by an IMP procedure via an IMP instruction. An example of this could be a particular queueing (causing to wait) function that is invoked via an IMP instruction as well as an HMC procedure. When done via an IMP instruction, this is referred to as an *explicit* invocation. When done via an HMC procedure, it is referred to as an *implicit* invocation. While executing an HMC built-in procedure or function, no IMP instructions are executed. That function is accomplished below the IMP interface.

The IMP processor can also remain idle while waiting for work; in this condition, no dispatchable task exists.

Transfer of control from an HMC procedure via the SVL (supervisor linkage) mechanism is performed only for exceptions.

Horizontal Microcode Built-In Functions

A built-in function consists of processor operations that have the following attributes:

- If page faults or other exceptions are detected during execution, a machine check occurs. (See Chapter 9 for a description of machine check handling.)
- Built-in functions are not associated with any task.
- Built-in functions are not implemented by IMP procedures.

Built-in functions include:

- I/O event handler (see Chapter 4 for a description)
- Task dispatcher (see Chapter 5 for a description)
- Clock comparator and interval timer event signaling (see Chapter 9 for a description)
- Exception handler
- Machine check handler

Built-in functions can be invoked by:

- IMP instructions
- HMC procedures
- Other built-in functions
- Asynchronous events

TASK DISPATCHING

The execution of procedures by the processor is controlled by the tasking structure. Each IMP task is represented by an IMP object called a TDE (task dispatching element). A task then may be thought of as a unit of executable work and is composed of one or more procedures that are synchronously executed to perform that unit of work.

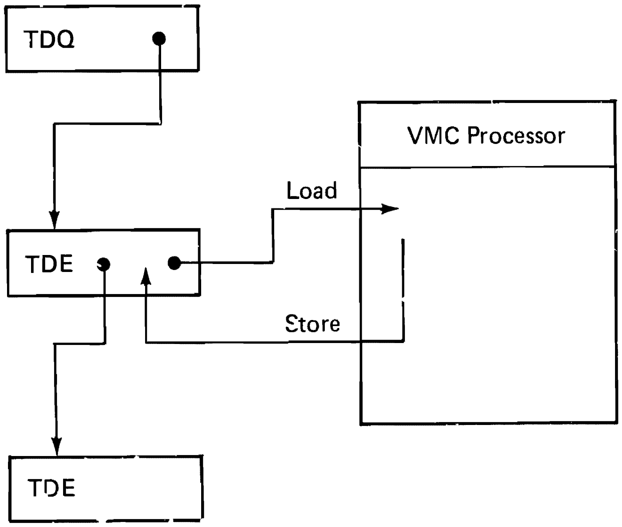
Since any task may have to wait periodically (for example, due to I/O requests or page faults), provisions are made for multiple tasks to compete for the resources of the processor, each task being represented by a TDE.

The TDEs representing dispatchable tasks (those tasks not waiting for the completion of some operation) are enqueued in priority sequence on a chained list known as the TDQ (task dispatching queue), which is also an IMP object.

To initiate processing, a built-in function known as the task dispatcher is invoked. The task dispatcher accesses the TDQ and selects the first TDE on the list as the task to begin executing.

For other than the current task, the TDE contains the current state of a task (IAR, condition code, base registers, and so forth). Therefore, the dispatcher accesses the TDE from which the processor is loaded to begin executing the current procedure of that task. This task is then referred to as the current or active task.

Conversely, the task dispatcher may place the active task in the inactive state by reversing the previous process. That is, the state of an active task is stored in its TDE. The loading and storing of the state of an IMP task is illustrated in the following figure:



Chapter 4. The Processor

The processor is the control center of the machine. It contains the sequencing and processing controls for instruction execution, tasking and exception handling, timing facilities, initial program loading, and other machine related functions.

The processor can process binary integers (in fixed or variable length), floating-point numbers, decimal integers of variable length, and logical information in fixed or variable lengths.

The processor can reference and change virtual addresses (see *Operands* in Chapter 2) in the 16 base registers. These registers are designated by a 4-bit B (base register) field in an instruction. Some instructions provide for addressing multiple base registers by having more than one B field.

Processor States

When machine power is on, the processor is in either the operational state or the stopped state.

OPERATIONAL STATE

The operational state is the normal execution state of the processor. In this state, instruction execution can proceed, built-in functions are enabled, timers are operational, and the I/O channel facilities are active.

Within the operational state, the processor may be either in the run state or in the wait state.

When the processor is in the run state, it is executing either an IMP (internal microprogramming) procedure, an HMC (horizontal microcode) procedure, or a built-in function. Conversely, when the processor is in the wait state, there are no tasks that are dispatchable.

An IMP task is dispatchable only if its TDE (task dispatching element) is enqueued to the TDQ (task dispatching queue).

Note: A task is not dispatchable if its TDE is enqueued to the wait list of an SRQ (send/receive queue) or SRC (send/receive counter). In this case, the TDE is said to be inactive and waiting.

The processor is placed in the wait state by the task dispatcher as the result of a task switch when the TDQ contains no TDEs. In this state, the processor is waiting for additional work.

Note: The processor is removed from the wait state either as the result of a built-in function issuing an implicit send operation that causes a TDE to be moved to the TDQ or as the result of a machine check.

STOPPED STATE

The processor can be put into three different stopped states:

- Processor stop
- Microprocessor stop
- Check stop

The processor can be put into the processor stopped state through the machine console. In the processor stopped state, no IMP instructions are executed and the interval timers are not updated. The time-of-day clock and the clock comparator are still operational. Events from I/O, timers, and SCA (system control adapter) are still handled.

The processor can be put into the microprocessor stopped state through the machine console. In this state, no HMC or IMP instructions are executed. The interval timers, time-of-day clock, and clock comparator are not updated, and exceptions from I/O and timers are not handled.

The processor can be put into the check stopped state via one of the following mechanisms:

- The built-in HMC machine-check function determines that a terminating machine check has occurred.
- The processor hardware encounters a terminating machine check.
- A VMC procedure issues a Terminate Immediately instruction while in machine-check mode.

The check-stopped state is a special form of the microprocessor stopped state that normally requires an IMPL (initial microprogram load) operation in order to restart the processor (see *Check Stop* in Chapter 9 for further information).

Input/Output and Asynchronous Events

In IMP, all I/O operations and communications with asynchronous processing run concurrently with task execution, and are handled as intertask exchanges of messages on queues. I/O devices, external processors, and asynchronous operations appear to have characteristics similar to an IMP task. Rather than interrupting IMP processing to signal an event or condition, I/O devices and asynchronous events are handled by the I/O event handler and the OU (operational unit) task.

QUEUE INTERFACE

All I/O operations are handled by a queuing structure. An IMP task, in control of an I/O operation, sends a command to an OU command queue used as input to an OU task. After command execution, the OU task sends the command response to the IMP task. The IMP task (for example, IOM) completes the I/O processing cycle when it accepts the response.

I/O EVENT HANDLER, OPERATIONAL UNIT TASK, AND I/O DEVICES

The I/O event handler and the OU task connect tasks executing in the processor with I/O devices. The OU task directs the flow of information between main storage and I/O devices, relieves the processor of communicating directly with the devices, and lets IMP task execution proceed concurrently with I/O operations. I/O devices include card readers and punches, magnetic tape units, disk storage, printer-keyboard devices, printers, and teleprocessing equipment.

I/O device operation can be handled by a control unit. The control unit can be an integral part of the I/O device attachment or an external unit. The control unit provides the logic and buffering necessary to operate its I/O device. From a programming view, control unit functions merge with I/O unit functions. Other I/O device operations are controlled by hardware adapters. The processor and storage use the channel as an interface to I/O devices and their control units. The channel directs the flow of data between I/O devices and storage (for details of the channel interface, see *IMP Channel Objects* in Chapter 7).

SYSTEM CONSOLE

The system console is used to operate the machine. The console consists of an operator/service panel, a display, and a keyboard.

The operator/service panel indicates system status and provides the operator with controls to intervene in normal programmed operations. The display and keyboard allow the operator to communicate with supervisory and problem programs. (See *System Control* in Chapter 9 for a list of the system console functions.)

Chapter 5. Tasking

This chapter describes the structures and operations of the tasking functions used by the processor.

IMP (internal microprogramming) tasking is the process of controlling the execution of IMP tasks. An IMP task is characterized by the synchronous execution of one or more IMP procedures. An IMP procedure is composed of an IMP instruction stream, the data used by the instruction stream, and the parameters and arguments used to pass information between IMP procedures.

PROCEDURE EXECUTION

At any time, the status of the processor is one of the following:

- Executing an IMP instruction in an IMP procedure

- Executing an HMC (horizontal microcode) built-in function

- Executing an HMC instruction in an HMC procedure

- Idle (waiting for additional work)

An IMP procedure is executed in the environment of an IMP task. The primary control structure of an IMP task is the TDE (task dispatching element). While executing an IMP procedure, IMP instructions are fetched sequentially by the processor from the storage location addressed by SID (segment identifier) register 0 plus the 2-byte IAR (instruction address register). As each instruction is fetched, the IAR is incremented by the number of bytes in that instruction so as to address the next instruction. The current instruction is then executed, and the same steps are repeated, using the new IAR value. Sequential execution of instructions within the current procedure can be changed by branching within the procedure or by causing a transfer of control to another procedure. If control is transferred to a new procedure, the new procedure can execute under the same task (as with a supervisor linkage operation) or can execute under a different task (as with a task switching operation). In addition, control can be transferred to a new procedure in any of the following ways:

- The current IMP procedure specifies an implicit or explicit SVL (supervisor linkage) instruction (see *Supervisor Linkage* in Chapter 6).

- A program exception occurs causing a built-in processor function (see *Horizontal Microcode Built-In Functions* in Chapter 3) to pass control to the IMP exception handling procedure. The processor passes control to the IMP exception handling procedure via a special invocation of the SVL function termed the exception SVL function (see *Supervisor Linkage* in Chapter 6).

- The current IMP procedure issues a send or receive instruction and a task switch occurs.

- An I/O or timer event occurs causing a task switch.

- A machine check occurs causing control to be passed to the IMP machine check handler (see *Machine Check Handling* in Chapter 9).

No IMP instructions are executed during the execution of an HMC procedure or built-in function.

The processor can also be in an idle condition waiting for work. In this condition, no dispatchable task exists.

BASE REGISTER ASSIGNMENTS

There are 16 base registers that can contain addresses during IMP procedure execution (see *Register Descriptions* in Chapter 2).

Base register 0 points to the start of the instruction stream. All instruction addressing and branching within a procedure is relative to B(0). Base registers hex 1, 2, and E, and byte register hex E are designated to receive parameters during explicit or implicit SVLs (see *Supervisor Linkage Control* in Chapter 6). Base register 3 is used by the Function Call Double instruction to point to the stack. Byte register hex F is used by the Translate and Test instruction, the Move Packed Shifted instruction, and the Move Packed Shifted Zero instruction. Byte registers hex A and B and base registers hex E and F are used by the Edit Pack Decimal instruction. Halfword registers hex E and F are used by the Convert Character to SNA, Convert Characters to MULTI-LEAVING Remote Job Entry, Convert MULTI-LEAVING Remote Job Entry to Character and Convert SNA to Character instructions. Halfword register hex E is used by the Trim instruction. However, these uses do not preclude the use of these registers for other operations.

The remaining base registers have no specific assignments and can be used to address various spaces (up to 64 K-bytes each) in virtual storage.

I/O INTERRUPTIBILITY

Pending interrupts are normally granted following instruction execution. Two additional special interrupt tests ensure minimal interruption delay.

The first special interrupt test is at the end of a unit of operation (the amount of CPU processing that occurs between interrupt points). Uninterruptible IMP instructions normally use one unit of operation. Interruptible IMP instructions, built-in functions, and HMC procedures may use multiple units of operation. The microcode handles interruptible IMP instructions and built-in functions by checkpointing the function to the beginning of the next unit of operation and then granting the interrupt. When control is returned to the procedure containing the interrupted function, execution is resumed at the checkpointed unit of operation.

The checkpoint facility varies with the IMP instruction or built-in function (for example, MVCL or GHRF). The interrupt checkpoint facilities are described as a part of the instruction specification.

The second special interrupt test is required by instructions that take an unusually long time to execute. This class of instructions has the following characteristics:

- The instruction or built-in function is not designed as being interruptible.
- The worst-case path exceeds 450 microseconds, including overlay time and lookaside buffer-miss time.

The special interrupt tests are inserted into HMC to ensure a response time of 450 microseconds. If an I/O interrupt is pending when a special test is performed, one of two procedures is used:

- If the result is computed internally in an HMC work area and is not stored back until the computation has been completed, the partially computed result is discarded, the IAR (instruction address register) is nullified, and the interrupt is granted. Execution is resumed at the beginning of the instruction or built-in function that was interrupted. The IMP interface interprets the partially completed computation as if it were never performed by the CPU.

  Note: Interrupt tests are performed before IMP facilities (result field and condition code) have been modified.

- If the result is stored back as it is being computed, HMC checkpoints the instruction or built-in function internally, processes the I/O interrupt (any resultant task switch does not occur until the instruction or built-in function is completed), and resumes processing the instruction at the point of interrupt detection.

This type of interrupt processing can occur during the execution of a TR or TRR instruction, or during the execution of built-in functions that perform CRE (call/return element) chain searches or move-TDEs (task dispatching element [from one queue to another]), or during IMP queuing instructions. The queuing (built-in function or IMP instruction) checkpoint mechanism is described under *Send/Receive Queue Busy Status* later in this chapter.

INTERNAL MICROPROGRAM TASKING

IMP tasking allows task switching from a procedure in a given task to a procedure in a different task. All task switches are caused by a built-in processor function known as the task dispatcher.

The following paragraphs deal with the tasking structure of the IMP, describe the objects that make up the tasking structure, and describe how the tasking structure is used.

TASKING STRUCTURE

Task Dispatching Queue

One or more TDQs (task dispatching queues) exist in the system. The prime TDQ is used by the task dispatcher to allocate processor time to the active tasks in the system. The elements chained to the prime TDQ are those TDEs (task dispatching elements) associated with dispatchable tasks (for example, tasks not waiting for a message from an SRQ [send/receive queue]). TDEs for the dispatchable tasks are ordered on the prime TDQ in priority sequence according to the priority field in the TDE.

The format of a TDQ is as follows:

Descriptor	First TDE Address

0 Bytes 2 8

Bytes (Hex)	Bits	Description
0-1		**Descriptor**:
	0-2	Identifies this IMP object as a TDQ (= 000).
	3	0 The TDQ is empty.
		1 This TDQ has one or more TDEs.
	4-15	Reserved.
2-7		**First TDE Address**: First TDE if any, associated with this TDQ.

When accessed as the result of a send or receive type instruction, a TDQ must be resident in storage, fullword aligned, and must not cross a page boundary; otherwise a machine check will occur when the TDQ is accessed. However, if any of the above conditions are not met for EQTDE (Enqueue Task Dispatching Element) or DQTDE (Dequeue Task Dispatching Element) instructions, a specification, addressing, or address translation exception occurs.

Task Dispatching Element

A TDE (task dispatching element) is an IMP object used to identify a task and the attributes (including a priority) associated with that task. It also contains fields used to store or load the current state of the task at the time of a task switch. The TDE for a particular task can appear as an element on a TDQ (task dispatching queue) or can be enqueued to an SRQ (send/receive queue) or an SRC (send/receive counter) wait list. If a task is eligible for instruction execution, the associated TDE appears on the prime TDQ.

The format of the TDE is as follows:

Descriptor	Next TDE Address	Priority
0 Bytes	2	8

Control Mode	First CRE Address	Exception Mask
C Bytes	E	14

Current Queue Address	Exception Occurrence
16 Bytes	1C

TDQ Address	Time Quantum
1E Bytes	24

Time Quantum	Status	Address Register	Base Registers	
29 Bytes	2C	30	32	

Used by VMC	TDE Identifier	Hold Count	Exception Code
92 Bytes	94	96	98

PEM Start Address	Computational Attributes	Not Used
9A Bytes	A0	A1

PEM Stop Address	Used by VMC	
A2 Bytes	A8	

Reserved for VMC		Checkpoint Area	Reserved	for HMC
B0 Bytes		C0	C6	

Used by VMC	
D0 Bytes	FF

Bytes (Hex)	Bits	Description
0-1		**Descriptor:**
	0-2	Identifies this IMP object as a TDE (= 001).
	3	Reserved.
	4	0 This is the last TDE on the chain. 1 This is *not* the last TDE on the chain.
	5	0 This TDE is free to be enqueued to a queue header. 1 This TDE is already enqueued to a queue header. On a Dequeue TDE instruction, this bit is reset, indicating that this element is no longer enqueued to any queue. On an Enqueue TDE instruction, this bit is checked first. If it is one, a specification exception is raised. If it is zero, the TDE is enqueued and this bit is set to one.
	6	0 This TDE was not removed from the prime TDQ by a SENDMW instruction. 1 This TDE was removed from the prime TDQ by a SENDMW instruction.
	7	0 The address of the TDQ to which the TDE will be moved when dequeued from a wait list is contained in bytes hex 1E-23 of the TDE. 1 The TDE removed from a wait list by a send operation is to be enqueued on the prime TDQ.
	8-15	Used by the processor to monitor access of this TDE. If not hex 00, the TDE is monitored for access exceptions.
2-7		**Next TDE Address**: Address of the next TDE. If this is the last TDE in the chain, descriptor bit 4 = 0.
8-B		**Priority:**
	0-31	Highest priority is zero. TDEs are enqueued in priority sequence, last within the same priority, when moved to a TDQ by a send type operation or Enqueue TDE instruction.
C-D		**Control Mode**:
	0-1	Reserved.
	2	0 Do *not* perform trace function. 1 Perform task trace function.
	3	0 Any SVLM1 instruction executes as a no-operation. 1 Any SVLM1 instruction executed in this task defaults to an SVL1 instruction.
	4	0 The CRE (call return element) list is empty. 1 One or more CREs are chained to this TDE.
	5	0 Task not timed. 1 Timed task.
	6	0 Not in PEM (program event monitor) mode. 1 In PEM mode.

Bytes (Hex)	Bits	Description
C-D (cont)	7	0 Any SVLM instruction executes as a no-operation. 1 Any SVLM instruction executed in this task defaults to an SVLO instruction.
	8-15	The maximum number of available CREs to be left chained to the TDE by the execution of an SVX instruction. Whenever the number of available CREs would exceed this number as a result of an SVX instruction, one is returned to the ACQ (available CRE queue). At least one available CRE is always left chained to the TDE by an SVX instruction.
E-13		**First CRE address:** If no CREs are associated with this TDE, bit 4 of byte hex C = 0.
14-15		**Exception Mask:** Bit off = masked, bit on = allowed.
	0	Binary overflow.
	1	Decimal overflow.
	2-4	Reserved.
	5	Monitored SRQ header.
	6	Monitored SRM header.
	7	Monitored TDE header.
	8	PEM (see Chapter 9).
	9	Address compare (see Chapter 9).
	10	Floating-point overflow.
	11	Floating-point underflow.
	12	Floating-point zero divide.
	13	Floating-point inexact result.
	14	Floating-point invalid operand.
	15	Reserved.
16-1B		**Current Queue Address:** Address that TDE is enqueued to. If TDE is not enqueued to an SRC, SRQ, or TDQ, descriptor bit 5 = 0.
1C-1D		**Exception Occurrence:**
	0	Binary overflow.
	1	Decimal overflow.
	2-4	Reserved.
	5	Monitored SRQ header.
	6	Monitored SRM header.
	7	Monitored TDE header.
	8-9	Reserved.
	10	Floating-point overflow.
	11	Floating-point underflow.
	12	Floating-point zero divide.
	13	Floating-point inexact result.
	14	Floating-point invalid operand.
	15	Reserved.
1E-23		**TDQ Address:** If descriptor bit 7 = 0, these bytes contain the address of the TDQ to which the TDE it to be enqueued when removed from a wait list by a send operation. If descriptor bit 7 = 1, the address contained in these bytes is ignored and the TDE is enqueued to the prime TDQ.
24-2B	0-41	**Time Quantum:** The time remaining in this task (bit 41 = 1024 microseconds).
	42-63	Reserved.

Bytes (Hex)	Bits	Description
2C-2F		**Status:**
	0	Reserved.
	1	0 IMP procedure.
		1 HMC procedure.
	2-7	Not used.
	8-15	Reserved.
	16-20	Zero.
	21-23	ILC (instruction length count).
	24-27	Zero.
	28-31	Condition Code: When initializing the condition code field in a new TDE, at least one, but not all, of the bits must be set to 1. Failure to do so may cause branch instructions to work incorrectly.
30-31		**Address Register**:
		IAR (instruction address register) if bit 1 of byte hex 2C = 0.
		CSAR (control storage address register) if bit 1 of byte hex 2C = 1.
32-91		**Base Registers**: Note that all 16 base registers are always saved and restored on a task switch. The registers occupy 6 bytes per register beginning with byte hex 32.
92-93		**Used by VMC.**
94-95		**TDE Identifier.**
96-97		**Hold Count**: Object hold count for this TDE.
98-99		**Exception Code**: Refer to bytes hex 74-75 of the CRE definition in Chapter 6.
9A-9F		**PEM Start Address.**
A0		**Computational Attributes.**
	0	Reserved.
	1-2	Rounding Mode.
		00 Round toward positive infinity.
		01 Round toward negative infinity.
		10 Round toward zero.
		11 Round to nearest.
	3-7	Reserved.
A1		**Not used.**
A2-A7		**PEM Stop Address.**
A8-AF		**Used by VMC.**
B0-BF		**Reserved for VMC.**
C0-C5		**Checkpoint Area.** The hold/free functions use this area to pass exception information.
C0-C1		**Hold Hash Table Entry Offset**: Contains the offset when a second chain search exception is encountered during a Grant instruction, or when a monitored exception is encountered during a Free instruction.

Bytes (Hex)	Bits	Description
C2-C3		**Index or Pointer:** Contains either • An index to the hold record preceding the hold record that is at the head of the secondary chain after a second chain search exception for a Grant instruction, • A pointer to the head of the secondary chain for a second chain search exception on a Free instruction, or • A pointer to the hold record (on the primary chain) preceding the monitored hold record on a monitored exception.
C4-C5		**Index:** An index pointing to an available hold record.
C6-CF		**Reserved for HMC.**
D0-FF		**Used by VMC.**

The TDE extends beyond byte hex CF. This portion, which is not used by HMC, is described in the *Vertical Microcode Data Areas* manual.

Task switching mode/task control mode (bytes hex C-D) for bits 2, 3, 5, 6, and 7 is established by the processor at task switch time when a task becomes active. If the task control mode bits 2, 3, 5, 6, and 7 are changed in a TDE while its task is active, the new mode does not become effective until the next time the task becomes active as the result of a task switch. However, the Dispatch TDQ instruction will test bit 6 and will appropriately enable or disable PEM mode for an active task. The Dispatch TDQ instruction also tests bit 1 for a task switch trace, tests control mode bits 3 and 7, and enables or disables the task-wide SVL-monitored no-operation.

The computational attributes for a task are set when the task executes its first floating-point instruction after being dispatched. The floating-point inexact result exception mask and the exception occurrence bits are also set at this time. The attributes remain in effect (even though bytes 15, 1D, or A0 of the TDE may change) until the task is dispatched once again. In order to ensure that the computational attributes being used match the computational attributes in the TDE, a Dispatch TDQ instruction for the current TDQ should be executed. The exception mask (bytes 14 and 15) is tested each time a maskable exception is recognized to determine if the exception should be taken.

The following exceptions are maskable:

- Address compare (see Chapter 9)
- Binary overflow
- Decimal overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Floating-point zero divide
- Monitored SRQ header
- Monitored SRM header
- Monitored TDE header
- Program event monitor (see Chapter 9)

The following exceptions can occur at the same time as any other IMP exception:

- Monitored ACQ descriptor (SVL receive)
- Monitored CRE descriptor (SVL receive)
- Monitored TDE descriptor (SVL receive)
- Address compare
- Task interval timer

Bytes hex 14 and 15 (exception mask) are tested each time a maskable exception is recognized. If the exception is not masked, the occurrence is recorded in the CRE (call/return element) and an SVL (supervisor linkage) is taken. If the exception is masked, the occurrence is recorded in the exception occurrence field of the TDE and an SVL is not taken. See Chapter 6 for further information in the handling of exceptions.

TDEs must be resident in storage, fullword aligned and must not cross a page boundary; otherwise a machine check occurs when the TDE is accessed as the result of a send or receive type operation. When the TDE is accessed as a result of an EQTDE or DQTDE instruction, a specification, addressing, or address translation exception occurs.

Send/Receive Queue

An SRQ (send/receive queue) is an IMP object used to exchange intertask information and to synchronize the flow of control between tasks. One task can communicate with another task by issuing a send type instruction to an SRQ or an SRC (send/receive counter). Another task can then obtain the information from the queue or counter by issuing a receive type instruction.

Task synchronization is provided by using send/receive messages and an SRQ in the following manner. When a procedure within the active task issues a Receive Message instruction and the target SRQ either (1) has no messages, or (2) has no message that satisfies the search argument for the Receive Message instruction, the task does not proceed. Instead, the task is placed in the receive wait state by the processor by dequeuing its TDE (task dispatching element) from the TDQ (task dispatching queue) and enqueueing that TDE to the wait list of the target SRQ. The task dispatcher is then invoked to determine the next task to be activated.

A Send Message instruction is the counterpart to a Receive Message instruction. If a message has been enqueued by a Send Message instruction to an SRQ and there are TDEs waiting the value of byte 0 bit 7 of the SRQ determines the action taken. If bit 7 equals:

0 The TDEs are dequeued from the SRQ wait list and enqueued in priority sequence on their appropriate TDQ.

1 The *first* TDE is dequeued from the SRQ wait list and enqueued in priority sequence on their appropriate TDQ.

The task dispatcher is then invoked if the task switch control bit is zero (bit 15 of the SENDM instruction) and a TDE was enqueued to the TDQ at a higher priority than the current TDE. If these conditions are present, a task switch occurs. This switch is referred to as a *preempt wait* to the task issuing the send operation.

Send and receive type operations are executed explicitly as instructions by IMP tasks as well as implicitly by HMC functions.

The format of the SRQ header is as follows:

Descriptor	First TDE Address	Reserved	Key Lth-1
0 Bytes	2	8	9

First Message Address	Reserved	
A Bytes	10	20

Bytes (Hex)	Bits	Description
0-1		**Descriptor:**
	0-2	Identifies this IMP object as an SRQ (= 010).
	3	0 The SRQ header contains no TDEs (task dispatching elements).
		1 One or more TDEs are enqueued to the wait list.
	4	0 The SRQ contains no SRMs (send/receive messages).
		1 One or more SRMs are enqueued to the message list.
	5	0 An access is *not* in progress.
		1 An access is in progress by a task whose TDE address is indicated in the reserved field. An access attempt by any other task causes a busy exception.
	6	0 No monitored TDEs are enqueued to the wait list.
		1 One or more monitored TDEs are enqueued to the wait list.

Note: This bit is set/reset by the VMC and tested by the HMC.

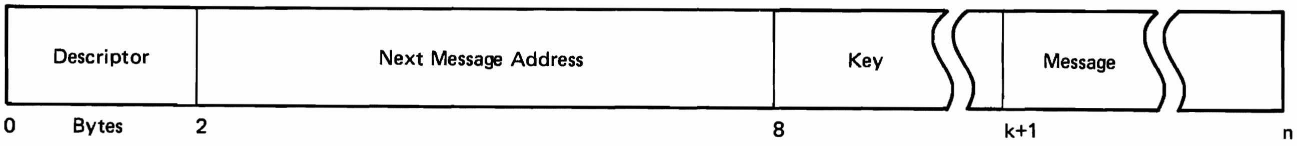
Bytes (Hex)	Bits	Description
0-1	7	0 All TDEs are moved to the appropriate TDQ when a message is enqueued by a Send Message instruction to the SRQ.
		1 The first TDE is moved to the appropriate TDQ when a message is enqueued by a Send Message instruction to the SRQ.
	8-15	Used by the processor to monitor accesses to this object. If not hex 00, the SRQ is monitored for access exceptions.
2-7		**First TDE Address:** If no TDEs are waiting, descriptor bit 3 = 0.
8		**Reserved.**
9		**Key Length-1:** Number of bytes of message key, starting with byte 8 in the SRM.
A-F		**First Message Address:** If no SRMs are enqueued, descriptor bit 4 = 0.
10-1F		**Reserved:** Bytes hex 10-11 and hex 18-19 contain checkpoint status. Bytes hex 12-17 contain the owner TDE address if the queue is busy (descriptor bit 5 = 1). Bytes hex 1A-1F contain the last message searched address if interrupted during a message search.

Note: An SRQ header must be fullword aligned and not cross a page boundary; otherwise a specification exception occurs when the object is referenced.

Send/Receive Message

The messages that are enqueued to the SRQ (send/receive queue) take the form of an IMP object called an SRM (send/receive message).

The format of an SRM header is as follows:



Bytes (Hex)	Bits	Description
0-1		**Descriptor:**
	0-2	Identifies this IMP object is an SRM (= 011).
	3	Reserved.
	4	0 This is the last SRM on the chain. 1 This is not the last SRM on the chain.
	5	0 SRM is free to be enqueued to a queue header. 1 SRM is already enqueued to a queue header. This bit is set to zero by those instructions that dequeue an SRM from a queue header. Those instructions that enqueue an SRM check this bit first. If it is one, a specification exceptions occurs. If it is zero, the SRM is enqueued to the designated queue and this bit is set to one.
6	0	A SENDMW instruction has not been issued.
	1	A SENDMW instruction has been issued. When the OU task has finished processing this SRM, the TDE specified at byte 122 of the SRM is enqueued to the TDQ indicated by this TDE. When the OU attempts to restore the TDE to the TDQ, the TDE address field of the SRM must be contained in the same page as byte 0 of the SRM or a machine check occurs. When a SENDMW instruction is being executed, the SRM must contain the address of a TDE at byte hex 7A. This address must reside on the same page as the SRM descriptor or a machine check will occur.
	7	Reserved.

Bytes (Hex)	Bits	Description
0-1	8-15	Used by the processor to monitor accesses of this object. If not hex 00, the SRM is monitored for access exceptions.
2-7		**Next Message Address:** If no additional SRMs are enqueued, descriptor bit 4 = 0.
8-k		**Key:** Message key of the length indicated in the SRQ, plus the message text. The length of the key will determine the location of the starting point (k+1) of the message.
(k+1)-n		**Message:** Text of message does not have a set length.

Note: While the SRM message text may cross page boundaries, the SRM descriptor, next SRM pointer, and SRM message key must be in the same page and the SRM descriptor must be fullword aligned or a specification exception occurs when the SRM is referenced.

The key field is not checked for a page crossing when an enqueue-first or enqueue-last operation is performed.

Send/Receive Counter

An SRC (send/receive counter) is an IMP object used in much the same way as an SRQ (send/receive queue) except that no messages are enqueued. Instead, a Send Count instruction causes the count field in the SRC header to be increased by 1 (see note).

A Receive Count instruction causes the count field value to be decreased by the counter limit value (see note) if the count field value equals or exceeds the count limit value. The count field and limit values are treated as 16-bit unsigned numbers. If the count value does not equal or exceed the limit value, the task is placed in receive wait state by dequeuing its TDE (task dispatching element) from the TDQ (task dispatching queue) and enqueuing that TDE to the wait list of the target SRC. The task dispatcher is then invoked to determine the next task to be dispatched.

When a Send Count instruction is issued, the count is incremented by 1; then a check is made to determine if the count value has reached or exceeded the limit value. If so, the associated SRC wait list is interrogated. If there are TDEs waiting and byte 0 bit 7 equals:

0 The TDEs are dequeued from the SRC wait list and enqueued in priority sequence to the TDQ specified by byte hex 1E of the TDE.

1 The *first* TDE is dequeued from the SRC wait list and enqueued in priority sequence to the TDQ specified by byte hex 1E of the TDE.

The task dispatcher is then invoked if the task switch control bit is zero (bit 15 of the SENDC instruction) and a TDE was enqueued to the current TDQ at a higher priority than the current TDE. This switch is referred to as a *preempt wait* to the task issuing the send operation.

Note: The HMC handles the increasing and decreasing count field value. See the *Processing Unit Theory-Maintenance* manual.

The format of an SRC is as follows:

Descriptor	First TDE Address	Count	Limit

0 Bytes 2 8 A C

Bytes (Hex)	Bits	Description
0-1		**Descriptor:**
	0-2	Identifies this IMP object as an SRC (= 100).
	3	0 No TDEs are enqueued to the wait list. 1 One or more TDEs are enqueued to the wait list.
	4-6	Reserved.
	7	0 All TDEs are moved to the appropriate TDQ when the limit value has been reached or exceeded. 1 The first TDE is moved to the appropriate TDQ when the limit value has been reached or exceeded.
	8-15	Reserved.
2-7		**First TDE Address:** If no TDEs are waiting, then descriptor bit 3 = 0.
8-9		**Count value.**
A-B		**Limit value.**

Note: An SRC must be fullword aligned and not cross a page boundary; otherwise, a specification exception occurs when the SRC is referenced.

Enqueue/Dequeue Instructions

Enqueue and Dequeue instructions are used to control IMP objects composing the tasking structure. The enqueue instructions insert SRMs (send/receive messages) on SRQs (send/receive queues) and insert TDEs (task dispatching elements) on wait lists of SRQs or SRCs (send/receive counters), or on TDQs (task dispatching queues). Dequeue instructions remove SRMs from SRQs and remove TDEs from the wait lists of SRQs, SRCs, or TDQs. Unlike send/receive instructions, enqueue/dequeue instructions neither invoke the task dispatcher nor cause a task switch.

Send/Receive Queue Busy Status

Some of the send, receive, enqueue, and dequeue instructions are interruptible. Figure 5-1 shows the interruption causes for instructions that are interruptible, can set busy status, and observe busy protocol. An SRQ (send/receive queue) is set into busy status by the processor whenever an instruction that accesses it is interrupted before completion. Busy status occurs as follows. When the processor detects that an SRQ access instruction must be interrupted, the descriptor busy bit (byte 0, bit 5) is set, and the pin count for the page containing the SRQ is incremented if the page is V=V page (pin count is the value of a counter used to indicate that a page is pinned [or held] in storage). A nonzero value of pin count indicates that the page is in use and should not be removed from storage (see *Primary Directory* in Chapter 8). The TDE (task dispatching element) address of the task that was executing the instruction and the instruction interruption point are then recorded in the reserved area of the SRQ.

If any task attempts to execute an instruction that accesses a busy SRQ, the TDE address of that task is compared to the TDE address saved in the header of the busy queue. If the TDE addresses are not equal, the accessing instruction is nullified and a descriptor access busy exception is raised to the issuing task. If the TDE addresses are equal, this implies to the processor that the interrupted instruction of the owner task is resuming the operation. In this case, the busy bit is reset, the pin count of the page containing the SRQ is decreased, the instruction interrupt point is restored from the checkpoint area, and instruction execution proceeds normally.

Note: The exception handling routine should not perform an operation (other than the original one) that accesses a busy SRQ in the task producing the exception (task whose TDE address is stored in bytes hex 18-23 of the SRQ). If this condition occurs, the processor assumes that the suspended instruction is being resumed, resulting in an unpredictable operation.

	Page Fault Due to a Nonresident SRM on a Message List	**Access Exception Monitored (Note 1)**		
Interruptible Instruction		**TDE**	**SRQ**	**SRM**
EQM	X		X	X
DQM	X		X	X
EQTDE		X (Note 2)	X	
DQTDE		X (Note 2)	X	
SENDM	X	X	X	X
SENDMW	X	X	X	X
RECM	X	X	X	X
SVLO		X	X	X
SVL1		X	X	X
SVL2		X	X	X

Notes:

1. Refers to nonzero descriptor byte 1 of TDE, SRQ, or SRM.
2. On Enqueue and Dequeue TDE instructions, if the first operand is a TDE or SRC, a monitored TDE does not cause an access exception.

Figure 5-1. Interruptible Instruction Summary Chart

Interrupts due to I/O are recognized as follows:

- When the processor searches the second or subsequent SRM on an SRQ (before moving a second or subsequent TDE from an SRC or SRQ wait list), or

- When the processor searches a TDE or CRE chain

  These interrupts are ignored when the processor searches the first SRM on an SRQ, or when the processor moves the first TDE from an SRC on an SRQ. When the I/O interrupt is handled, only the address events and the load multiple register events are processed. The remaining events are handled when the resumed queueing operation completes.

TASK CONTROL

Task control consists of:

- Task dispatching
- Task switching
- Task timing

Task Dispatching

The dispatching of IMP tasks is handled by an HMC function known as the task dispatcher. The task dispatcher is invoked explicitly by the Dispatch TDQ (task dispatching queue), the Enable Task Dispatching instruction, or, under certain conditions, by the send/receive type instructions or implicit send/receive operations. It is the responsibility of the task dispatcher to determine when a task switch is necessary, to determine which task should be dispatched next, and to accomplish the indicated task switch. During a task switch, the status of the old task is saved (as described under *Call/Return Element* in Chapter 6) in that task's TDE (task dispatching element). The status of the new task is taken from the new task's TDE.

The primary IMP structure associated with the task dispatching function is the TDQ. The TDE that represents the active task is located in the TDQ and is referred to as the current TDE. The current TDE is normally the top TDE on the TDQ whenever the task dispatcher is enabled. When the task dispatcher is disabled or if a send without task switch occurred, the current TDE may or may not be the top TDE on the TDQ.

The task dispatcher is invoked:

1. When a DTDQ (Dispatch TDQ) instruction is encountered.
2. When an implicit or explicit send operation (Send Message or Send Count instruction) occurs and a TDE is placed on the TDQ at a higher priority than the current TDE.
3. When an implicit or explicit receive operation (Receive Message or Receive Count instruction) occurs and the receive is not satisfied. In this case, the current TDE is removed from the TDQ and placed on an SRC or SRQ (send/receive queue) wait list by the receive operation.
4. When an Enable Task Dispatcher instruction is issued and the top TDE on the TDQ is not the current TDE.

The task dispatcher functions as follows:

- In the above cases a task switch may occur to the top TDE on the TDQ. For the third case, the TDQ may be empty. If the TDQ is empty, the processor waits until a new TDE is placed on the TDQ.
- When a new task is dispatched as a result of any of the previous conditions, the TDQ and the current TDE addresses in the control address table are updated as required. If the exception code is nonzero in the new task TDE, the exception is presented via an exception SVL. If no exceptions are present, instruction processing then commences with the instruction addressed by the IAR (instruction address register) or CSAR (control store address register) of the new task.

Dispatcher control addresses are accessed and maintained by the processor in support of the task dispatching function. These control addresses, located in the control address table (see Figure 2-2), are composed of the TDQ address and the current TDE address.

Task Switching

Having determined that a task switch is required, the task dispatcher stores the state of the old task in the TDE (task dispatching element) of the task. Stored status includes the condition code, the instruction length, the IAR (or CSAR if a horizontal microcode function), base registers hex 0-F, and the exception code. Also, bit 1 of byte hex 44 (status) in the TDE is set to indicate either an IMP or HMC procedure. The task dispatcher then determines which task is to be dispatched next as previously described. The new task TDE address is then stored in the current TDE address field of the control address table and the status of the new task is loaded from the new TDE. The control mode (defined as part of the TDE in this chapter) is established from the TDE. Any pending exceptions are presented via an exception SVL (supervisor linkage). Otherwise instruction execution is initiated beginning with the instruction addressed by the IAR and register S(0) or by the CSAR.

The processor enters the wait state as follows. The active task is the only TDE on the TDQ, and it issues a receive type operation that is not satisfied. This causes its TDE to be removed from the TDQ and placed on the wait list of the SRQ (send/receive queue) or SRC (send/receive counter) referenced by the receive type operation. The task dispatcher is then invoked and the task status is stored in the TDE of the task. The processor is then placed in the wait state since the TDQ contains no TDEs.

Subsequently, when one or more TDEs are placed on the TDQ and the task dispatcher is invoked, the status of the first TDE on the TDQ is loaded into the processor, the control address table is updated, and instruction execution commences.

Task Dispatcher Enable/Disable Functions

Two instructions are provided to allow disabling and enabling of the task dispatcher function. The Disable Task Dispatching instruction inhibits the task dispatcher and stops the task interval timer. While in this mode, no task switches can occur. Furthermore, a machine check occurs if a Receive Message, Receive Count, or Dispatch Task Dispatching Queue instruction, or an SVL (supervisor linkage) is attempted. This condition is also entered implicitly as a result of a machine check. The Enable Task Dispatching instruction resets this condition, starts the task interval timer if the current task is timed and no task switch occurs, and invokes the task dispatcher. If an exception occurs while the task dispatcher is disabled, it is reported as a machine check (see *Processor Machine Check Handler* in Chapter 9).

Task Timing

An IMP task can be either timed or untimed as indicated by bit 5 of byte hex 12 in the task TDE (task dispatching element). Task timing is provided by a built-in function called the task interval timer. When a timed task is activated by the task dispatcher, the time quantum bytes (hex 24-2B) of the TDE are loaded into the task interval timer. If this timer decreases to zero while a timed task is active, a task timer exception occurs. For untimed tasks, the time quantum field is not used and the task interval timer is not decreased.

When a timed task is set to the wait state as part of a task switch, the contents (residual value) of the task interval timer are stored into the time quantum field of the task TDE.

Note: If the task dispatcher is disabled by either the Disable Task Dispatching instruction or a machine check and the active task is timed, the task interval timer is stopped. When an Enable Task Dispatching instruction is issued, the following operation results. If a task switch occurs, the new task TDE specifies timed or untimed. If a task switch did not occur, timing is resumed by the task interval timer if so indicated by the current TDE.

If an untimed task issues a Set Interval Timer instruction to the task interval timer, a specification exception is presented. A Store Interval Timer instruction by an untimed task that specifies the task interval timer stores unpredictable results.

INTERTASK COMMUNICATIONS AND SYNCHRONIZATION

Communication between tasks and control of synchronization is provided by the send/receive mechanism within the tasking structure.

An example of intertask communication is shown in Figure 5-2. The SRQ-A is initially empty. Task A then executes a Send Message 1 instruction which enqueues the message to the SRQ. Subsequently, when task B is dispatched and issues a Receive Message instruction 2, the message is dequeued from the SRQ for use by task B.

The synchronization function of these two instructions is illustrated in Figure 5-3. The first two tasks on the TDQ are TDE B and TDE A; TDE B has the higher priority. Task B is the active task and the processor is executing procedure B (Figure 5-3[a]).

When the RECM C instruction is executed, a message cannot be dequeued since SRQ C has no message.

Therefore, the IAR is not incremented and the receive operation places task B in the receive wait state as follows:

- Dequeues TDE B from the TDQ
- Enqueues TDE B to the wait list of SRQ C (Figure 5-3[b])
- Invokes the task dispatcher

The task dispatcher then performs a task switch as follows:

- Determines that the current TDE (TDE B) is not first on the TDQ
- Stores the state of task B in TDE B
- Determines that TDE A is now the highest priority dispatchable task
- Loads the state of task A (procedure A) from TDE A
- Updates the control address table entry for the current TDE
- Initiates processing of procedure A

Procedure A now issues a SENDM C instruction which enqueues an SRM to SRQ C. Since TDE B is on the wait list, the send operation also:

- Dequeues TDE B from SRQ C
- Enqueues TDE B to the TDQ above TDE A (Figure 5-3[c]) since TDE B has a higher priority
- Invokes the task dispatcher since a TDE was moved to the TDQ

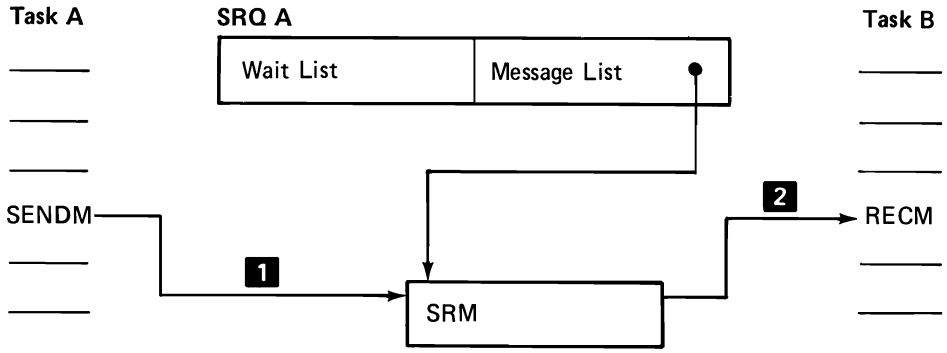


Figure 5-2. Intertask Communications

The task dispatcher then:

- Determines that the current TDE (TDE A) is not first on the TDQ
- Stores the state of task A in TDE A
- Determines that TDE B is now the highest priority dispatchable task
- Loads the state of task B
- Updates the control address table entry for the current TDE
- Initiates processing of procedure B

Since the IAR for task B still points to the RECM C instruction, it is again executed. The SRM is now dequeued and execution of procedure B continues under task B since the receive operation was satisfied.

Similar functions are also associated with the SRC (send receive counter) object together with the Send Count and Receive Count instructions except that no messages are passed.

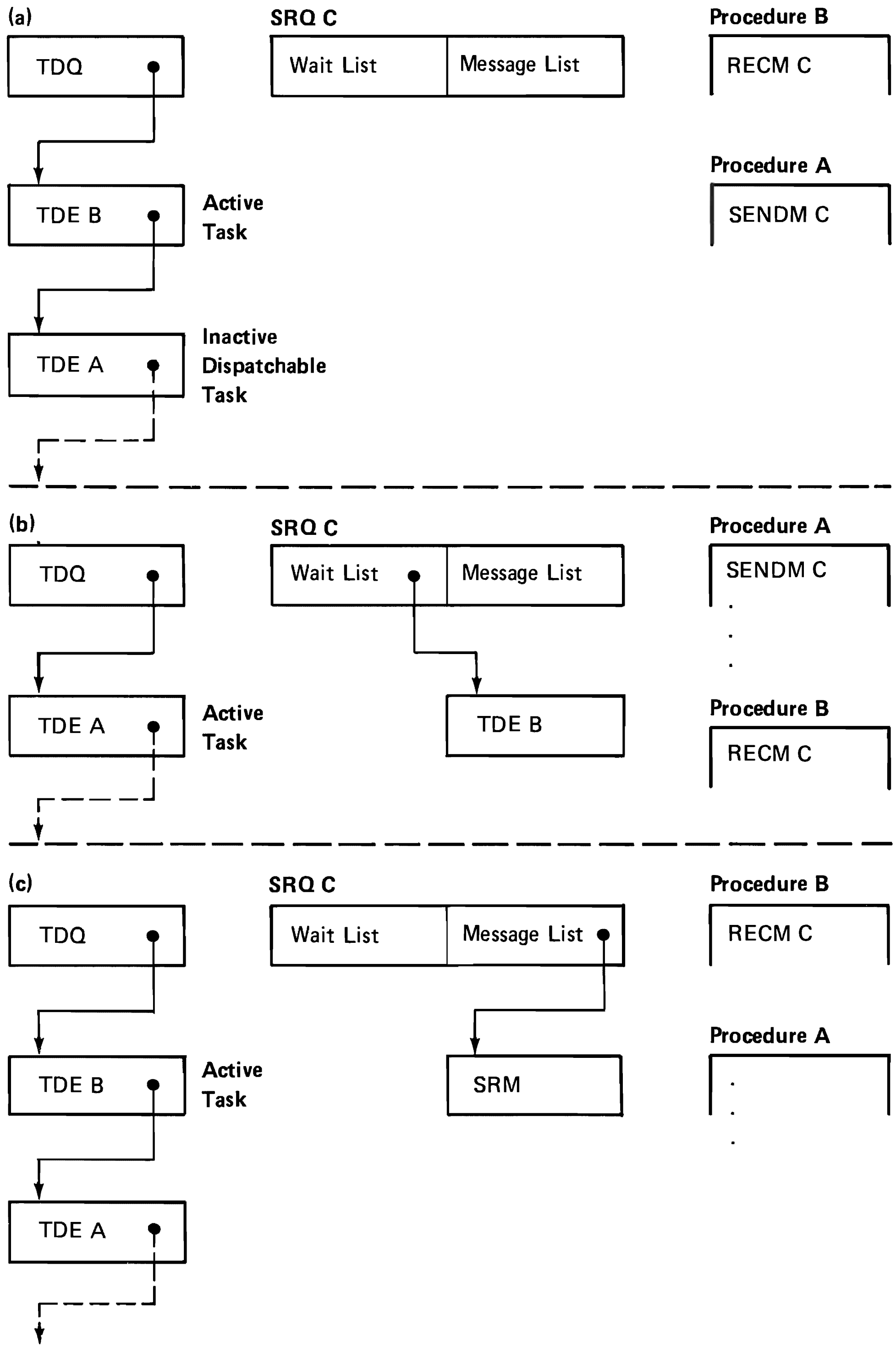


Figure 5-3. Task Synchronization Example

Chapter 6. Supervisor Linkage and Exception Presentation

This chapter describes IMP (internal microprogramming) supervisor linkage concepts and IMP exception presentation.

- *IMP Supervisor Linkage* is the method by which IMP procedure switching within the same task is accomplished. A supervisor linkage can be explicit or implicit and saves the status of the procedure from which the switch occurred.

- *IMP Exception Presentation* is the mechanism by which a defined set of exception conditions is presented, including the invocation of the IMP exception handling procedure.

Supervisor Linkage

The IMP extended program linkage facility calls an SVL (supervisor linkage) routine to perform one of the following functions:

- An extended IMP operation whose entry point is not addressed directly (explicit SVL).

- Simulation of an IMP instruction that has been trapped by the IMP interpreter (implicit SVL).

- Handling of a processor exception (exception SVL).

For all three functions, the IMP routine performing the function returns via an explicit Supervisor Exit instruction.

The basic services provided by the SVL mechanism are selective IMP procedure status saving/restoring and entry point resolution via a specialized SVL table.

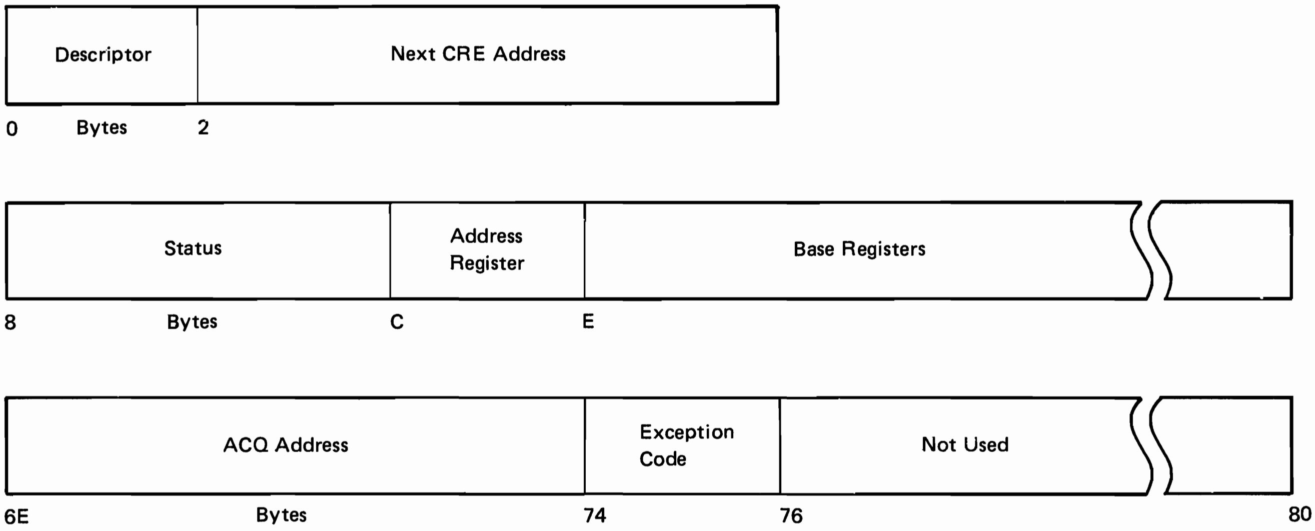
SUPERVISOR LINKAGE STRUCTURES

Three structures are used to control the supervisor linkage operation: The CRE (call/return element), the ACQ (available CRE queue), and the SVL (supervisor linkage) table. The ACQ address and the SVL table address are contained in the control address table (see Figure 2-2).

Call/Return Element

A CRE (call/return element) is a resident storage area used to save the status of a procedure during an SVL (supervisor linkage). If the CRE is not resident, is not fullword aligned, or crosses a page boundary, a machine check occurs when the CRE is accessed.

The descriptors of the CRE and the SRM (send/receive message) are identical. The key is not significant since all queuing functions are first on the chain. A CRE has the following format:



Bytes (Hex)	Bits	Description
0-1		**Descriptor:** Element descriptor (same as SRM).
	0-2	Identifies this IMP object as an SRM (011).
	3	Reserved.
	4	0 This is the last CRE on the chain. 1 This is *not* the last CRE on the chain.
0-1	5	0 This CRE is *not* enqueued in a chained list. 1 This CRE is enqueued in a chained list (either an SRQ or TDE). This bit is set to zero by those instructions that dequeue an SRM from an SRQ header. Those instructions that enqueue an SRM check this bit first. If it is a one bit, a specification exception is raised.
	6-7	Reserved.
	8-15	Used by the processor to monitor accesses to this object. If not hex 00, the CRE is monitored for access exceptions.

Bytes (Hex)	Bits	Description
2-7		**Next CRE Address:** Address of the next CRE in the chain. If this is the last CRE in the chain, descriptor bit 4 = 0.
8-B		**Status:** CRE Status.
	0	0 Available. 1 In use.
	1	0 IMP procedure CRE. 1 HMC procedure CRE.
	2-7	Reserved.
	8-11	First base register saved.
	12-15	Number of base registers saved minus one; must include base register 0 (base register addresses wrap around from hex F to 0).
	16-20	Zero.
	21-23	ILC (instruction length count).
	24-27	Zero.
	28-31	Condition code. When initializing the condition code field in a new CRE, at least one, but not all, of the bits must be set to a one value. Failure to do so may cause branch instructions to work incorrectly.
C-D		**Address Register:** IAR (instruction address register) if byte 8, bit 1 = 0. CSAR (control store address register) if byte 8, bit 1 = 1.
E-6D		**Base Registers:** Saved by the SVL mechanism; if all registers are not saved, the unused area is available to the SVL routine as a scratch work area. The registers occupy 6 bytes per register beginning with byte hex E.
6E-73		**ACQ Address:** Address of ACQ used by the SVX operation.
74 (Note 1)		**Exception Code:**
	0-2	Reserved.
	3	Soft address compare.
	4	Task interval timer.
	5	Monitored ACQ descriptor (SVL receive) (Note 2).
	6	Monitored CRE descriptor (SVL receive) (Note 2).
	7	Monitored TDE descriptor (SVL wait) (Note 2).

Bytes (Hex)	Bits	Description
75 (Note 1)		
	0-7	**Exception Code: (Hex)**
		00 No exception in bits 0-7.
		02 Invalid descriptor.
		04 Busy (Note 2).
		06 Reserved.
		08 Allocate Page Frame.
		0A Monitored SRQ descriptor (Note 2).
		0C Monitored SRM descriptor (Note 2).
		0E Monitored TDE descriptor (Note 2).
		10 SRC (send/receive counter) overflow.
		12 Address translation.
		14 Program event monitoring.
		16 Execute.
		18 Specification.
		1A Addressing.
		1C Effective address overflow.
		1E Data.
		20 Binary overflow.
		22 Binary divide.
		24 Decimal overflow.
		26 Decimal zero divide.
		28 Floating-point overflow.
		2A Floating-point underflow.
		2C Floating-point inexact result.
		2E Floating-point zero divide.
75		**Exception Code: (Hex)**
		30 Operation.
		32 Stack.
		34 Verify.
		36 Chain conflict.
		38 End-of-chain.
		3A Edit digit count.
		3C Length conformance.
		3E Edit mask syntax.
		40 Invalid segment group address.
		42 Floating-point invalid operand.
		46 Second chain search.
		48 Conversion.
		4A Invalid floating-point conversion.
		80 Invalid segment.[1]
		81 Invalid page.[1]
		82 Page read error.[1]
		83 Invalid pool state.[1]
		84 Invalid pin request.[1]
		85 Invalid write request.[1]
		86 Main store error.[1]
76-7F		**Not used.**

Notes:

1. The exceptions indicated in byte hex 74 can occur simultaneously and are not mutually exclusive with themselves or with an exception encoded in byte hex 75.
2. Exception code bits 5 through 7 in byte hex 74 and exception codes hex 04, 0A, 0C, and 0E in byte hex 75 form the general category of descriptor access exceptions described under *Descriptor Access Exception*, later in this chapter.

[1] Implicit SVL codes. For description, see Appendix B.

Available Call/Return Element Queue

The ACQ (available CRE queue) is the mechanism by which CREs (call/return elements) are made available to the processor and eventually to a TDE (task dispatching element).

An ACQ has the following format:

Descriptor	First TDE Address	Reserved	Key Lth-1
0 Bytes	2	8	9

First Available CRE Address	Reserved	
A Bytes	10	20

A CRE is taken from the ACQ by an implicit receive operation, when needed, to perform an SVL (supervisor linkage) instruction (explicit or implicit). A CRE is returned by an implicit send to the ACQ when the SVX (supervisor exit) instruction is issued. The descriptors of the ACQ and the SRQ (send/receive queue) are identical (if no available CREs are enqueued, descriptor bit 4 = 0). See the SRQ in Chapter 5 for the byte descriptions. All send/receive operations involving the ACQ must specify enqueue first/dequeue first.

If the ACQ is not resident in storage, is not fullword aligned, or crosses a page boundary, a machine check will occur when the ACQ is accessed.

Supervisor Linkage Table

The SVL (supervisor linkage) table is located in resident storage and consists of 256 4-byte entries. The format of each 4-byte entry in the SVL table is as follows:

Base	Flag	Entry Address	
0	1	2 Bytes	4

Bytes (Hex)	Bits	Descriptions
0		**Base:** Base registers to be saved.
	0-3	First base register saved.
	4-7	Number of base registers to be saved minus 1. The registers saved must include B(0). Base register addresses wrap around from hex F to zero.
1		**Flag:** Flag byte.
	0	0 IMP offset in bytes 2 and 3. 1 HMC CSAR in bytes 2 and 3.
	1	Reserved.
	2	0 If not an exception SVL (SVL table index = 0), then the SVL procedure will be inhibited and the SVL instruction will be executed as a no-operation. The exception SVL (SVL table index = 0) will always be executed regardless of the value of this bit. 1 The SVL procedure will be executed as described.
	3-7	Reserved.
2-3		**Entry Address:** Entry address of SVL routine. Bytes 2 and 3 are either an offset into the segment identifier of the SVL table if Byte 1, bit 0 = 0, or a CSAR value if byte 1, bit 0 = 1.

All SVLs refer to entries in this table.

SUPERVISOR LINKAGE CONTROL

The SVL (supervisor linkage) operation allows:

- Program to program invocation without explicit knowledge of program locations by the invoking program.

- A single, common interface for exception signaling.

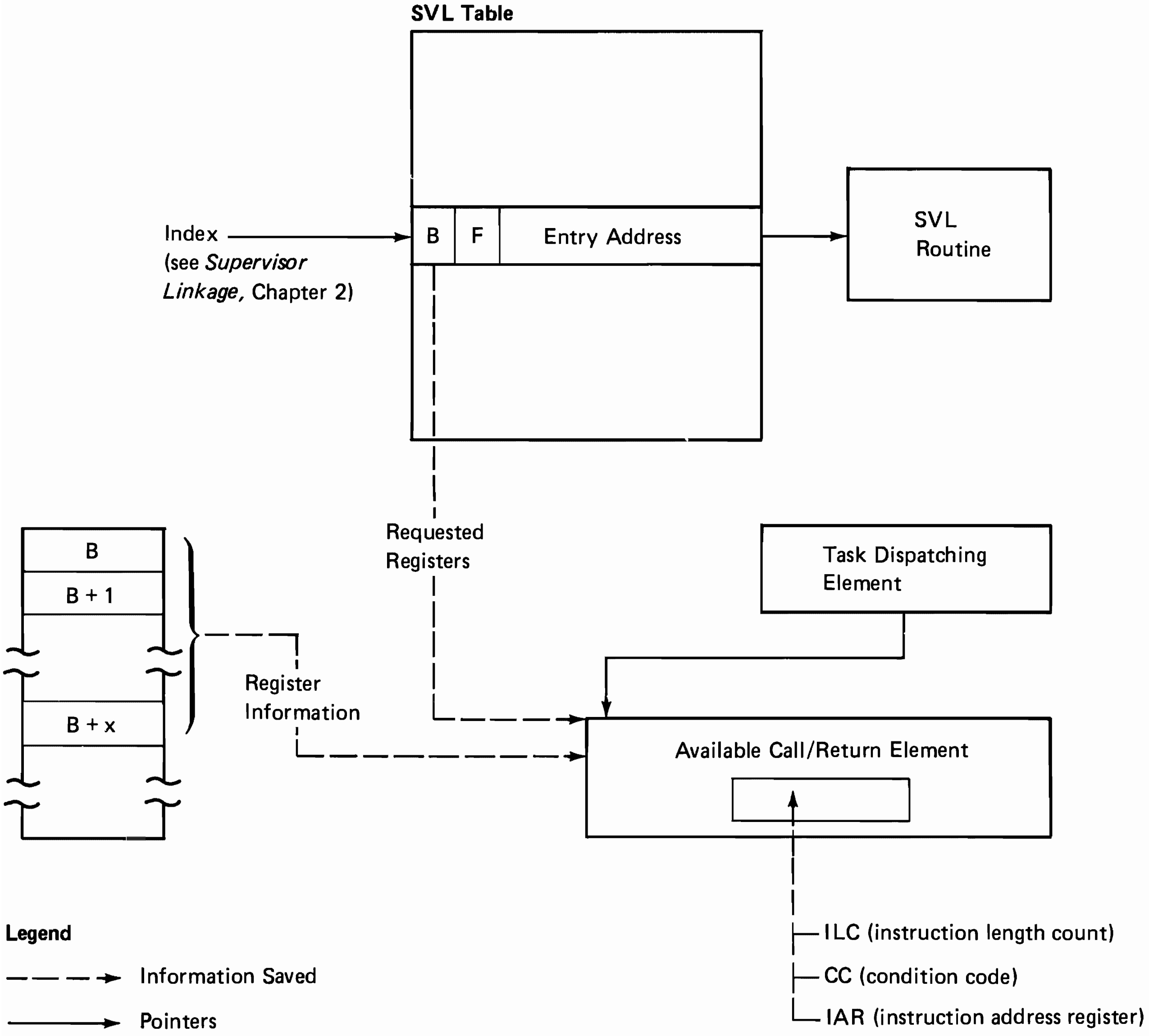
The SVL operation can be understood by considering the usage of CREs (call/return elements) for status saving and the SVL table for indirect SVL routine entry point resolution (see the following diagram). An index into the SVL table can be generated either explicitly or implicitly, based on the cause of the SVL. Implicit SVLs and, therefore, implicit entries into the SVL table, are generated by the processor for either trapped instructions (using the trapped instruction operation code), or for exceptions. Explicitly generated SVL instructions use the I-byte of the SVL to index the SVL table. The assignment of SVL indexes (binary values) is as follows:

- All exceptions use SVL table index 0.

- Other implicit SVLs use indexes that do not correspond to the basic operation codes or to the unused extension fields in those instructions that make use of operation code extenders. However, operation codes hex 00, 40, and FF are reserved and are treated as invalid; operation code 0D extended with E or F is reserved and will yield unpredictable results if executed. An operation exception results if execution of one of these operation codes is attempted. With this exception, the execution of any operation code that is not implemented as a basic operation code results in an implicit SVL operation. In this case, the SVL table index value equals the operation code.

- Explicit SVLs can use any index value.

SVL Table and CRE Usage



The occurrence of an exception results in an SVL unless the exception is masked in the TDE (task dispatching element) mask field. If the exception is not masked, the exception code is stored in the exception field of the CRE and the pending exception is cleared in the processor. If the exception is masked, the occurrence is recorded in bytes hex 1C and 1D of the TDE, no SVL occurs, and the pending exception is cleared.

When an SVL occurs either explicitly or implicitly, byte 0 of the SVL table entry and the status of the procedure are saved in the available CRE. The status includes the following:

- Instruction length count
- Condition code
- Specified base registers
- Exception code
- IAR (instruction address register)
- CSAR (control store address register)

For an exception SVL in an IMP procedure, the stored IAR points to the instruction that caused the exception if the instruction was nullified or suspended. Otherwise, the stored IAR is the updated address of the next instruction to be executed. The CRE is then flagged as being in-use and the address of the CRE is loaded into base register hex E. In addition, as shown in the following table, optional registers are loaded with parameters, depending on whether the SVL is implicit or explicit and whether zero, one, or two operands are present in the SVL instruction.

SVL Register Loading

	Parameters Loaded	
Implicit Type SVL	**Register**	**Parameter**
RR (2 bytes)	r(E)	I-byte[1]

RS, SI (4 bytes, operation codes ≥ hex 40, <hex A0) r(E)	r(E)
	B(1)
		I-byte[1] First- or second-operand address

SS (6 bytes, operation codes ≥ hex A0)	r(E)	I-byte[1]
	B(1)	First-operand address
	B(2)	Second operand address

[1] In the case of an instruction with an extended operation code, the low order 4 bits of the I field contain the operation code extender.

	Parameters Loaded	
Explicit Type SVL	**Register**	**Parameter**
SVL0[1, 2]	None	
SVL1[1, 2]	B(1)	First-operand address
SVL2[1, 2]	B(1)	First-operand address
	B(2)	Second operand address
Address translation exception	B(1)	Faulting virtual address
Allocate page frame exception	B(1)	Virtual address
All other exceptions	None	

[1] B (E) is loaded with the address of the CRE used to save status for all SVL types.
[2] See Chapter 10 for the format of the SVL instruction types.

Effective addresses are calculated and loaded into base registers for those SVLs having effective address operands. HMC or IMP instruction processing continues at the address indicated in the SVL table entry. If bit 0 of the SVL table flag byte is one, the SVL routine address in the table entry is loaded into the CSAR. This causes an SVL microprogram indirect branch. If bit 0 of the flag is zero, the halfword address in the SVL table entry is an offset into the SID (segment identifier) where the SVL table is located. This real address then becomes the target of an indirect SVL branch to an IMP procedure (register S[0] contains the SID of the SVL table, register R[0] contains the SVL entry address, and the IAR contains the value that was placed in R[0]).

When an SVL is executed, a search of the CRE list chained to the TDE is performed. The current status is stored in the last available CRE. If there are no available CREs on the list, or if the list is empty, a CRE is implicitly received from the ACQ (available CRE queue) and is enqueued first, when a CRE becomes available, on the TDE CRE list. The current status is then stored in that CRE. In either case, the status of the CRE is set to in-use.

SVL access exceptions (monitored ACQ, CRE, and TDE), associated with the implicit receive operation within an SVL, are detected and presented as follows:

1. Busy is ignored and not presented.

2. If the ACQ and CRE are monitored, these exceptions are presented, after completion of the implicit receive, in byte 0 of the exception code in the TDE.

3. Or, if the implicit receive is not satisfied (no CREs on the ACQ) and the current TDE is monitored, this access exception is reported in byte 0 of the exception code in the TDE.

Programming Note: When replenishing the ACQ with CREs, you should specify send message first in order to place the CRE first on the ACQ (busy is ignored by an SVL implicit receive).

For trapped operations (implicit SVLs) and explicit SVLs, the original SVL function is nullified (the IAR still points to the SVL of the trapped operation code) and the ACQ access exception is identified in byte 0, bits 5-7 of the exception code. If an SVL access exception is detected while presenting an exception, the access exception is presented concurrently with other exceptions posted in byte 1, bits 0-7 of the exception code in the CRE. Bit 6 or 7 of the CRE exception code, byte 0, can be presented only after a CRE is received. While the task dispatcher is disabled, due to a machine check, the function of the exception SVL is altered (as described in Chapter 9, under *Machine Check*).

If there are no CREs on the ACQ, the implicit receive is not satisfied, the TDE for the current task is moved to the wait list of the ACQ, and the task dispatcher is subsequently invoked. For trapped operations and explicit SVLs, the original SVL function is nullified. For all exceptions except page fault, the exception code is saved in the TDE as part of the task switch. The exception is presented again after the ACQ is replenished and the task is dispatched. For page faults, the exception code and the faulting address are discarded. Because the instruction or HMC function causing the fault is nullified or marked by a checkpoint, the exception is regenerated when the task becomes dispatchable and the operation is again performed. If an HMC procedure causes an access exception, the queue function, the queue header address, and optionally, the message or TDE address are saved in the base register field of the CRE. The register assignments for the values are documented in Chapter 7, under *Operational Unit Task*. The same IMP exception handler is invoked for HMC exceptions as for IMP exceptions. The second byte of the exception code always contains a value from hex 00 to 12 or to 1C for HMC procedure exceptions.

This page is intentionally left blank.

An explicit SVX (supervisor exit) instruction is used in all cases to return from an IMP routine called via the SVL instruction. During execution of the SVX instruction, the condition code, IAR, or CSAR and base registers are restored from the first in-use CRE on the current TDE and the CRE status is set to available. The exception code and ILC (instruction length count) are not restored from the CRE. If the number of available CREs encountered before this CRE was equal to or greater than the number specified in the control mode field of the TDE, the first CRE is returned to the ACQ via an implicit send. The CRE is returned to the proper ACQ using the ACQ address stored in the CRE. This allows gathering of CREs added to the ACQ due to an earlier ACQ access exception. The in-use CRE is then used to restore status and is flagged available. Note that when an SVX is executed, if no in-use CRE is found, or if the CRE list is empty, a specification exception occurs. Also, descriptor access exceptions are not detected by the implicit send when a CRE is returned to the ACQ.

Programming Note: A minimum of one available CRE is always left chained to the TDE by the SVX instruction, even if the maximum number is set to zero.

SUPERVISOR LINKAGE SUMMARY

Figure 6-1 and the following text summarize an SVL operation. Assume that the maximum number of available CREs specified in the control mode of the TDE is one.

1 Procedure Z is being executed and the condition of the TDE and the ACQ are as shown below the procedure in the figure.

2 A condition arises (explicit or implicit) in procedure Z requiring an SVL to procedure X. At the time of the SVL, since CRE A is in use, there are no available CREs on the TDE to store the status and base registers of procedure Z. Therefore, a CRE (CRE B) is obtained from the ACQ and is enqueued to the TDE. The status of procedure Z is then saved in CRE B.

3 Again an SVL occurs; this time to procedure Y. CRE C is obtained from the ACQ, the status of procedure X is stored in CRE C, and the execution of Y begins.

4 Procedure Y is completed and an SVX instruction issued. The SVX restores the status of procedure X flags, CRE C as available, and restarts the execution of procedure X.

5 Procedure X is completed and an SVX instruction issued. Because an available CRE exists on the TDE, the top CRE (CRE C) is returned to the ACQ. The status of procedure Z is then restored from CRE B, CRE B is flagged available, and the execution of procedure Z is restarted.

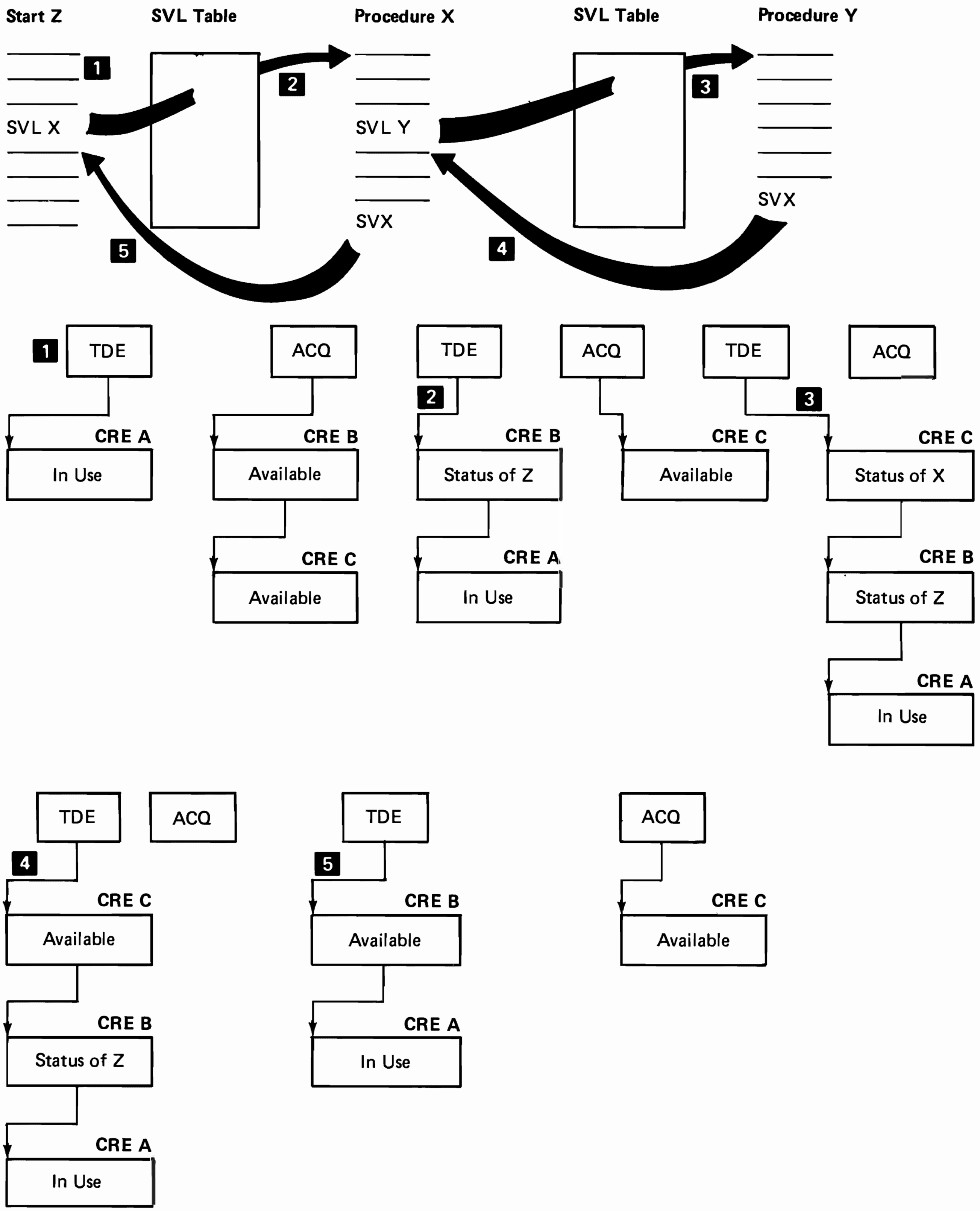


Figure 6-1. SVL Summary

Exceptions

PRESENTATION

Exceptions can occur during the execution of an IMP task. Causes of exceptions include the improper specification or use of instructions and data, the detection of a page fault, the detection of a program event, and task interval timer. Because an exception is the direct result of the current task, as opposed to some external event, the resolution of the exception is handled under control of the current task. The function of the exception SVL during the handling of a machine check is described in Chapter 9 under *Machine Check*.

Exceptions are presented through the use of the 2-byte exception code area in the CRE (call/return element). The two types of exceptions, concurrent and program, and the CRE bytes in which they occur are described in the following paragraphs.

CONCURRENT EXCEPTIONS

Concurrent exception conditions are presented in the first byte of the CRE exception code field. These exception conditions are bit significant and can occur simultaneously.

PROGRAM EXCEPTIONS

Exceptions that result from the execution of IMP instructions in an IMP procedure or HMC instructions in an HMC procedure are called program exceptions. These exceptions include the improper specification or use of instructions and data, address translation faults, and detection of program events.

The cause of an exception is identified in the exception code field of a CRE (call/return element). The bit assignments for this field are described as a part of *Call/Return Element*, earlier in this chapter.

The exception mask field in the TDE (task dispatching element) allows some exceptions to be masked. A program exception can only occur when the corresponding mask bit is 1. When the mask bit is 0, the occurrence of the condition is recorded in the exception occurrence field of the TDE but no program exception occurs.

The following paragraphs describe each type of program exception.

Address Compare Exception

A programmable address compare exception occurs when:

- An address compare for the address and type of compare (instruction, I/O, or data) is detected.

For instruction stream address compare, the instruction is nullified. For other types of address compares (data, I/O, or other), the instruction or current unit of operation is completed. See the SACM instruction in Chapter 10 for additional information.

Address Translation Exception

An address translation exception (or page fault exception) occurs when the processor is unable to translate a virtual address into a real address using the VAT (virtual address translator) facilities described in Chapter 8 because:

- No primary directory entry exists for the page in the primary directory.

- A primary directory exists for the page in the primary directory and the valid status bit is set to zero.

- The index field is zero in the hash table entry.

The instruction is nullified except for GHRF, GHR, FHRF, and FHR instructions. For these instructions the instruction is completed.

Programming Note: When the exception is presented, base register 1 contains the faulting address.

Addressing Exception

An addressing exception is recognized when:

- A virtual = real address SID (segment identifier) is used that refers tc a storage location that is beyond the range of real storage configured to the processor. Such an address is also invalid.

Model	V=R Address SID Range (Hex)
3, 4, and 5	0000 0100 - 0000 011F
6, 7, and 8	0000 0100 - 0000 01FF

The operation is suppressed when the address of the instruction is an invalid address. The operation is terminated for an invalid operand address.

Allocate Page Frame Exception

An allocate page frame exception occurs when:

- An OU task requests a page frame to be allocated and cleared in main storage.

The instruction is nullified.

Programming Note: When the exception is presented, base register 1 contains the virtual address to be associated with the allocated page frame.

Binary Divide Exception

A binary divide exception occurs when:

- The size of the quotient exceeds the size of the resultant field in a binary divide operation.
- Division by zero is attempted during a binary divide operation.

The instruction is suppressed.

Binary Overflow Exception

A binary overflow exception occurs when:

- The carry from the sign-bit position and the carry from the high-order numeric bit position do not agree during a signed binary add, subtract, or zero and add operation.
- The results of a Convert Packed to Binary instruction exceeds 31 bits.
- Significant bits are lost during a Shift Left Arithmetic instruction.

The instruction is completed.

Chain Conflict Exception

A chain conflict exception occurs when:

- A hold conflict is found on an object hold operation.
- An object free operation attempts to free a monitored hold.

The instruction is nullified but the first-operand base register is updated to point at the offending hold record.

Conversion Exception

A conversion exception occurs when:

- The length field of a string control byte is 0 for a CVTMC instruction.
- The end of source is encountered prior to the end of a compression string for a CVTSC instruction.
- A compression string describes a character string that would cross a record boundary in the receiver for a CVTSC instruction.
- The length field of a string control byte is 0 for a CVTSC instruction.

Data Exception

A data exception occurs when:

- The sign or digit codes of operands in the decimal instructions or in a Convert Packed to Binary instructions are invalid.

- The multiplicand in a Multiply Packed instruction has an insufficient number of leftmost zeros.

The instruction is suppressed when a sign code is invalid; otherwise, the instruction is terminated.

Decimal Overflow Exception

A decimal overflow exception occurs when:

- One or more significant digits are lost because the destination field in a decimal operation is too small to contain the result.

The instruction is completed.

Decimal Zero Divide Exception

A decimal zero divide exception occurs when:

- Division by zero is attempted by a Divide Packed instruction.

The instruction is suppressed.

Descriptor Access Exceptions

Descriptor access exceptions occur as:

- Descriptor access busy (SRQ is in use)

- Monitored (nonzero byte 1) in an SRQ (send/receive queue) descriptor

- Monitored (nonzero byte 1) in an SRM (send/receive message) descriptor

- Monitored (nonzero byte 1) in a TDE (task dispatching element) descriptor

A descriptor access busy exception occurs when bit 5 of descriptor byte 0 is a one during a reference to an SRQ. A descriptor access busy exception also occurs during object hold/free operations if the hold record chain of the object is found to be busy. This bit indicates when an access to the object is in progress. The other three exceptions (monitored) occur during a reference to an IMP object whose descriptor byte 1 does not contain all zeros. The particular exception that occurs depends on the type of object being referred to. When the processor encounters a monitored (nonzero byte 1) SRQ, SRM, or TDE while executing an instruction, an access exception occurs. The instruction causing the exception is then suspended, the SRQ is set busy, and the checkpoint information is stored in the reserved area of the header before the exception is presented. On return from the exception handler, checkpoint information is restored, busy is reset, and the normal instruction execution begins by resuming the suspended instruction. For the Receive Message and Dequeue Message instructions on an SRM access exception, the element is dequeued before the exception is taken. For Send Message and Enqueue Message instructions the exception is taken before the element is enqueued.

For Enqueue and Dequeue TDE instructions with a TDQ (task dispatching queue) or SRC (send/receive counter) as the target, a TDE descriptor access exception does not occur.

The following chart shows the instructions for which access exceptions can occur and the sequence (numbers 1, 2, and 3) of occurrence for each IMP instruction and IMP object.

		Sequence	
Instructions	**SRQ[4]**	**SRM**	**TDE**
DQM	1	2[5]	
DQTDE	1		2[1, 5]
EQM	1	2	
EQTDE	1		2[1]
SENDM	1	2	3
SENDMW	1	2	3
RECM	1	2[5]	2
SVL0[2]	1	2	2
SVL1[2]	1	2	2
SVL2[2]	1	2	2
SVX[2]			

A TDQ or SRC cannot be set busy or monitored. If descriptor bit 5 (busy) or byte 1 (monitored) are nonzero, the condition is ignored by the processor.

The instruction is nullified for SRQ busy and hold free chain busy. It is suspended for monitored descriptors of SRQ and SRM.

Note: If the exceptions occur while an SVL instruction is being serviced, the exception is reported in byte hex 74 of the CRE. The instruction causing the exception is suspended.

Edit Digit Count Exception

An edit count exception is recognized in EDPD when:

- The end-of-source field is reached and there are more control characters corresponding to digits in the edit-mask field than in the source field.

- The end of the edit-mask field is reached and there are more digit positions in the source field.

The instruction is terminated.

Edit Mask Syntax Exception

An edit mask syntax exception occurs when:

- An invalid control character is in the EDPD mask.

- An end-of-string character is missing.

The instruction is terminated.

Effective Address Overflow Exception

An effective address overflow exception may occur when a carry from the offset portion of a virtual address occurs during the calculation of a storage operand address or a branch address.

An effective address overflow exception occurs when:

- A carry from bit 24 of a virtual address occurs during the calculation of a storage operand address.

- A storage operand crosses a segment boundary.

The instruction is suppressed.

[1] If the second operand is a TDQ or SRC, no descriptor access exception occurs.

[2] Descriptor access exceptions are not detected by the implicit send when a CRE is returned to the ACQ.

[3] The SRM and TDE exceptions are mutually exclusive for a Receive Message instruction and an SVL if an implicit receive is necessary.

[4] The SRQ descriptor byte 0, bit 6 is a summary indicator for any TDEs that have monitor bits set and are enqueued to the wait list. This bit is not maintained by the processor, but is used to test for a TDE access exception on send type operations. Therefore, the IMP exception handler is responsible for appropriately setting and resetting the bit within the exception routine.

[5] The SRM or TDE access exception is taken after the SRM has been dequeued.

End-of-Chain Exception

An end-of-chain exception occurs when:

- An empty (null) chain is found on a free operation.
- An end-of-available (hold record) chain is found on a hold operation.
- No matching hold record is found on a free operation.

The instruction is nullified with the first operand unchanged.

Execute Exception

An execute exception occurs when:

- The subject of an Execute instruction is another Execute instruction.

The instruction is suppressed.

Invalid Descriptor Exception

An invalid descriptor exception occurs when:

- An invalid descriptor field is encountered during the execution of an operation on an IMP object.

Whether or not a descriptor type is valid depends on the operation being performed. The following chart summarizes operations on IMP objects.

	Header Type			**Element Type**	
Instruction	**SRQ**	**TDQ**	**SRC**	**SRM**	**TDE**
DQM	V	I	I	N	N
DQTDE	V	V	V	N	N
DTDQ	N	N	N	-	-
EQM	V	I	I	V	I
EQTDE	V	V	V	I	V
RECM	V	I	I	N	N
RECC	I	I	V	-	-
SENDM	V	I	I	V	I
SENDMW	V	I	I	V	I
SENDC	I	I	V	-	-

Legend:
V = Valid
I = Invalid descriptor exception
N = Descriptor not checked
- = No element involved

Note: An invalid ACQ descriptor encountered by an SVL implicit receive or an SVX implicit send causes a machine check. An invalid SRM descriptor encountered by an SVX implicit send causes a machine check. An invalid SRC descriptor encountered by an I/O event SENDC causes a machine check.

The instruction is terminated.

Floating-Point Inexact Result Exception

A floating-point inexact result exception occurs if the rounded result of an operation is not exact. The result is inexact because:

- One or more bits have been lost in the rounding process.
- A floating-point overflow occurred while the overflow was masked, and the result has been set either to infinity or to the largest finite number for that specific format.

The setting of the floating-point inexact result mask does not affect the result of the operation. The rounded or overflowed result is still available in the result operand.

Floating-Point Invalid Operand Exception

A floating-point invalid operand exception occurs when an operand is invalid for the operation to be performed. The operand is invalid because:

- An operand is an unmasked not-a-number.
- Addition or subtraction of infinity with infinity was attempted.
- Multiplication of zero times infinity was attempted.
- Division of zero by zero, or division of infinity by infinity was attempted.

The setting of the floating-point invalid operand mask affects the result of the operation.

- If the exception is masked, the result of the operation is a masked not-a-number value:
  - If the exception was because of one or more operands being an unmasked not-a-number, then the resulting masked not-a-number value is set with a fraction value equal to the largest not-a-number operand fraction value.
  - If the exception was not because of an operand being an unmasked not-a-number, then the resulting masked not-a-number value is set with a fraction value consisting of a 1 in the leftmost bit position followed by zeros for the remaining fraction bits.
- If the exception is not masked, the operation is suppressed, and the exception is signaled.

Floating-Point Overflow Exception

A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. For this exception to occur, the exponent must exceed 127 in the short format and 1023 in the long format. The setting of the floating-point overflow mask affects the result of the operation. In addition, the result of the operation depends on the rounding mode and the sign of the ntermediate result, as follows:

		Rounding Mode–Toward			
Overflow Exception Is:	**Sign of Intermediate Result Is:**	**Zero**	**Positive Infinity**	**Negative Infinity**	**Nearest**
Masked	Positive	To largest positive number	To infinity correctly signed	To largest positive number	To infinity correctly signed
	Negative	To largest negative number	To largest negative number	To infinity correctly signed	To infinity correctly signed

Overflow Exception Is:	**Sign of Intermediate Result Is:**	**Significant**	**Sign**	**Exponent**
Unmasked	Positive or negative	Correctly rounded	Correct	Modified (see note)

Note: The modified exponent is set from the overflowed normal biased exponent minus a bias adjust of 192 for short format and 1536 for long format. The following summarizes the relationship between what would be the overflowed values for the true exponent signed exponent, the normal biased exponent, and the modified biased exponent.

	Short Format			Long Format		
Overflowed Exponent	**True Signed**	**Normal Biased**	**Modified Biased**	**True Signed**	**Normal Biased**	**Modified Biased**
Minimum	128	255	63	1024	2047	511
Maximum	255	382	190	2047	3070	1534

Floating-Point Underflow Exception

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. For this exception to occur, the exponent must be less than -126 in the short format and -1022 in the long format. The setting of the floating-point underflow mask affects the result of the operation and the setting of the occurrence bit.

- If the exception is masked, the result of the operation is produced by first denormalizing the unrounded result, then rounding, then putting it in its result field. Only when the result is not exact is the underflow exception occurrence bit set on.

- If the exception is not masked, the result of the operation is a correctly rounded significant, a correct sign, and a modified exponent. The underflow exception occurrence bit is set on. The modified exponent is set from the underflowed normal biased exponent plus a bias adjust of 192 for short format and 1536 for long format. This bias adjust is chosen to translate underflowed biased exponents as nearly as possible to the middle of the representable biased exponent range for the short and long formats. This allows the exception handler to provide appropriate information for later reconstruction of the correct result. The following diagram summarizes the relationship between what would be the underflowed values for the true signed exponent, the normal biased exponent, and the modified biased exponent.

	Short Format			Long Format		
Underflowed Exponent	**True Signed**	**Normal Biased**	**Modified Biased**	**True Signed**	**Normal Biased**	**Modified Biased**
Minimum	-126	1	193	-1022	1	1537
Maximum	-298	-171	21	-2148	-1125	411

Floating-Point Zero Divide Exception

A floating-point zero divide exception occurs if the divisor is 0 and the dividend is a finite nonzero number. If the exception is masked, the result of the operation is a correctly signed infinite value. If the exception is not masked, the operation is suppressed.

Invalid Floating-Point Conversion

An invalid floating-point conversion exception occurs during conversion from floating-point to a fixed-point format when overflow, infinity, or not-a-number precludes an accurate representation in the fixed point format. This exception cannot be masked and has no corresponding occurrence bit. The instruction is suppressed.

Invalid Page Exception (Synchronous Requests Only)

An invalid page exception occurs when:

- The page does not exist in the segment. The segment group exists, but has these properties: (1) the size allocated is less than 16 megabytes and (2) a reference was made to an address which would have been legitimate had the segment been made larger. This exception is raised only on Bring and Clear (VMC function) and on page faults.

The instruction is suppressed.

Invalid Pin Request Exception (Synchronous Requests Only)

An invalid pin request exception occurs when:

- Pin failed because an attempted pin was the two hundred fifty-sixth pin for that page.

- An unpin was attempted on an unpinned page.

The instruction is suppressed.

Invalid Pool State Exception (Synchronous Requests Only)

An invalid pool state exception occurs when:

- A Bring or Clear (VMC function) with pin cannot be performed because too many pages are already pinned.

The instruction is suppressed.

Invalid Segment Exception (Synchronous Requests Only)

An invalid segment exception occurs when:

- The page does not exist on a Bring and Clear (VMC function) or page fault. The requested segment either never existed or has been destroyed.

The instruction is suppressed.

Invalid Segment Group Address Exception

An invalid segment group exception occurs when:

- The leftmost 3 bytes of the 6-byte virtual address are invalid for a BI (Branch Internal) instruction.

- The calculated low-order 3-byte segment group address offset is not a positive value or is not between a designated lower boundary and 16 megabytes minus 1 inclusive, for a CAL, CALH, or CALHI instruction.

- An overflow is generated in the intermediate or final calculation of an instruction which performs 3-byte address arithmetic.

The instruction is suppressed.

Invalid Write Request Exception (Synchronous Requests Only)

An invalid write request exception occurs when:

- A write was requested to a pinned page.

The instruction is suppressed.

Length Conformance Exception

A length conformance exception occurs in EDPD when:

- The end of the edit-mask field is reached and there are more character positions in the result field.
- The end of the result field is reached and there are more character positions in the edit-mask field.
- The number of hex B2 control characters following a hex B1 (floating string) field cannot accommodate the longer of the two floating strings.

A length conformance exception occurs in a CVTMC instruction when the converted form of the record is larger than the result record length.

In either case, the instruction is terminated.

Main Storage Error Exception

A main storage error exception occurs when:

- Changed data in main storage could not be accessed due to a memory failure. This exception initiates the reporting of logical damage.

The corresponding page on disk is marked logically bad, and the instruction is terminated.

Monitored ACQ Exception

A monitored ACQ (available CRE queue) exception occurs when:

- An implicit receive operation attempts to take a CRE from the ACQ when the ACQ wait list is empty.

The instruction is nullified.

Monitored Call/Return Element Exception

A monitored call/return element exception occurs when:

- A CRE is accessed during an SVL (supervisor linkage) and there are no CREs available.

The instruction is nullified.

Monitored SRM Descriptor Exception

See *Descriptor Access Exceptions* earlier in this chapter.

Monitored SRQ Descriptor Exception

See *Descriptor Access Exceptions* earlier in this chapter.

Monitored TDE Descriptor Exception

See *Descriptor Access Exceptions* earlier in this chapter.

Operation Exception

An operation exception occurs when:

- The execution of an instruction with an invalid operation code is attempted.

Operation codes of hex 00, 40, and FF are invalid. The instruction is suppressed.

Page Read Error Exception

A page read error exception occurs when:

- A bring (VMC function) or a page fault could not read a given page from the disk. This exception initiates the reporting of logical damage to the System/38 instruction set.

The instruction is terminated.

Program Event Monitoring Exception

A program event monitoring exception occurs when:

- Program event monitoring is specified and the designated event occurs.

The instruction is nullified.

See *Program Event monitoring* in Chapter 9 for a description of the exception condition.

Second Chain Search Exception

A second chain search exception occurs when:

- The Grant Hold or Free Hold instructions determine that a secondary chain must be searched.

The instruction is nullified.

Send/Receive Counter Overflow Exception

A send/receive counter overflow exception occurs when:

- A carry from the high-order position of the count field occurs during a send operation.

The exception cannot be masked and the operation is terminated. This exception does not occur for the implicit send count operations caused by the event or timer functions.

The instruction is terminated.

Specification Exception

A specification exception occurs when:

- An instruction address does not designate a location on an even-byte (halfword) boundary.
- An instruction stream crosses a segment boundary. The exception is recognized after the execution of an instruction that ends on, but does not cross a segment boundary.

  Programming Note: Because of a hardware restriction on Models 3, 4, and 5, the last fullword of a segment should not contain any portion of an instruction.
- An operand address does not designate an integral boundary in an operation requiring such integral boundary designation.
- A branch, call, or jump address does not designate a location on an even-byte (halfword) boundary.
- The multiplier or divisor in decimal arithmetic exceeds 15 digits and sign.
- The first-operand field is shorter than or equal to the second-operand field in decimal multiplication or division.
- No in-use CRE (call/return element) is on the TDE (task dispatching element) during an SVX (supervisor exit) operation.
- An IMP object used in any queuing operation or a key operand in a dequeue or receive type operation crosses a page boundary or is not fullword aligned.
- The second or third operand of a Compute Subscript Address instruction is zero.
- An enqueue or send type operation designates a TDE or SRM (send/receive message) that is currently enqueued (descriptor bit 5 = 1).
- Invalid I (immediate) field in dequeue, receive, or interval timer instructions.
- The source field of an EDPD (Edit Packed Decimal) instruction is greater than 31 digits.

- The current stack entry is not doubleword aligned when accessed by the FCN2 instruction.

- The current stack entry is too short for the FNC2 instruction.

- The address of the FRAT (function routine address table) is not halfword-aligned when accessed by the FNC2 instruction.

- The first operand address of a CALLI, STST, or LVT instruction does not start aligned on a 16-byte boundary.

- Both operand addresses of a MVAST instruction do not start aligned on a 16-byte boundary.

- The second-operand address of an AHSPOI, AHSPO, or AFSPO instruction does not start aligned on a 16-byte boundary.

- The address of the available hold record does not start aligned on a 16-byte boundary for the GHR or GHRF instructions.

- In an EPDE instruction the primary directory entry specified is for a V=R address.

- In an EPDE instruction the primary directory entry specified and its associated virtual address are not on the same hash chain.

- In an RPDE instruction the primary directory entry specified is not on the specified hash chain.

- The length is negative in a TRIM instruction.

- In an STSOP instruction a binary underflow was detected when the space locator was subtracted from the address contained in the first operand.

- In a CVTMC instruction the result record length is 0.

- In a CVTCM instruction the source record length is 0.

- A source or result field offset is specified beyond the end of the related source or receiver operand for a CVTCM, CVTMC, CVTCS, or CVTSC instruction.

- An algorithm modifier other than hex 00 or 01 is specified for a CVTCM or CVTMC instruction.

- The length of the first operand as specified in register R14, or the length of the second operand as specified in register R15 is 0 for a CVTCM, CVTMC, CVTSC, or CVTCS instruction.

- A single mode SCAN is specified and the mode control value in byte 0 of the control operand (operand 1) specifies different modes for the base string and scan character (bits 0 and 1 are not equal) for a SCAN instruction.

- Bits hex 2 through 7 of byte 0 (mode control field) of the control operand are not 0 for a SCAN instruction.

- A source record length of 0 is specified, and record processing is also specified for a CVTCS instruction.

- The value of the unconverted source record bytes parameter is greater than the source record length for a CVTCS instruction.

- Invalid values are specified in the algorithm modifier byte for a CVTCS or CVTSC instruction.

- A result field record length of 0 is specified, and record processing is also specified for a CVTSC instruction.

- The value of the unconverted result field record bytes parameter is greater than the result field record length for a CVTSC instruction.

- The algorithm modifier bit 2 = 0 (no transparent data in source), and transparency conversion status is active for a CVTSC instruction.

- The field length of a CVSFB or CVLFB instruction is invalid.

The instruction is suppressed.

A specification exception condition associated with (1) an SVL (supervisor linkage) implicit receive for a CRE, (2) and SVX implicit send of a CRE, (3) a timer event SENDC (send count), or (4) an I/O event SENDC causes a machine check.

A specification exception occurs when the Enqueue Task Dispatching Element instruction references the TDQ (task dispatching queue) or a TDE that is misaligned or crosses a page boundary or when the Dequeue Task Dispatching Element instruction references a similarily invalid TDQ. All other references to an invalid (misaligned or page boundary crossing) TDQ or TDE by the processor result in a machine check.

Implicit SVL instructions are indicated in Appendix B.

Stack Exception

A stack exception occurs when:

- A stack operation attempts to add a stack entry which extends beyond the stack limit value.
- An unstack operation attempts to remove the first stack entry within the segment of the stack (bit 15 of the last halfword of stack header = 1).

The instruction operation is suppressed.

Task Interval Timer Exception

A task interval timer exception occurs if the task interval timer is decremented through zero during the execution of a timed task. A task interval timer exception causes control to be passed to the exception handling routine.

Verify Exception

A verify exception occurs when:

- An LVT, AHSPOI, ASHPO, or AFSPO instruction detects an invalid operand.

The instruction is suppressed.

INSTRUCTION LENGTH COUNT AND INSTRUCTION ADDRESS REGISTER SETTINGS

Program Exceptions and Instruction Length Count Settings in Chapter 2 describes, in general, how the ILC (instruction length count) and IAR (instruction address register) fields of the CRE (call/return element) or TDE (task dispatching element) are set after an exception occurs. This section provides more detail about how certain specific situations are handled.

The IAR value stored into the CRE or TDE is reduced by the value contained in the instruction length register and zero is stored into the ILC field of the CRE or TDE for the following situations:

- A programmable address compare exception.
- An address translation exception.
- A completed implicit or explicit SVL (supervisor linkage) that encounters one or more access exceptions.
- Any queuing instruction that encounters an access exception.
- Any of the interruptible operations that are suspended due to an external interrupt or page fault. The interruptible instructions are:
  Dequeue Message
  Enqueue Message
  Receive Message
  Send Message
  Send Message and Wait
  Move Characters Long
  Compare Logical Characters Long
  Edit Packed Decimal
  Trim
  Convert Characters to Multi-Leaving
    Remote Job Entry
  Convert Multi-Leaving Remote Job
    Entry to Characters
  Convert Characters to SNA
  Convert SNA to Character
- An unsatisfied receive for the following operations:
  Receive Count
  Receive Message
  Supervisor Linkage Short
  Supervisor Linkage Single
  Supervisor Linkage Double
  Any of the implicit SVLs

The ILC field value is made zero and the IAR field value is not reduced for the following situations:

- When a programmable address compare exception that is not on the instruction stream occurs, for example, data or I/O.

- When filling the MCLB (machine check log buffer) on a soft machine check report.

- A task interval timer exception occurs.

- When a task is switched out (except when no CRE was available during an exception SVL). In the case of an exception SVL with no CRE available, the value of the IAR and ILC fields are determined by the type of exception that occurred.

- If in PEM (program event monitor) mode and a PEM exception occurs.

The value of the ILC field is unpredictable in the CRE or TDE if a HMC procedure is also indicated in the CRE or TDE as follows:

- When a new task is switched in, the ILC field value is loaded into the instruction length register. If no program exception is pending, the new task is switched in and the IAR is reduced by the ILC prior to the execution of the first instruction. When initially built, a new TDE should have the ILC initialized to zero.

- When a Supervisor Exit instruction is executed, the ILC field value is ignored. If no specification exception is detected during execution of the Supervisor Exit instruction, the instruction length register is made zero; if a specification exception is detected, the instruction length register is left as is (containing a value of 2) and the exception is presented.

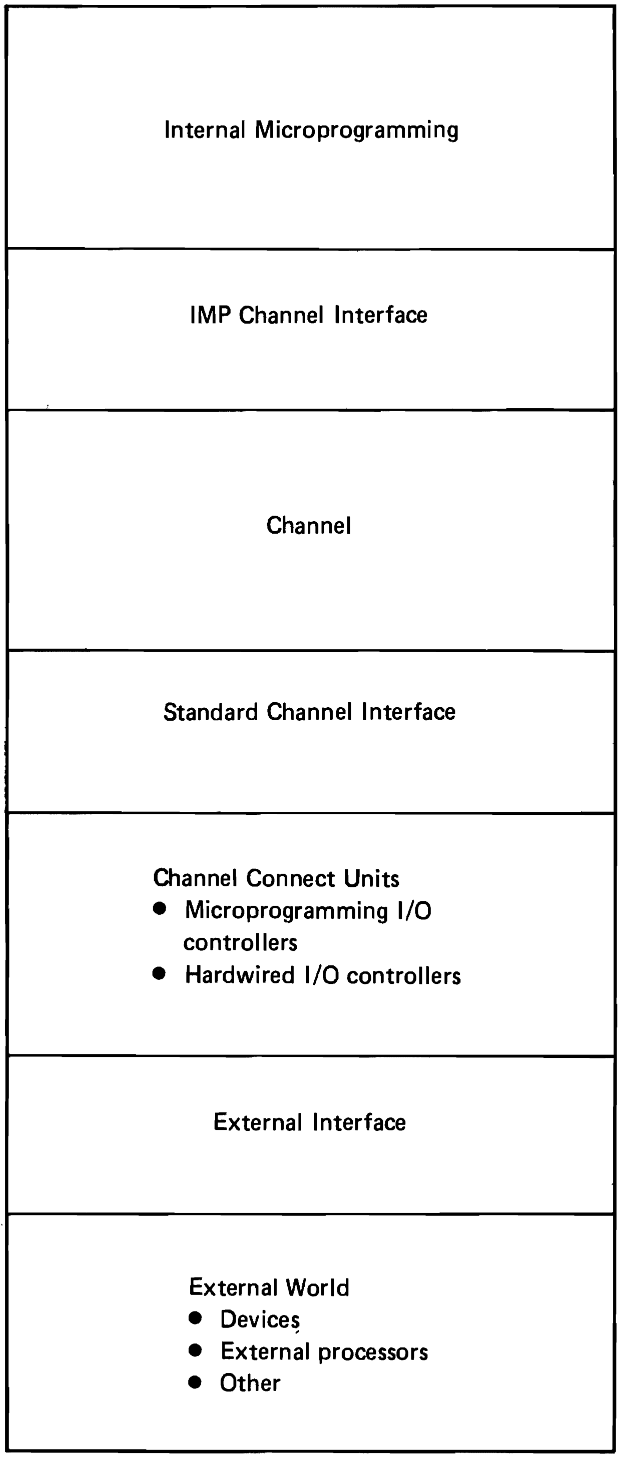
Chapter 7. I/O and Asynchronous Events

This chapter describes the interface between IMP-IOM (I/O manager) tasks, which translate system I/O requests into a form recognizable by the I/O channel (channel-directed commands), and OU (operational unit) tasks, which execute the translated I/O requests within the channel. The format and meaning of the information passed across this interface is described in detail.

A general view of System/38 I/O structure is shown in the following diagram. The interface under discussion is the IMP-channel interface.

The other levels of interaction are described in the *Channel Theory-Maintenance* manual.

Interface Overview



Internal Microprogramming

IMP Channel Interface

Channel

Standard Channel Interface

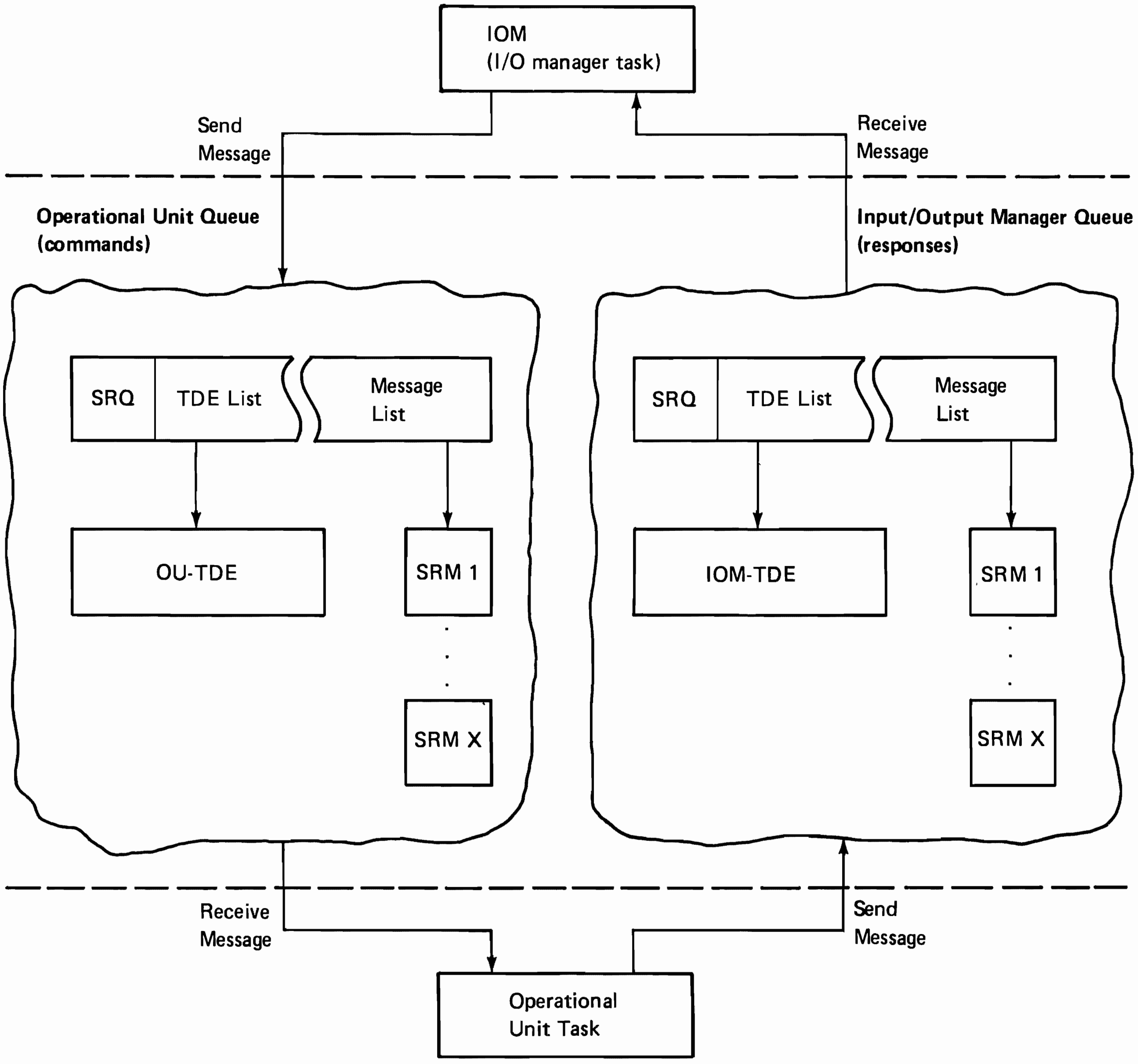
Channel Connect Units
- Microprogramming I/O controllers
- Hardwired I/O controllers

External Interface

External World
- Devices
- External processors
- Other

The IMP-channel interface is sufficiently generalized to allow user to user communications with any source of asynchronous events; for example, I/O devices, external processors, and operator commands from the system console.

The interface mechanism used for the IMP-channel interface is the send/receive queuing structure described in Chapter 5. All interfaces are handled by exchanges of messages between tasks on send/receive queues. The following illustration is an overview of the IMP channel interface mechanism.



From the viewpoint of an IMP-IOM task, I/O and other asynchronous operations appear similar to any other running IMP task.

Rather than interrupting IMP processing to signal an event or condition, all I/O and/or other asynchronous event sources are handled by channel processing functions and the OU (operational unit) task. The OU task can receive messages, send messages, or both.

An overview of System/38 I/O structure is shown in Figure 7-1.

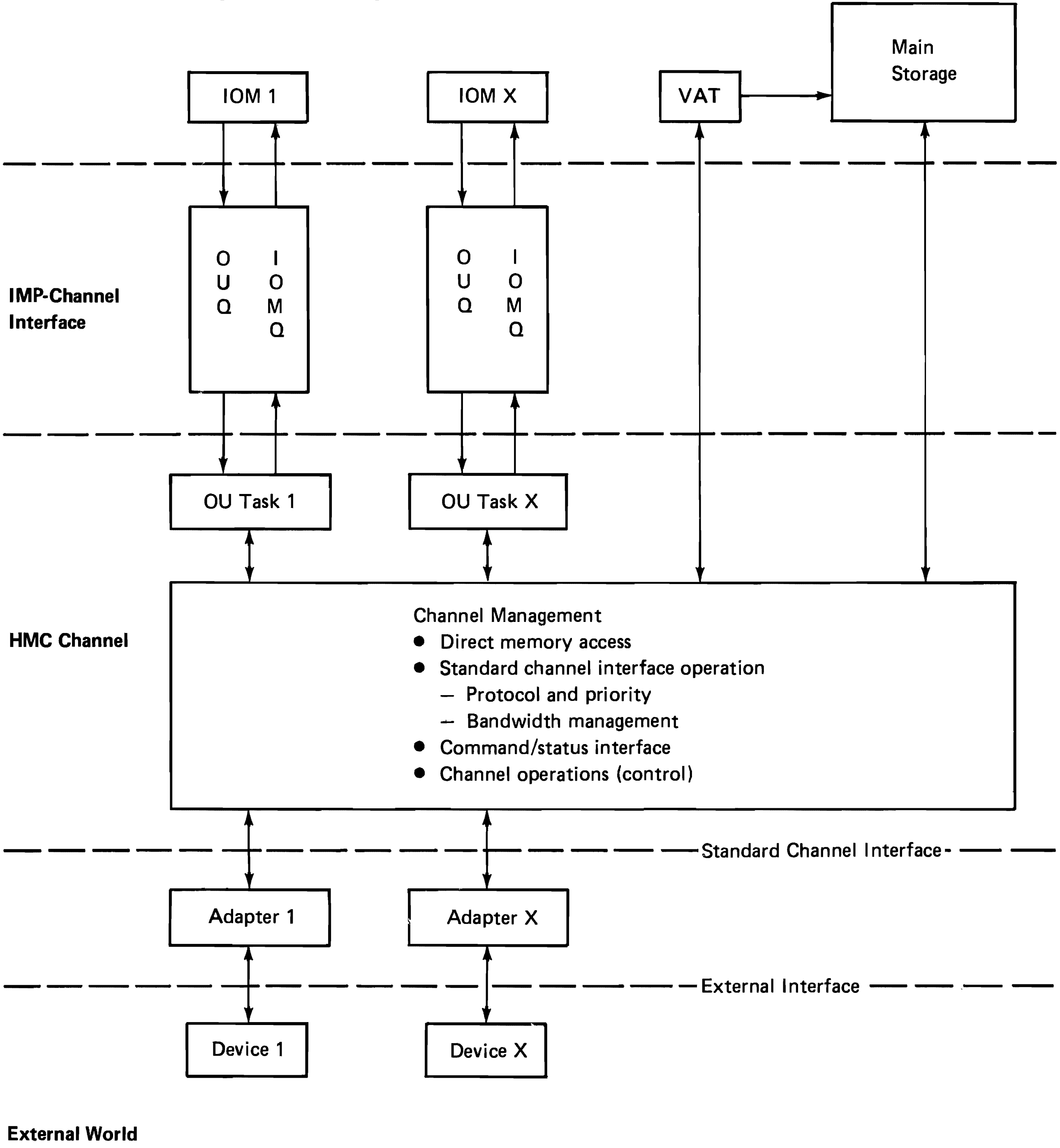


Figure 7-1. System/38 I/O Structure

Asynchronous Operations

Asynchronous I/O operations are requested or enabled by IOM (I/O manager)-formed, channel-directed work requests contained within an ORE (operation request element). An ORE is a part of an SRM (send/receive message) in which the text portion has a special meaning to I/O OU (operational unit) tasks. The text portion of the ORE consists of two fields:

- A 2-byte OU response field (BSTAT).

- A 16-byte, IOM-formed, channel-directed command OB (operation block). An OB, of which there are five types, is, in effect, a channel instruction executed by the OU task.

The OB always has a channel order field executed by the OU task and may have a device order field containing a specific device command and command-related information. The device order field is passed to the device for execution. Of the five OB types only the FOB (function operation block) contains, in addition to the channel order field, the device order field.

Refer to Figure 7-2. An IMP-IOM task can request an asynchronous I/O operation by sending an ORE to a predefined OUQ (operational unit queue). The OU task responsible for servicing the queue receives the ORE, initiates the requested action by executing the OB of the ORE, and passes any required device command and command related information to the device identified in the OB.

An I/O device (or other source of asynchronous event), together with the task that controls the device (or event), is called an OU (operational unit).

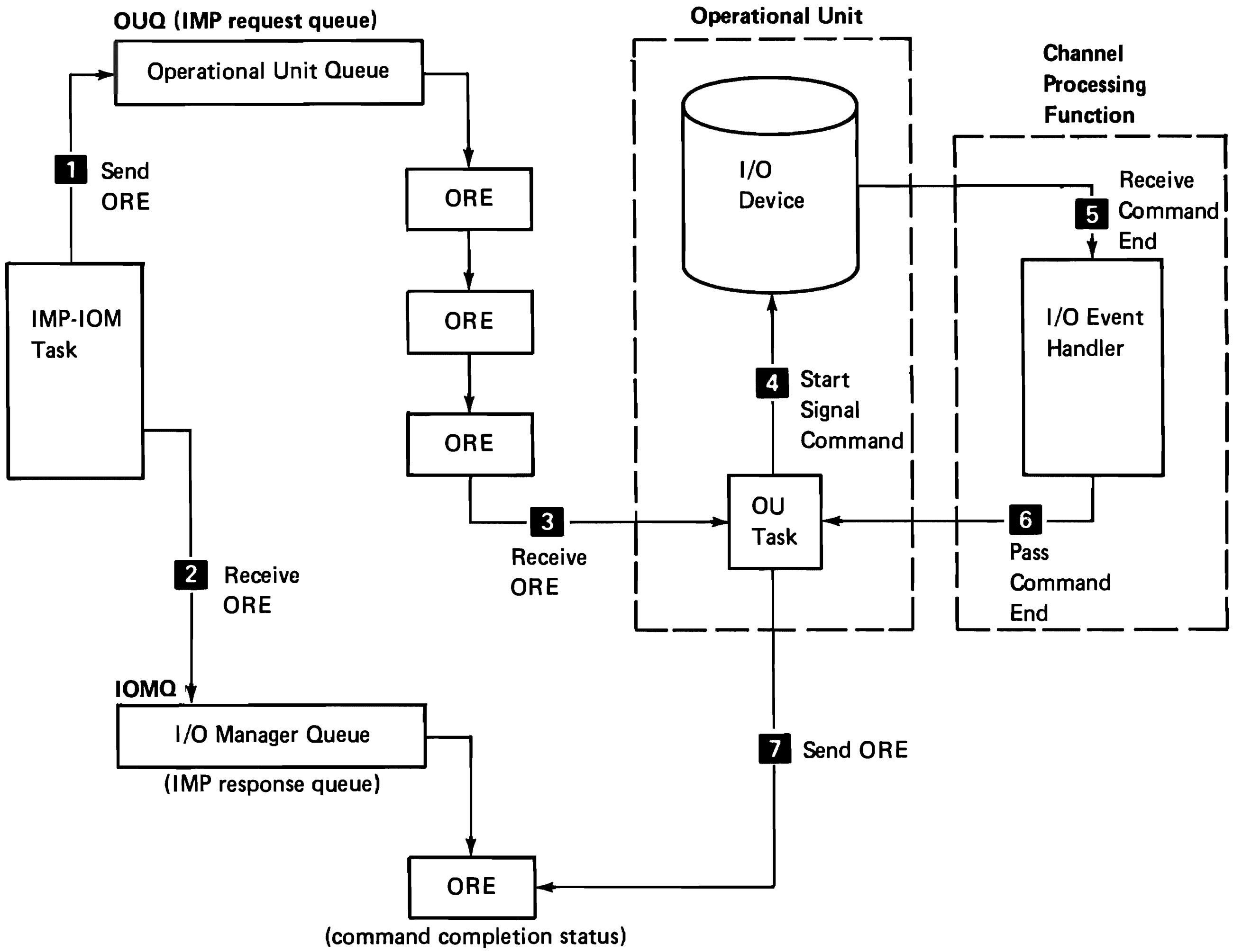
There is one OU task and one pair of queues (OUQ and IOMQ) for each OU.

In general, there is a one-to-one relationship between IOMs and OUs. However, a single IOM can serve multiple OUs.

Information about the completion of the requested operation is sent back to the IMP-IOM task when the OU task places completion status into the 2-byte OU response field of the ORE and sends the ORE to an IOMQ (I/O manager queue) used as a response queue. The IMP-IOM task obtains the information by issuing a receive to the IOMQ.

If a SENDMW instruction was used to send the SRM to an OUQ, the OU task places the information about the completion of the requested operation into the 2-byte OU response field of the ORE. The OU task then causes the task, whose TDE address is in bytes hex 7A-7F or 122-127 of the SRM, to be enqueued to the TDQ. It is not necessary for the OU task to issue a Send Message instruction or for the IMP-IOM task to issue a Receive Message instruction because the TDE created the ORE and knows its address.

The response may indicate successful completion or error conditions for a requested operation.



IMP-IOM Task View:

1 Issues I/O request–send ORE to OUQ.
2 Wait for I/O completion–receive from IOM queue.

I/O Event Handler and OU Task View:

3 Gets command–receive from OU queue.
4 Gives command to OU–start signal to device.
5 Wait for device completion–wait for command end.
6 Gives completion to OU task.
7 Indicates command completion to IMP–send ORE to IOM queue.

Figure 7-2. Asynchronous Operation Queuing Structure

Operational Unit Task

The OU (operational unit) task uses HMC functions that allow an I/O unit to participate in the IMP send/receive queuing structure. A single OU task exists for each I/O unit, and like an IMP-IOM (I/O manager) task, is represented as a separately identifiable unit of execution in the machine by a TDE (task dispatching element). The OU task competes for the processor with IMP tasks and other OU tasks through the priority mechanism of the TDQ (task dispatching queue). The IMP facilities for virtual addressing, addressing exceptions, message queuing, machine check, and task dispatching are available to OU tasks.

The OU task associated with an I/O unit is invoked by either a channel-processing function (I/O event handler) or an IMP-IOM task. The specific operation performed by the OU task is contained in either the ORE (operation request element) or an I/O event field located in the queue control table event stack of the task. The operations involve command completion functions (command end and command end-fetch next command), execution of OBs (operation blocks) and OPs (operation programs), page fault resolution, and the modification of addresses contained in I/O resolved address registers.

The components of an OU task are referred to in the following diagram:

- A set of HMC procedures.

- Task control information:
  - A QCT (queue control table) that contains task control parameters.
  - A QCT event stack that contains I/O event fields to be processed by the OU task.
  - An address list containing ALEs (address list elements) used in page chaining operations.

- A TDE (task dispatching element).

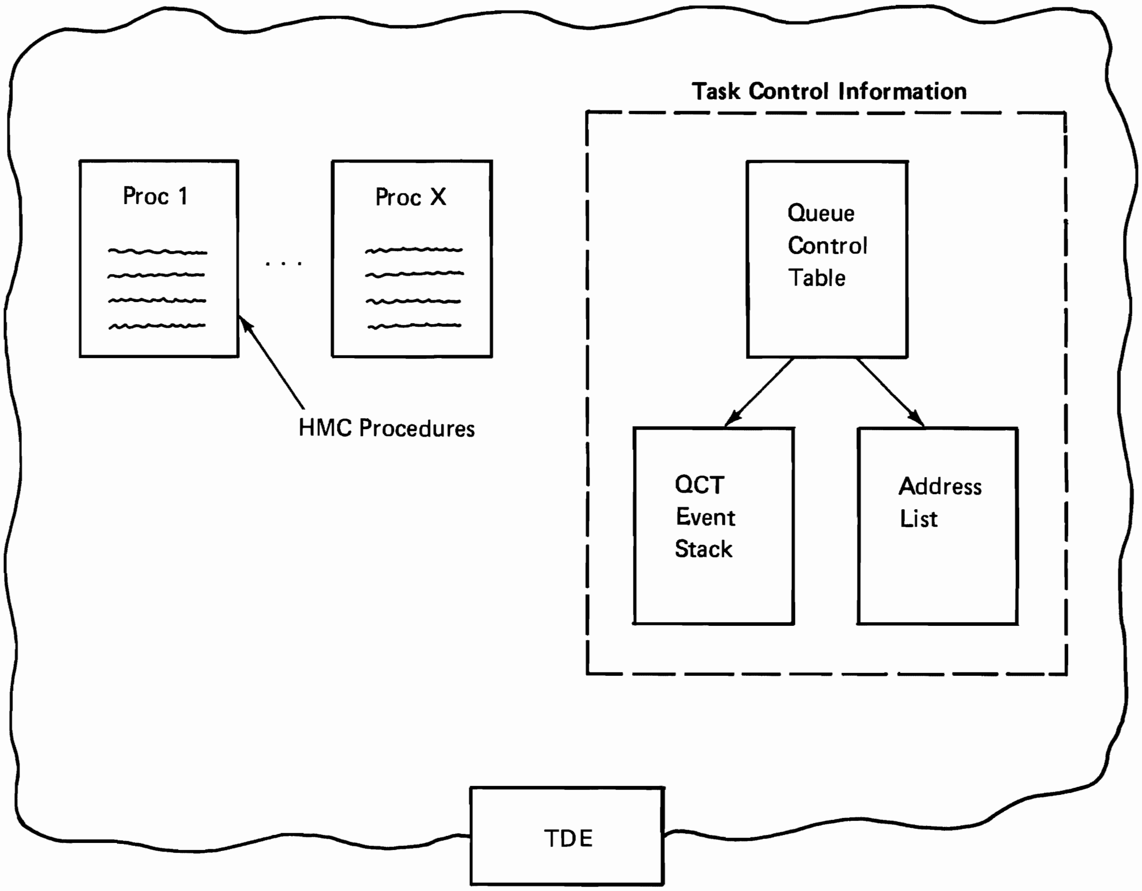
The OU task performs:

- Execution of OBs and OPs.

- Command completion functions.

- Modifications of I/O resolved address registers.

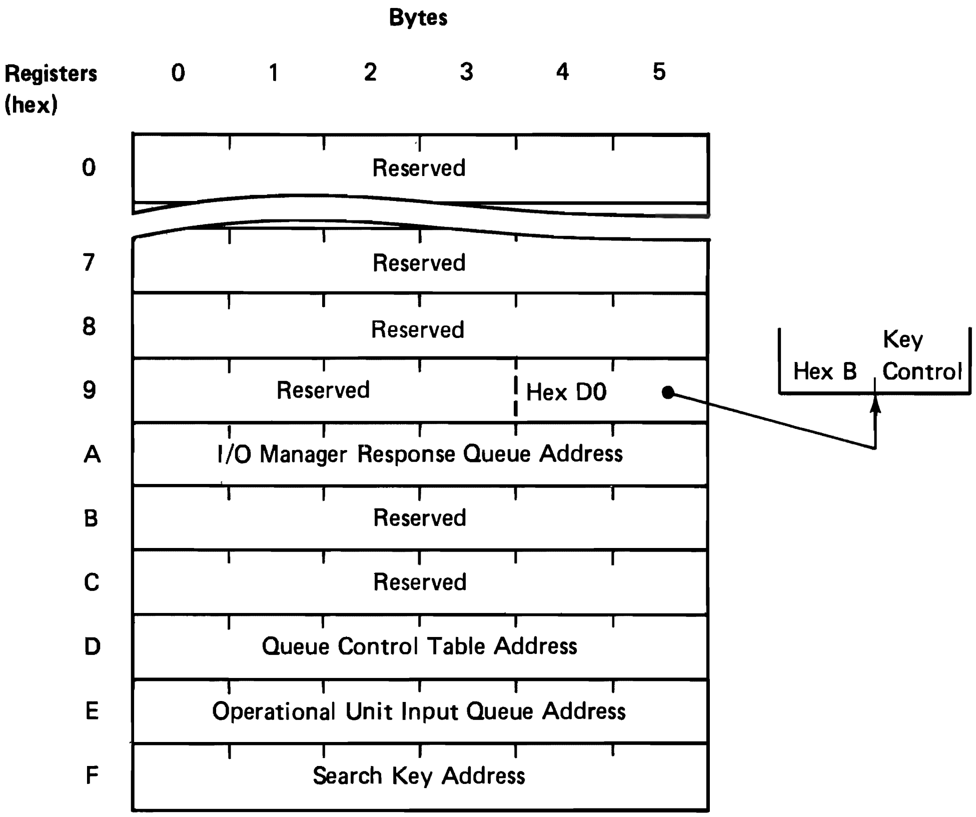
Operational Unit Task Components



Programming Notes:

1. Initialization of the task control information is the responsibility of the IOM of the OU.
2. The OU has a base register work area (bytes hex 32-91 in the TDE), which must be initialized by the IOM of the OU in accordance with the following diagram.

OU Task Base Register Assignment



Registers (hex)	Bytes 0	1	2	3	4	5
0	Reserved					
7	Reserved					
8	Reserved					
9	Reserved				Hex D0	
A	I/O Manager Response Queue Address					
B	Reserved					
C	Reserved					
D	Queue Control Table Address					
E	Operational Unit Input Queue Address					
F	Search Key Address					

Hex B	Key Control

IMP OBJECTS: THEIR FORMATS AND OPERATION

The formats and operation of the IMP system objects TDQ (task dispatching queue), TDE (task dispatching element), SRQ (send/receive queue), SRM (send/receive message), and SRC (send/receive counter), are described in Chapter 5.

The application of some of the objects in the I/O structure and the formats and operational characteristics of objects unique to the I/O structure are described in the following sections.

Unless otherwise stated, *all unassigned fields in the I/O object formats are considered to be reserved and must not be used. Such fields should contain zeros.*

Internal Microprogramming Channel Objects

OPERATIONAL UNIT

An OU (operational unit) is an I/O object consisting of an OU task and the I/O unit (device).

An OU has a unique byte code descriptor used for I/O device addressing by an IOM (I/O manager) task, called the operational unit number. The OU number is in all OBs (operation blocks) contained in an operation program.

The OU number in the FOB (function operation block) is used by the channel to initiate a start device channel hardware operation.

Assignments

The assignment of OU numbers, channel priorities, and I/O resolved address registers for I/O units attached to System/38 is controlled via the system configurator.

Programming Notes: The OU number is a unique, 8-bit code. For some I/O devices the code's format allows hardware field-replaceable unit personalization for multiple I/O devices of a given type.

The format of the OU code implemented by I/O adapters is:

Modifier	Group	Device Code

0 Bits 2 4 8

Bits	Description
0-1	Modifier: Indicates multiple devices of a given type. The 2 bits are hardware programmable at the card field replaceable unit level on the I/O port.

00	First device of a given type.
01	Second device of a given type.
10	Third device of a given type.
11	Fourth device of a given type.

2-3	Group: A functional group or category of I/O devices (magnetic media, card I/O, communications, or other).

00	Group 0
01	Group 1
10	Group 2
11	Group 3

4-7	Device Code: A specific device address code.

OPERATIONAL UNIT QUEUE

The OUQ (operational unit queue) is an IMP send/receive queue used to communicate I/O command request information to a device OU (operational unit) task from an IMP IOM (I/O manager) task.

The TDE (task dispatching element) of the OU task is enqueued to the OUQ when the OU task is not busy.

A Send Message instruction issued to the OUQ from an IOM procedure will cause the OU task TDE to be enqueued to the TDQ (task dispatching queue) for dispatching.

Elements on the OUQ message list are OREs (operation request elements) containing OBs (operation blocks) to be processed by the OU task.

The ORE is obtained by the OU task performing a receive message operation on the queue. The key control (search type) and search key used to dequeue the ORE are contained in the OU task base registers and the queue control table.

Format: The OUQ header format is the same as an SRQ (send/receive queue) header with a key length specification of 4 bytes. This format is shown in Figure 7-3 and the description is in Chapter 5.

Programming Notes:

1. The OUQ header is fullword aligned and must not cross a page boundary. It may be located in any virtual or real storage location (subject to the above restriction).
2. The address of the OUQ header and the key control used during the receive message operation is provided in the base registers of the OU task. Note that the fields must be initialized before an IOM procedure issues a Send Message instruction to the queue.
3. The 4-byte search key used by the OU task to dequeue an ORE during the receive message operation is in the queue control table. Note that the field must be initialized before a Send Message instruction is issued by an IOM task.
4. The key length of all elements on the OUQ is 4 bytes. One OUQ exists for each OU task in the system.

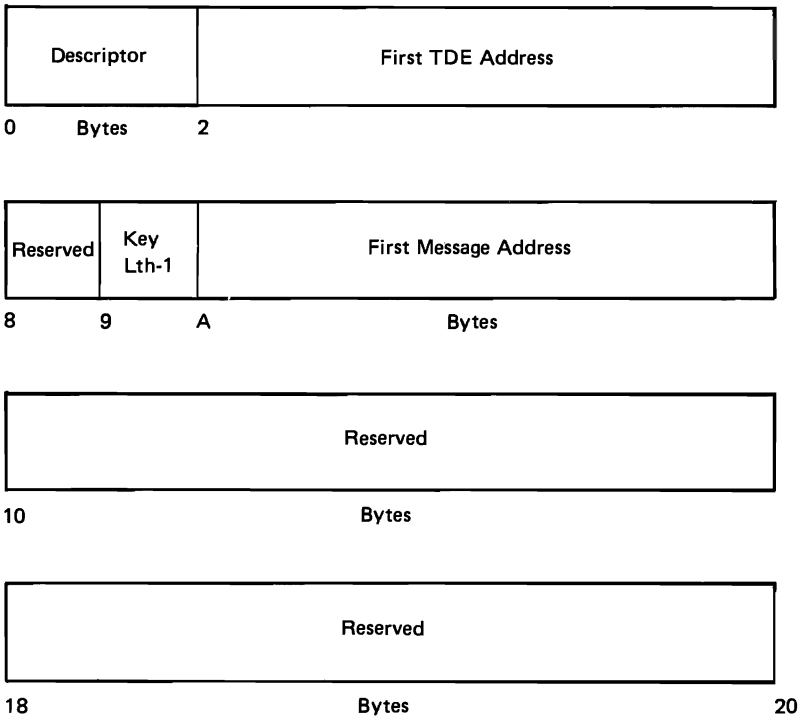


Figure 7-3. I/O Manager Queue, Operational Unit Queue and Send/Receive Queue Headers

I/O MANAGER QUEUE

The IOMQ (I/O manager queue) is an IMP send/receive queue used to communicate I/O command response information to an IOM (I/O manager) task from a device OU (operational unit) task. Elements on the IOMQ message list are OREs (operation request elements) that have been processed by the OU task and contain command completion status. The completed elements are enqueued on the list by the OU task performing a send message operation.

The position of the ORE on the message list is determined by the key field in the ORE and the key control used for the send message operation.

If the TDE (task dispatching element) of an IOM task is enqueued to the IOMQ wait list, it is enqueued to the TDQ (task dispatching queue) when the OU task performs the send message operation.

Format: The IOMQ is an SRQ (send/receive queue) with a key length specification of 4 bytes. This format is shown in Figure 7-3 and the description is in Chapter 5.

Programming Notes:

1. The IOMQ header must be fullword aligned and must not cross a page boundary. It may be located in any virtual or real storage address.
2. The address of the IOMQ to be used by a OU task is contained in the base registers of the OU task. Note that the address must be initialized before any Send Message instructions are issued to the OUQ serviced by the OU task.
3. The key length specification of all elements on the IOMQ is 4 bytes. In general, one IOMQ will exist for each IOM task in the system.

OPERATION REQUEST ELEMENT

The ORE (operation request element) is an IMP send/receive message element used to communicate I/O command and response information between an IOM (I/O manager) procedure, and an OU (operational unit). As illustrated in Figure 7-2, the ORE is formed and enqueued to the OUQ (operational unit queue) by the IOM procedure using a Send Message instruction (label 1 in the figure). The Send Message instruction contains both the OUQ header address and the address of the ORE to be enqueued.

The OU task removes each ORE from the OUQ by performing a receive message operation. The address of the dequeued ORE is contained in the OU task base registers during the processing of the operation blocks contained in the ORE (label 3 in Figure 7-2).

When all operation blocks in the ORE have been processed by the OU task and its associated I/O device, the ORE, containing command completion status, is enqueued to the IOMQ by a send message operation performed by the OU task (label 7 in Figure 7-2).

Note: The ORE is not physically moved in main storage during the above operations. The ORE is enqueued and dequeued from the OU and IOM queues through the manipulation of addresses.

Format:

Descriptor	Next Message Address
0 Bytes	2

Key	Reserved	OU Status (BSTAT)
8 Bytes	C	E

Operation Block	
10 Bytes	20

Bytes (Hex)	Bit	Description
0-1		**Descriptor:** The object descriptor for an SRM (send/receive message) element. See Chapter 5 for the SRM bit description.
2-7		**Next Message Address:** The virtual address of the next ORE in a list, when the ORE object is enqueued on an OU or IOMQ.
8-B		**Key:** The value used to enqueue and dequeue the ORE from the OU and IOM queues.
C-D		**Reserved:** Must be zeros.
E-F		**OU Status (BSTAT):** OU status information for the IOM procedure that initially issued the ORE. The OU status may be formed by either the device adapter or OU task. Excluding the operation program error status provided by the OU task, the OU status field is updated for each FOB (function operation block) executed in an ORE.

The usage of each status bit is described below.

Bytes (Hex)	Bit	Description
E	0	Reserved: Must be zero.
	1	Operation Program Error: Is set when an error condition is detected during the processing of OBs (operation blocks) by the OU task. The specific type of error is indicated in byte hex F of the ORE.
	2-3	Reserved: Must be zeros.
	4	I/O Exception: Is set to indicate a device exception condition during the execution of an FOB command by a device adapter (the I/O exception may be suppressed if command complete bit is also set).
E	5	Command Reject: Is set when a device adapter detects an invalid command or is in a state that prevents execution of the FOB command.
	6	I/O Error: Is set when a device dependent error condition is detected during execution of an FOB command.
	7	Command Complete: Is set when an FOB command has been executed to successful completion by the OU.
F	0-7	OU Status: May contain either device dependent status provided by the I/O device at the completion of a FOB command or an OP (operation program) error provided by the OU task. Use of this byte is optional and device dependent.
10-1F		**Operation Block:** Can contain one of the following operation blocks: POB program operation block), FOB (function operation block), AOB (address operation block), or MOB (message operation block). A fifth operation block, the LOB (loop operation block), may not appear in an ORE. Descriptions of the operation block formats and their operations are provided under *Operation Blocks* in this chapter.

Programming Notes: The ORE must be doubleword aligned and may not cross a page boundary. It may be located at any virtual or real address.

OPERATION BLOCKS

The OBs (operation blocks) contain command requests to an OU (operational unit) from an IOM (I/O manager) procedure. The OBs are included in an ORE (operation request element) and are processed by the OU task of the OU.

The five OB types are:

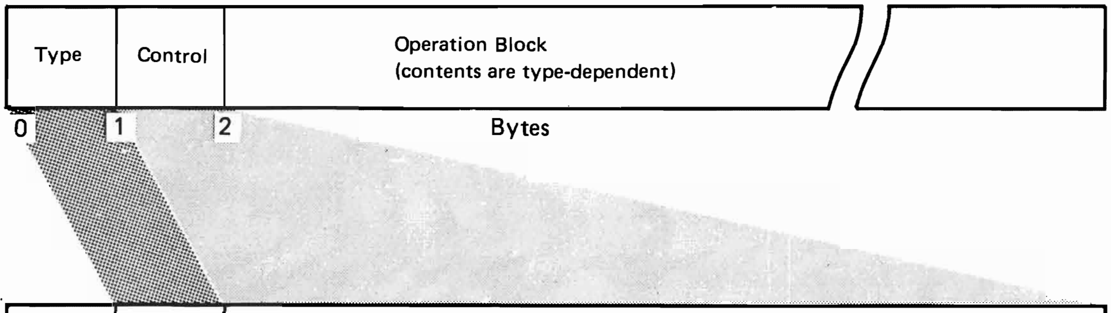
- POB (program operation block)
- FOB (function operation block)
- AOB (address operation block)
- LOB (loop operation block)
- MOB (message operation block)

The formats and operations of the five OBs are described in the following sections.

Formats: The OB is a 16-byte object that must be aligned on a doubleword boundary, and may not cross a page boundary.

As indicated in the first diagram, byte 0 contains a type code to indicate the specific OB. Byte 1 contains control information used by the OU task during the execution of the OB. The type code and control bit assignments for each OB type are shown in the second diagram.

Operation Blocks: Types and Control Information.



Operation Block Type	Byte 0 (Type in Hex)	Byte 1 (Control)							
		0	1	2	3	4	5	6	7
POB	D7	Reserved (hex 00)							
FOB	C6	End of Operation Program				Set Data Address	Page Chain	Load Unique I/O Registers	
AOB	C1	End of Operation Program				Save Data Address	Modify Address	Decrement	Set Data Address
MOB	D4	End of Operation Program				Message			
LOB	D3	End of Operation Program							

Types of Operation Blocks

Program Operation Block

The POB (program operation block) specifies that a sequence of OBs (operation blocks) are to be executed by the OU (operational unit) before the ORE (operation request element) is returned to the IOM (I/O manager) program. The OBs associated with the POB are referred to as an operation program.

A POB contains the address of the first OB in the operation program and the address of the current (or last) OB in the operation program processed.

During the processing of OBs, the address of the current OB is maintained in bytes 2-7 of the POB and the current OB address field is incremented by 16 as each OB in the program is processed. The LOB (loop operation block) may be used in the operation program to modify the current OB address nonsequentially.

At the successful completion of an operation program, the current OB address field contains the address of the last OB processed.

Format:

Type D7	Control	Current Operation Block Address	
0	1	2 Bytes	

OU	Reserved	Operating Program Address	
8	9	A Bytes	10

Bytes (Hex) Description

0 **Type**: POB type code (hex D7).

1 **Control**: Control field (hex 00).

2-7 **Current OB Address** (virtual address): The field must be initialized with the address of the first OB to be executed in the operation program.

Following successful execution of the operation program, the field contains the address of the last OB processed. If the operation program is terminated due to an error, the field contains the OB address in process when the error was detected.

8 **OU**: A valid OU (operational unit) number.

9 **Reserved**: Must be zeros.

A-F **Operation Program Address**: Virtual address of the first OB (operation block) in the operation program. The field is not modified during the operation program.

Programming Notes:

1. The POB must be aligned on a doubleword address boundary, and may not cross a page boundary. It may not be imbedded in an operation program.
2. The current OB address field does not have to be the address of the first OB in the operation program.

Function Operation Block

The FOB (function operation block) conveys command information to an I/O device attached to the channel. The FOB may be contained in an operation program.

The virtual storage address used by the I/O device during execution of the FOB command is provided by the data address field and the control field as follows:

- If the control field indicates that page chaining is not used, then the data address field of the FOB contains the I/O address to be resolved and loaded into the data I/O resolved address register of the device.
- If page chaining is used during the data transfers, the data address field contains the address of a stack of ALEs (address list elements).

Bytes hex 8-15 of the FOB provide command information to the I/O device and optionally provide status from the device following execution of the FOB command.

Format:

Type C6	Control	Data Address	
0	1	2	Bytes

OU	Com- mand	Command/Response		
8	9	A	Bytes	10

Bytes (Hex)	Bits	Description
0		**Type:** FOB type code (hex C6).
1		**Control:** Control field.
	0	End-of-Operation Program: Is set to one if the FOB is the last OB in an operation program; otherwise is zero.
	1-3	Reserved: Must be zeros.
	4	Set Data Address: When set, causes a virtual address to be resolved and loaded into the I/O resolved address register specified by the data register field of the QCT (queue control table). The actual address to be resolved is determined by the page chaining bit.
	5	Page Chaining: If the page chaining bit is set, when the set data address bit is set, then the I/O address to be resolved and loaded into an I/O resolved address register is located in an ALE (address list element) addressed by the data address field of the FOB.
	6	Load-Unique I/O RAR (resolved address register): When the load-unique I/O RAR bit is set simultaneously with the set data address and page chaining bits, it indicates that the data address in the first ALE stack entry will be resolved and loaded into the I/O resolved address register specified by the CMD REG field of the queue control table, plus one. When the unique I/O RAR is loaded into the specified CMD REG field, HMC interrogates byte 9, bit 1 and byte hex A. If byte 9, bit 1 is zero and byte A does not equal hex 01, then the unique register is marked invalid and the store allowed bit in the QCT is set. After loading the unique I/O resolved address register, the OU task will load the address of the second ALE stack entry into the QCT ALE pointer field. When the I/O adapter transfers a load-multiple I/O register function event to the channel, the I/O event handler will load ALE data addresses into consecutive I/O resolved address registers, starting with the data register specified in the QCT. The unique I/O RAR is marked valid and the store allowed bit is set or reset to reflect the state of the store allowed bit in the QCT.
	7	Reserved: Must be zero.

Bytes (Hex)	Bits	Description
2-7		**Data Address:** Either an I/O address or the address of the first ALE (address list element) of a page chain address list if the set data address bit is set. The field should contain zeros if the set data address bit is zero. If page chaining is used, then the address must be virtual = real.
8		**OU:** A valid operational unit number.
9		**Command:** An I/O command code to be interpreted and executed by the I/O device. The format of the CMD field is as follows:
	0-4	Device Dependent Control: A device dependent command code defined by each device adapter.
	5	Control: An I/O command that may or may not involve a data transfer to or from main storage.
	6	Read: An I/O command that involves a data transfer from an I/O device to main storage. The change bit in the primary directory entry (bit 42) for the page to be resolved is set to the value of the read bit when the set data address bit (byte 1, bit 4) of the control field is set.
	7	Write: An I/O command that involves a data transfer from main storage to an I/O device. **Note:** The contents of bits 5-7 are not recognized by horizontal microcode.
A-F		**Command/Response:** Provides command information to the device and optionally provides extended response status from the device. Two formats of the command/response field are defined in the following diagram. Implementation of the field format is command dependent with only one format valid per FOB. The basic format provides 6 bytes of device/command dependent information in addition to the OU field and the CMD field (see the following diagram). The second command/response format provides 2 bytes of device/command dependent information (bytes hex A and B) and 4 bytes of response status referred to as FSTAT (functional status). The definition of FSTAT is device/command dependent (see Chapter 9) and may be used by the device to provide command completion status in addition to the completion status provided in the OU status field of the ORE (operation request element).

Formats:

FOB Command/Response Fields (Bytes 8-F)

Format 1

OU	Command	Device Command Dependent

8 9 A Bytes 10

Format 2

OU	Command	Device/Command Dependent	FSTAT

8 9 A Bytes C 10

Note: Unused portions of the command/response field should be filled with zeros.

Programming Notes:

1. The FOB must be aligned on a doubleword address boundary and may not cross a page boundary.
2. The I/O data address provided in either the data address field or an ALE (address list element) must be aligned on a doubleword address boundary.
3. The addresses in the ALE stack (page chain address list) must be virtual = real if the load unique I/O register bit is on.
4. Unused portions of the command/response field must be filled with zeros.
5. The address of the page chain address list (ALE stack) must be virtual = real.

Address Operation Block

The AOB (address operation block) provides the ability to save, modify, or load the I/O resolved address registers during the processing of an operation program.

The operation of the AOB is controlled by bits 4-7 of the control field. Proper settings of the control bits provide the ability to perform selected portions of a read, modify, and store cycle. The control bits provide the ability to load the address contained in the I/O resolved address register into the AOB, modify the address in the AOB, and to resolve the address contained in the AOB and load the selected I/O resolved address register.

The I/O resolved address register involved in the AOB operation is selected by adding the register modify field (byte hex A) in the AOB to the data register field (byte 3) of the QCT (queue control table). The register modify field is treated as an unsigned logical quantity. For example, if the data register field in the QCT of the operational unit task contains hex 10, then a register modify field value of hex 01 in the AOB would result in the selection of I/O resolved address register hex 11.

When an address in the AOB is resolved and loaded into an I/O resolved address register, the change bit in the page directory entry for the virtual page is set to the value of the read bit in the command field.

Format:

Type C1	Control	Data Address	Modify Address
0	1	2 Bytes	6

OU	Com-mand	Register Modify	Reserved	Address Modifier
8	9	A	B Bytes	E 10

Bytes (Hex)	Bits	Description
0		**Type:** AOB type code (hex C1).
1		**Control:** Control field.
	0	End Of Operation: 0 AOB is *not* the last OB (operation block) in an operation program. 1 AOB is the last OB in an operation program.
	1-3	Reserved: Must be zeros.
	4	Save Data Address: Causes the offset portion of the virtual address contained in the selected I/O resolved address register to be loaded into AOB bytes 6-7. The SID (segment identifier) portion of the virtual address must be preloaded into AOB bytes 2-5 by the associated IOM. The save operation is performed prior to any address modifications (modify address control control bit) or address resolution (set data address control bit).
	5	Modify Address: When set, causes the address contained in AOB bytes 6-7 to be modified by the value specified in AOB bytes hex E-F (address modifier field). The address modifier field is an unsigned logical quantity and can either be added or subtracted from the address according to the value of the decrement bit. Any carry or borrow generated during the modification is indicated as an operation program error. Bytes 2-5 of the AOB are not affected by the address modification.
	6	Decrement: When set, causes the address modification specified by the modify address bit to be an unsigned subtraction.
	7	Set Data Address: When set, causes the address in AOB to be resolved and loaded into the selected I/O resolved address register.

Bytes (Hex)	Bits	Description
2-5		**Data Address:** The SID portion of a virtual address used during an AOB operation.
6-7		**Modify Address:** The offset portion of a virtual address used during an AOB operation.
8		**OU:** A valid operational unit number.
9		**Command:** Provides control for setting the change bit in the PD (primary directory) entry for the virtual address that is resolved and loaded into an I/O address register. The byte is formatted like the command field of the FOB (function operation block); however, only the read bit (bit 6) is used during the processing of the AOB. If a set data address operation is selected, the change bit in the PD entry for the virtual address is set to the value of the read bit in the command field.
A		**Register Modify:** An unsigned logical quantity used to generate the effective I/O resolved address register number. For V=V data address the maximum value of the Register Modify field is hex 0E. A value greater than hex 0E will generate an operation program error. For V=R data addresses the value contained in the Register Modify field is not constrained.
B-D		**Reserved:** Must be zeros.
E-F		**Address Modifier:** An unsigned logical quantity used to modify the address in the AOB data address field.

Programming Notes:

1. The resultant virtual address in the data address field of the AOB must be aligned on a doubleword address boundary. On an AOB read (save data address) only the offset is obtained from the I/O register. Programming must supply the SID (segment identifier) in bytes 2-5 of the AOB and may have to supply the offset value.
2. The operations selected by the control bits in the control field are performed in the following sequence:
   a. Save data address
   b. Modify address
   c. Set data address
3. An operation program error (described later in this chapter) occurs if the selected register number exceeds the number of I/O resolved address registers available on the system (see *I/O Resolved Address Registers* in this chapter).
4. An operation program error occurs if the register modify field value is greater than hex 0E and the address in the data address field is not V=R.
5. An operation program error will be indicated if the modified address crosses a segment boundary.
6. If the AOB read/modify/write or read/write is to be done, then the 4-byte SID of the address to be read must be supplied by the user generating the AOB.
7. An operation program error will be indicated if the decrement bit is on while the modify bit is not on.

Loop Operation Block

The LOB (loop operation block) helps provide sequence control of the operation blocks in an operation program.

Each time the LOB is encountered during the execution of an operation program, the count field of the LOB is incremented by 1 and compared to the contents of the limit field. If the modified count is less than the limit value, then the next OB (operation block) to be processed by the OU (operational unit) task is located by subtracting the offset field from the current OB address field of the POB (program operation block) in the ORE (operation request element).

If the modified count field equals or exceeds the limit value, then the count field is set to zero and the next sequential OB in the operation program is executed. If the limit field is initially zero, the LOB is treated as a no-operation and the next sequential OB is executed.

If an operation program containing an LOB terminates before completion, the count field in the LOB will contain the number of times the OB loop was executed before the error situation was detected.

Format:

Type D3	Control	Reserved	Offset	
0	1	2 Bytes	6	

OU	Reserved	Count	Limit	
8	9 Bytes	C	E	10

Bytes (Hex)	Bits	Description
0		**Type:** LOB type code (hex D3).
1		**Control:** Control field.
	0	End-Of-Operation Program: 0 LOB is *not* the last OB in an operation program. 1 LOB is the last OB in an operation program.
	1-7	Reserved: Must be zeros.
2-5		**Reserved:** Must be zeros.
6-7		**Offset:** Used to modify the current OB address field in the POB. The offset value is the number of OBs in the loop multiplied by 16.
8		**OU:** A valid operational unit number.
9-B		**Reserved:** Must be zeros.
C-D		**Count:** The number of times the LOB has been processed. The count field is set to zero when the count equals or exceeds the limit value.
E-F		**Limit:** The number of times the OB loop is to be processed. The limit field is not modified by the OU task.

Programming Notes:

1. An operation program error occurs if the modified OB address is less than the operation field in the POB.
2. The LOB may not appear in an ORE (operation request element).
3. If the LOB is the last OB of an operation program, the program is not completed until the count of the LOB equals the limit value.

Message Operation Block

The MOB (message operation block) causes an IMP task or an OU (operation unit) task to become dispatchable during the execution of an operation program. This facility can be useful in prefetching, for example, items such as virtual storage pages and data translation operations.

If the message bit of the MOB is set, then an implicit Send Message instruction is performed by the OU task processing the MOB. The Send Message instruction causes the SRM (send/receive message) element whose address is specified by the message address field of the MOB to be sent to the SRQ (send/receive queue) designated by the address contained in the target address field. Any TDE (task dispatching element) on the SRQ wait list is enqueued on the TDQ (task dispatching queue) as a result of the operation.

If the message bit is reset, then an implicit Send Count instruction is performed with the address of the SRC (send/receive count) provided in the target address field of the MOB.

Format:

Type D4	Control	Target Address
0	1	2 Bytes

OU	Reserved	Message Address	
8	9	A Bytes	10

Bytes (Hex)	Bits	Description
0		**Type:** MOB type code (hex D4).
1		**Control:** Control field.
	0	End-Of-Operation Program: 0 The MOB is *not* the last OB (operation block) of the operation program. 1 The MOB is the last OB of the operation program.
	1-3	Reserved: Must be zeros.
	4	Message: 0 A Send Count instruction is to be performed. 1 A Send Message instruction is to be performed. The address of the SRM element to be used in the Send Message instruction is provided in the message address field.
	5-7	Reserved: Must be zeros.
2-7		**Target Address:** The virtual address of the SRQ (send/receive queue) or SRC (send/receive counter).
8		**OU:** A valid operational unit number.
9		**Reserved:** Must be zeros.
A-F		**Message Address.**

Programming Notes:

1. Depending on the priorities of the TDEs (task dispatching elements) placed on the TDQ (task dispatching queue), a task switch can occur following the execution of the MOB by the OU task. When the OU task that issued the MOB again becomes the current task, the processing of the operation program will resume with the OB following the MOB.
2. The key control (search type) used during a Send Message instruction is contained in the OU task base register.

Operation Program

An operation program consists of one or more OBs (operation blocks) associated with a single ORE (operation request element). The OBs of the rwnoperation program are processed to completion by the OU (operational unit) task before the ORE is returned to the IOM (I/O manager) task (via the I/O manager queue). The operation program can contain FOBs, AOBs, MOBs, and LOBs.

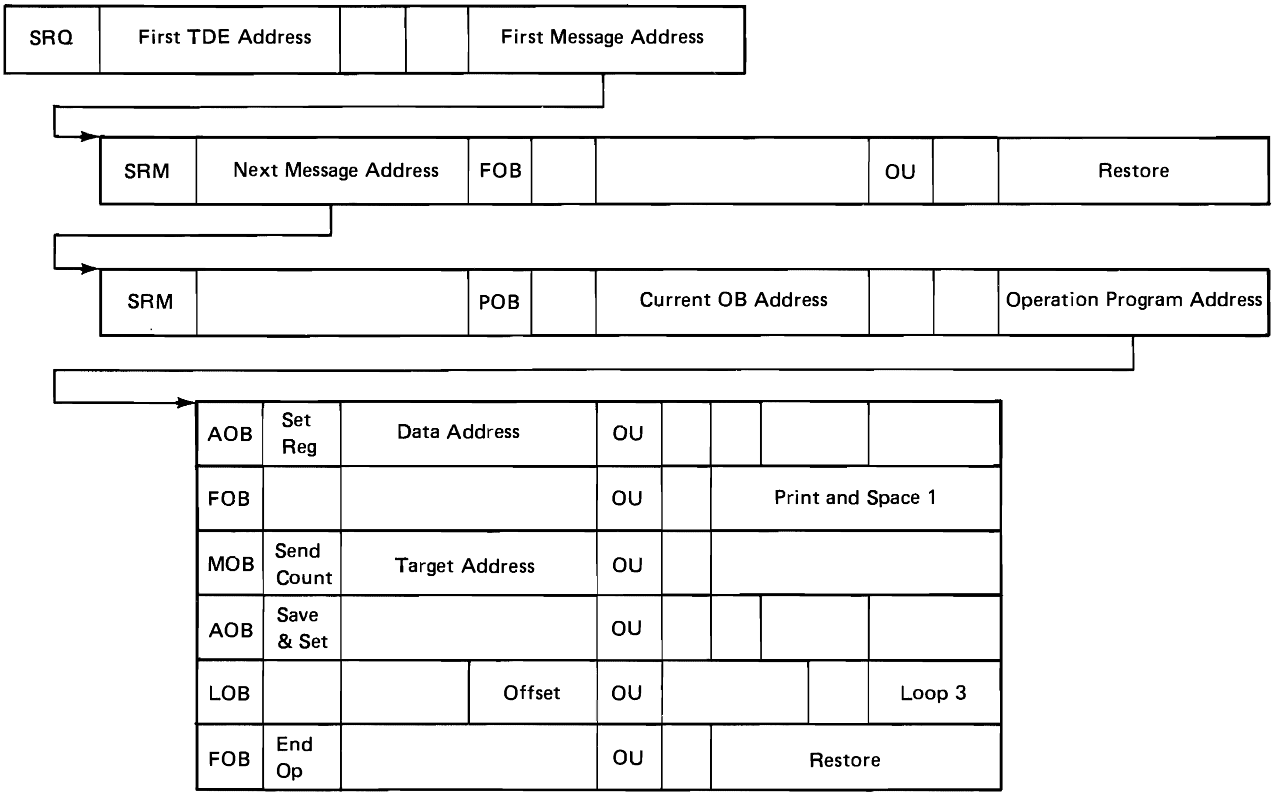
The LOB (loop operation block) is used to modify the current OB address in the POB (program operation block) during OU task execution, providing the capability for nonsequential execution of OBs.

Operation Program Example

The following example shows how different types of operation blocks can be combined to form an operation program. The example is a printer operation program.

An IMP task uses a send instruction to enqueue OREs to the OUQ (operational unit queue) of the printer. This OUQ is associated with a single OU (operational unit), for example, a printer and control adapter. The first ORE on the queue contains an FOB (function operation block) that causes the printer to restore and print a single line. At the completion of the operation, the OU task notifies the IMP task by placing a status byte in the ORE and issuing a Send Message instruction to send the entire ORE to the OU IOMQ (I/O manager queue). The IOMQ is not shown in the example.

Example of a Printer Routine


SRQ | First TDE Address | | | First Message Address
SRM | Next Message Address | FOB | | | OU | | Restore
SRM | | POB | | Current OB Address | | | Operation Program Address
AOB | Set Reg | Data Address | OU | | | |
FOB | | | OU | | Print and Space 1
MOB | Send Count | Target Address | OU | |
AOB | Save & Set | | OU | | | |
LOB | | | Offset | OU | | | Loop 3
FOB | End Op | | OU | | Restore

The print program:

- Prints three lines
- Indicates to a target queue each time a line is printed via an MOB

Programming Notes:

1. The OBs (operation blocks) of the program must be aligned on contiguous doubleword address boundaries.
2. The operation program can be located at any virtual (or virtual = real) address, subject to the above consideration. The operation program can not cross a segment boundary.
3. The last OB of the program must have the end of operation program bit set. If an LOB (loop operation block) is the last OB, then the program will not complete until the value of the LOB count field is equal to the value of the LOB limit field.

QUEUE CONTROL TABLE

The QCT (queue control table) is an OU (operational unit) task control object used by the IOM (I/O manager) procedure, the OU (operational unit) task receiving OREs (operation request elements) from an OU queue, and the I/O event handler. A QCT must exist for each OU task in the machine.

The QCT contains an SRC (send/receive counter) object used by the I/O event handler routine to dispatch an OU task and certain physical parameters associated with the particular I/O device. Certain fields of the QCT are used by the OU task during the processing of operation blocks in an ORE.

The device IOM task can access the QCT to modify certain parameters used by the OU task and I/O event handler.

Format:

QCT Entry

Type D8	Control	Command Register	Data Register	Event Count	Event Offset	Event Limit
0	1	2	3	4 Bytes	6	8

Current ALE Address	SRC Descriptor
A Bytes	10

First TDE Address	SRC Count	SRC Limit
12 Bytes	18	1A

Key	FOB Timer Count	FOB Timer Limit	
1C Bytes	20	22	24

QCT Event Stack Entry

I/O Event	
0 Bytes	4

Bytes (Hex)	Bits	Description
0		**Type:** QCT type code (hex D8).
1		**Control:** Control field.
	0	Lock: 0 The I/O event handler does not post any event field directed to the QCT to the channel IOM QCT. 1 The I/O event handler posts any event field directed to the QCT to the channel IOM QCT. The bit is set by the event handler if the QCT event stack is full and is set by a device IOM task when any fields in the QCT are being modified.
	1	Chain Address: Set by the OU task during the processing of the FOB (function operation block) with both set data address and page chaining bits set. The bit, when set, causes the event handler to obtain the virtual address to be resolved from an address list during the servicing of an address event.
	2	FOB In Progress: 0 A command has *not* been issued to an I/O adapter. 1 A command has been issued to an I/O adapter. This bit is set when an FOB is issued to an I/O adapter and reset when the I/O adapter responds with a command end or a command end/fetch next command.
1 (cont)	3	FOB Timing In Process: 0 VMC does *not* reset the FOB timers. 1 If this bit is set and the FOB-in-progress bit is reset, the VMC resets the FOB timers. This bit is set by VMC to indicate that VMC has updated the FOB timers. This bit is reset by the OU task when the I/O adapter issues a command end or a command end/fetch next command.
	4	Store Allowed On Data Registers: 0 Store *not* allowed. 1 When an I/O adapter requests a load multiple I/O register function, the store allowed bit is set in the I/O data register. This allows the I/O adapter to transfer data into storage. This bit is set when the OU task processes the FOB if bit 6 of the FOB command field is set. This bit is reset when the I/O adapter responds with a command end or command end/fetch next command.
	5-7	Reserved: Must be zeros.
2		**Command Register:** The address of the I/O resolved address register used by a device to access bytes hex 8-F of the FOB. Each device on the system must have an assigned command address register.

Bytes (Hex)	Bits	Description
3		**Data Register:** The address of the primary data address register used by the I/O device. If more than one data address register is required by the device, the data register byte contains the address of the lowest numbered register assigned to that device. If the device does not require any data address registers, the data register field must be the same as the command register field.
4-5		**Event Count:** The number of events in the QCT event stack. This field is incremented by the event handler when event fields are placed on the QCT event stack.
6-7		**Event Offset:** The address offset, within the same QCT segment, to the beginning of the QCT event stack.
8-9		**Event Limit:** The number of 4-byte event entries that can be put into the QCT event stack.
A-F		**Current ALE Address:** A virtual = real address to an ALE (address list element) in a page chain address stack. The address is valid only during the execution of an FOB in which page chaining is used. This field is modified by the event handler as each address is obtained from the page chain address list.
10-1B		**Send/Receive Counter:** Controls the dispatching of the OU task. The OU task issues a Receive Count instruction to increment the count of the SRC; the event handler issues a Send Count instruction. SRC fields are:

Hex Byte	Contents
10-11	SRC descriptor
12-17	First TDE address
18-19	SRC count
1A-1B	SRC limit

Bytes (Hex)	Bits	Description
1C-1F		**Key:** The search key operand during the Receive Message instruction performed by the OU task.
20-21		**FOB Timer Count:** Time elapsed for current FOB in n-second increments; maintained by VMC (vertical microcode).
22-23		**FOB Timer Limit:** Timing limit for FOB in n-second increments; maintained by VMC.

Programming Notes:

1. The QCT (queue control table) must be in V=R (virtual = real) storage and doubleword aligned. The QCT must not cross a segment boundary. The QCTs for all OU (operational unit) tasks must be in the same V=R segment.
2. Certain fields in the QCT must be initialized by the IOM (I/O Manager) of the OU prior to dispatching the OU task. All fields except the following should be initialized to zeros.
   a. Type (byte 0): Set to the QCT type code of hex D8.
   b. Cmd Reg (byte 2): Set to the address of the command address register for the device.
   c. Data Reg (byte 3): Set to the address of the data address register for the device. If no data address registers are required, the byte must contain the command address register number (same as byte 2).
   d. Event Offset (bytes 6-7): Set to the address, within the segment, of the first byte of the QCT event stack.
   e. Event Limit (bytes 8-9): Set to the number of 4 byte event fields allocated in the QCT event stack. The limit value must be greater than or equal to four.
   f. SRC Descriptor (bytes hex 10-11): Initialized to hex 8000.
   g. SRC Limit (bytes hex 1A-1B): Initialized to hex 0001.
   h. Key (bytes hex 1C-1F): Initialized to the key value used to dequeue OREs (operation request elements) from the OUQ (operational unit queue).
3. The IOM may modify QCT entries only if the OU task is on the wait list of either the OUQ or the QCT–SRC (send/receive counter) of the OU task.

QCT EVENT STACK

The QCT (queue control table) event stack contains I/O event fields to be processed by the OU (operational unit) task. The event fields are placed on the stack by either the IOM (I/O manager) task or the event handler and are removed and processed by the OU task in a first in, first out manner.

The event offset field of the QCT provides the offset address to the beginning of the stack. The event limit, event count, and SRC count fields of the QCT are used by the event handler and OU task when posting or removing fields from the stack.

Although the event fields are normally placed on the QCT event stack by the event handler; an I/O manager task, under certain circumstances, can place an event field on the stack prior to dispatching the OU task.

The IOM must perform the following steps to place an entry on the stack or modify the QCT:

1. Set the lock bit in the control field of the QCT, to prevent the event handler from posting an event while the IOM is modifying the QCT fields.

2. Test for open entries in the event stack field of the QCT.

3. If the event count equals the event limit, reset the event count to hex 0000.

4. Place the 4-byte event field on the event stack at the address equal to the event offset + four times the event count.

5. Increment the event count field by one.

6. Reset the lock bit to zero.

After the above steps are completed, the OU task is dispatched to service the event by the IOM issuing a Send Count instruction to the send/receive counter in the QCT.

Programming Notes:

1. The QCT event stack must be aligned on a word address boundary and located in the same virtual = real resident segment as the QCT.
2. One QCT event stack must be allocated for each QCT.
3. The maximum size of the QCT event stack associated with the particular QCT is determined by the number of event fields that may be posted in the stack from the device or event handler. A minimum size of four is specified (event limit = hex 0004).
4. The maximum number of entries allocated for the QCT event stack is determined by the number of function events that may be posted to the OU task. The minimum number of entries allocated must be greater than or equal to four. If the stack is full (event count is equal to the event limit field) when the event handler attempts to place an event on the stack, the event handler sets the lock bit in the QCT, changes the event field to an error event, and posts the event field to the QCT event stack of the channel operational unit.

ADDRESS LIST ELEMENT

The ALE (address list element) is an 8 byte object containing a virtual or virtual = real address to be used during page chaining operations. The ALE is a single element in a page chain address stack used during the processing of a function operation block command.

The first ALE data address field is loaded into the data register specified in the QCT (queue control table) and the address of the ALE is placed in the QCT in the ALE pointer field. If the end of multiple load bit in the ALE control field is not set, the next ALE is processed. This next ALE data address field is loaded into the data register calculated by adding 1 to the value of the previous data register number. The address of the ALE is then placed in the QCT-ALE pointer-field in the QCT. ALEs are continually processed until the end of multiple load bit is encountered. A maximum of 14 I/O data registers may be loaded in this manner.

Format:

Type C3	Control	Data Address
0	1	2 Bytes 8

Bytes	Bits	Description
0		**Type**: ALE type code (hex C3).
1		**Control**: Control field.
	0	End Of List: 0 *Not* the last member of an ALE stack. 1 The last member of an ALE stack.
	1	End Of Multiple Load: The I/O resolved address registers starting with the primary data register specified in the QCT, are loaded with resolved addresses contained in the ALE list up to and including the ALE having the end of multiple load bit set. A maximum of 14 contiguous I/O resolved address registers can be loaded.
	2-7	Reserved: Must be zeros.
2-7		**Data Address**: A virtual or virtual = real address to be used during page chaining.

Programming Notes:

1. The address list element must be aligned on a doubleword address boundary.
2. The address list element stack must be located in a virtual = real segment. The virtual = real address of the first ALE in the stack is in bytes 2-7 of the function operation block using page chaining.
3. The location pointed to by the ALE data address must be aligned on a doubleword boundary.

I/O Storage Addressing

I/O RESOLVED ADDRESS REGISTERS

The I/O resolved address registers are hardware registers in the processor that contain resolved virtual addresses and are used by an I/O device to access command information, post command completion status, and to transfer device data to or from real storage.

The I/O resolved address register assignments (made at system specialization time) are passed to devices at initialization time, for example, when an active session is first established.

The number of I/O resolved address registers assigned to a particular device is variable up to a maximum of 15. A single device requires a command register to address the command/response field of the function operation block and one or more data registers for the transfer of device data.

The I/O resolved address register, used during a channel operation, is selected by the device. Addressability of a maximum of 256 address registers is provided by all models of the 5381 System Unit. Addressability of a maximum of 384 address registers is provided by all models of the 5382 System Unit.

Whenever a newly resolved virtual address is loaded into an I/O resolved address register, the use count in the primary directory entry (see Chapter 8) for that address is incremented. When the I/O resolved address register is invalidated, the use count is decremented.

Notes:

1. Unless loaded by FOB (function operation block) or an AOB (address operation block), an I/O resolved address register contains an address that may have been modified by the last instruction of an FOB. The modification is device dependent.
2. For those devices using multiple contiguous I/O data address registers, only the first (primary) data address register is uniquely specified in the queue control table for that OU (operational unit) task.

PAGE CHAINING

The page chaining facility of the I/O structure transfers device data, during the execution of a single function operation block, to several noncontiguous pages. The operation is referred to as page chaining, since the address used by the device during the storage transfers is changed only on page boundaries.

Page chaining is invoked during the OU task processing of an FOB. The set data address and page chaining control bits of the FOB must be set prior to the initiation of the OU task. When both bits are a 1, the address in the data address field of the FOB is a V=R address of the first ALE (address list element) of a page chain stack. This address is placed into the QCT (queue control table) for use by the event handler. (See the following diagram.)

The event handler accesses the QCT (queue control table) of the device to determine if page chaining is being used. The QCT contains the page chaining control bit set by the OU task and the ALE pointer field, which is updated (from the data address field of the FOB) during each boundary crossing, to point to the next ALE in the page chain stack.

The first ALE of the page chain stack is accessed by the OU task and the address in the ALE is resolved and loaded into the data address register of the device. Following resolution of the address in the ALE, which may be either virtual or V=R, a start device command is issued to the channel hardware.

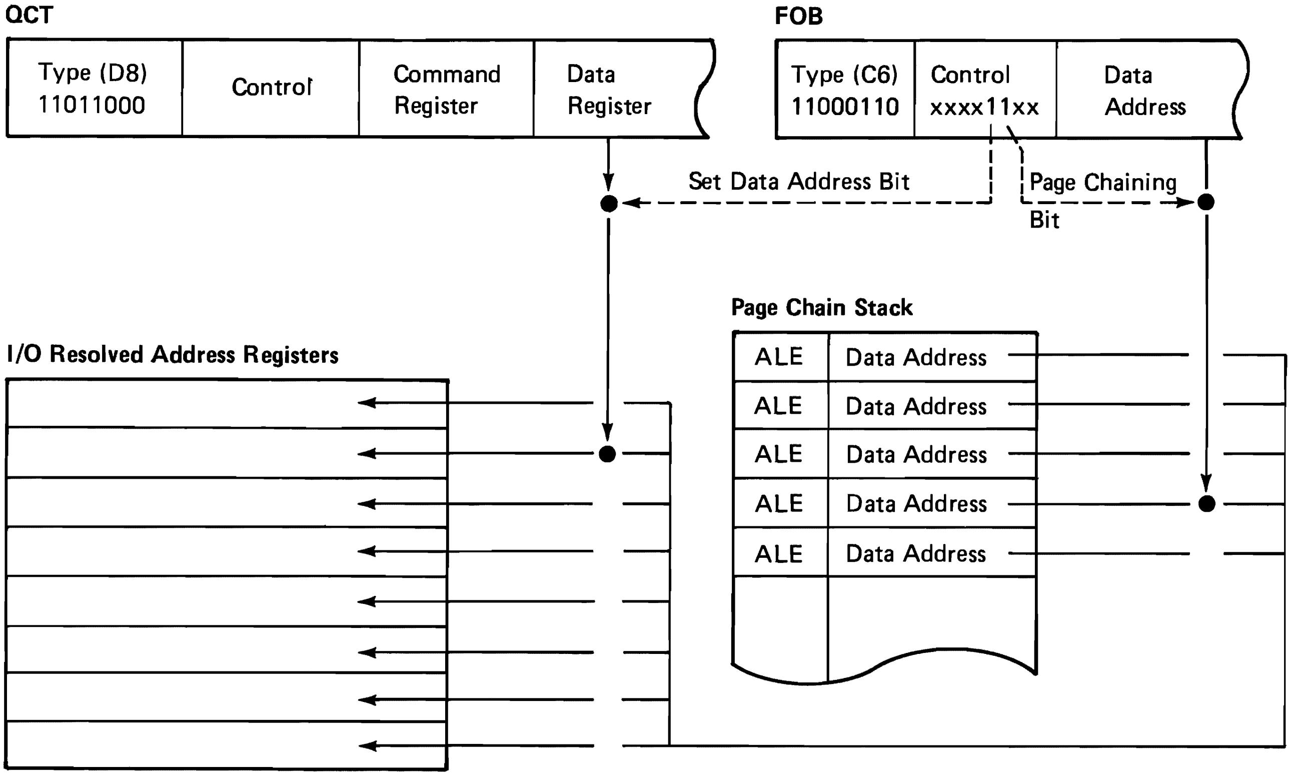
A page boundary crossing is always indicated by the VAT (virtual address translator) hardware if page chaining is used and the address contained in the register is modified across a page boundary.

The page boundary crossing is serviced by the event handler. The new address to be resolved is obtained from the next ALE in the page chain stack instead of using the address in the I/O resolved address register.

Notes:

1. The page chain must be in a V=R segment.
2. The data addresses contained in the address list element can be either virtual or V=R.
3. Page chaining still occurs at a page boundary even though the address in the ALE is V=R.

Page Chain Stacking



PAGE FAULTS

Page faults can occur during the modification of resolved virtual addresses in I/O resolved address registers during channel operations or during the resolution of virtual addresses contained in operation blocks.

If a resolved virtual address in an I/O resolved address register is incremented or decremented across a page boundary during channel operation, the I/O event handler attempts to resolve the address by reference to the primary directory. If the virtual address is not in the primary directory, the I/O event handler performs a send count to make the OU task dispatchable to resolve the page fault using the IMP exception mechanism.

A page fault occurring during the execution of an operation block by an OU task is also resolved through the IMP exception mechanism (address translation exception).

VIRTUAL = REAL

The IMP virtual address mechanism is used by I/O devices operating on the channel. Virtual addresses containing virtual = real SIDs (segment identifiers) can be used as I/O addresses in operation blocks.

Note: Page crossing and page chaining are handled by the event handler without need for a task switch.

I/O ADDRESSING RESTRICTIONS

Addresses of I/O objects have boundary and alignment restrictions (see the following table).

	I/O Object Address Restrictions		
I/O Object[1]	**Address Alignment**	**Address Type**	**Cross Page Boundary**
ORE (operation request element)	Doubleword	V=V, V=R	No[2]
OB (operation block)	Doubleword	V=V, V=R	No
OP (operation program)	Doubleword	V=V, V=R	Yes
QCT (queue control table)	Doubleword	V=R[3]	Yes
QCT Event Stack	Word	V=R[3]	Yes
ALE (address list element)	Doubleword	V=R	Yes
ALE Stack	Doubleword	V=R	Yes
I/O Event Stack	Word	V=V[4]	No
I/O Register Table	Halfword	V=R	No
I/O Event Stack	Word	V=V[4]	No
Data Address	Doubleword	V=V, V=R	Yes

[1] No object may cross an SID (segment identifier) boundary.

[2] The first 32 bytes may not cross a page boundary.

[3] Must be in the same V=R segment as the I/O register table.

[4] Must start and end on a page boundary, and the page must be pinned V=V.

I/O Events

An I/O event is a unit of work requested by the channel, device, or IOM (I/O manager) task of an OU (operational unit) task or the I/O event handler. This unit of work is described by 4 bytes called an I/O event field. The I/O event can be one of three types: function, address, or error.

I/O EVENT FIELDS

The general formats of the I/O event field for the three event types are:

Function Event

0000 ffff	Event-Dependent

0 1 Bytes 4

Legend: f = 4-bit function type code

Address Event

10e0 eeee	I/O Register Number	Hex 00	Hex 00

0 1 2 Bytes 4

Legend: e = event-dependent

Error Event

e1ee eeee	eeee eeee	eeee eeee	eeee eeee

0 1 2 Bytes 4

Legend: e = event-dependent

Function Event

The function event communicates device or IOM work requests to an OU task. The function event normally requested by an I/O device is the command completion indication (command end or command end/fetch next command). The function event used by the IOM task (fetch next command) is normally used to restart an OU task following an error situation.

Command End/Fetch Next Command

The command end/fetch next command function is requested by an I/O device at the completion of an FOB (function operation block) command. The function event, when requested by an I/O device, signifies that the current FOB has been successfully completed and the OU (operational unit) task can proceed to process operation blocks in the current ORE (operation request element) if they are available. If the operation blocks in the current ORE have been processed, then the OU task places the ORE on the IOMQ (I/O manager queue), the response queue, and requests (for example, receives) a new ORE from the OU queue.

The command end/fetch next command function event contains the BSTAT (basic status) information provided by the I/O device. The BSTAT information is placed into the OU status field of the ORE by the OU task.

Format:

Hex 01	IOREG	Basic Status

0 1 2 Bytes 4

Command End

The command end function event is used by an I/O device to communicate error or exception status to the device IOM task. The function request indicates that the device cannot proceed to execute commands until recovery operations are performed.

The command end event field contains the 2-byte BSTAT information provided by the device. The BSTAT information is placed into the OU status field in the current ORE before the ORE is enqueued on the IOMQ.

Format:

Hex 02	IOREG	Basic Status

0 1 2 Bytes 4

Fetch Next Command

The fetch next command function is normally used by an IOM task to restart the OU task following an error situation. The IOM task forms the event field, places the 4-byte field into the QCT (queue control table) event stack, and issues a Send Count instruction to the SRC (send/receive counter) in the QCT to cause the OU task to be dispatchable.

The fetch next command function causes the OU task to issue a Receive Message instruction to the OU queue to obtain a new ORE.

Format:

Hex 03	Hex 00	Hex 00	Hex 00

0 1 2 Bytes 4

Address Event

The address event indicates that a page boundary crossing occurred during the modification of a resolved virtual address contained in an I/O resolved address register. If address chaining is not being used by the device, the virtual address to be resolved is contained in the I/O resolved address register indicated by byte 1 of the field.

If address chaining is being used, the next virtual address from the address list will be resolved.

Format: Address events are not seen by the IMP channel interface; however, for completeness, the format of the address event is shown below:

10d0 dddd	I/O Register Number	Hex 00	Hex 00

0 1 2 Bytes 4

Legend: d = device-dependent

Error Event

The error event communicates error and/or exception conditions involving the channel hardware, interface, and specific conditions of devices to an IMP (IOM) channel error task. The channel IOM task performs logging and recovery operations and communicates with the OU tasks of the channel. Refer to *Channel Error Recovery* in this chapter for details.

Format:

e1ee eeee	eeee eeee	eeee eeee	eeee eeee

0 1 2 Bytes 4

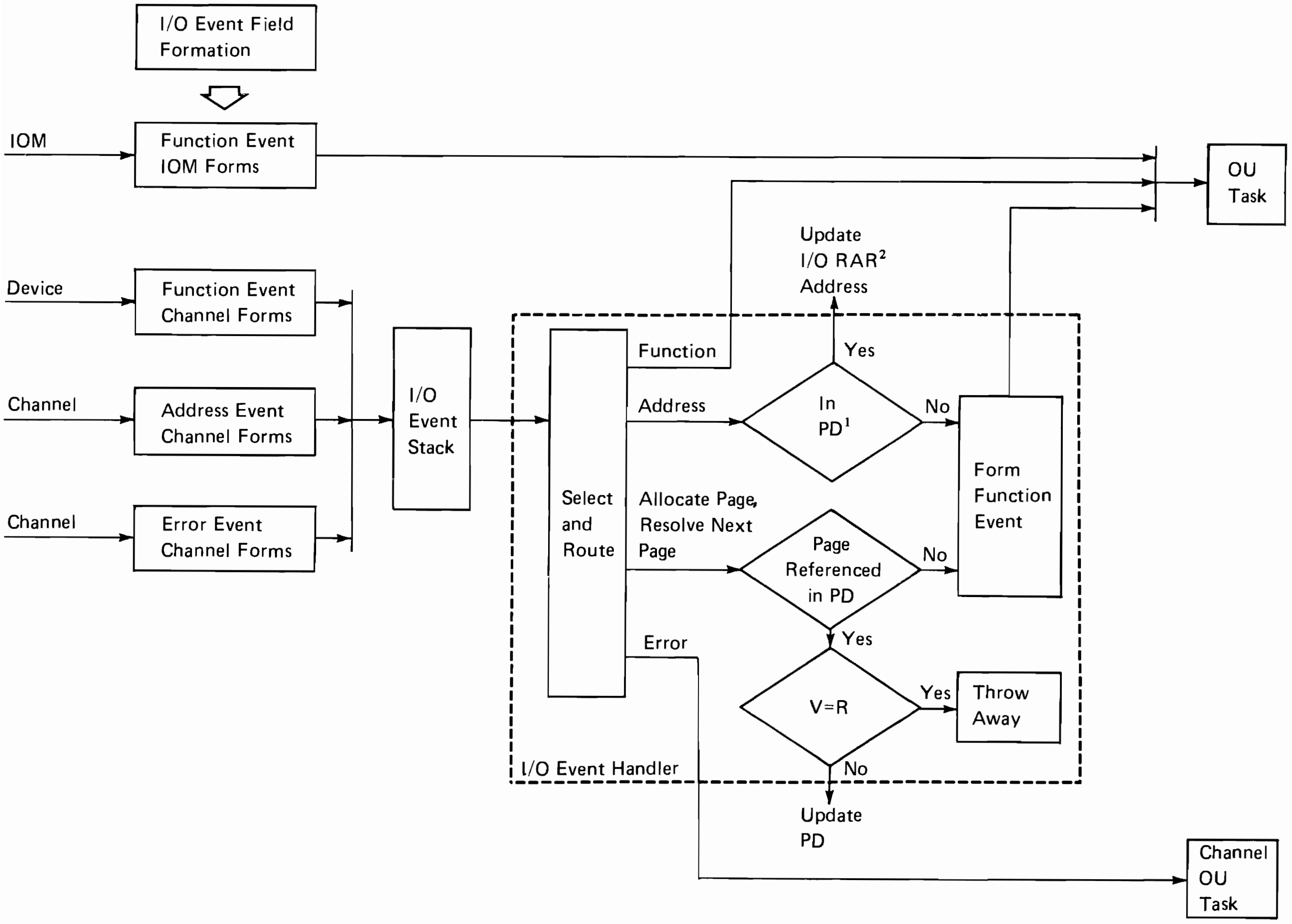
Legend: e = event-dependent

The other function events are described in the *Channel Theory-Maintenance* manual.

I/O EVENT HANDLER

The I/O event handler (Figure 7-4) is a horizontal microcode function that is invoked by the channel hardware to post an I/O channel event request to the processor. The horizontal microcode services the I/O events represented by the I/O event fields in the event stack.

Depending on the event field type, the event operation can be completely performed by the I/O event handler or the I/O event handler can send to an OU task to service the request. The I/O event handler relinquishes control when all entries in the event stack are removed and serviced.



[1] Primary directory
[2] Resolved address register

Figure 7-4. Event Handler Overview

I/O EVENT STACK

The I/O event stack is a list of contiguous 4-byte elements. The elements are I/O event fields that are placed on the stack by the channel hardware. The stack address used by the channel when placing an entry on the stack is in the event stack I/O resolved address register (hex 00).

The I/O event fields are removed from the event stack by the I/O event handler. The event stack I/O resolved address register is also used by the I/O resolved event handler when removing entries from the stack. In removing entries from the stack, priority is given to the address event class. Outside of this prioritization, events are removed on a last in, first out basis.

Programming Note: The event stack must start and end on a page boundary, must be pinned and V=V storage, and the page must be resident. This limits the stack to 128 entries. If the event stack overflows, a machine check will occur.

I/O REGISTER TABLE

The I/O resolved register table provides addressability to the various queue control tables in the machine. The table is used by the I/O event handler to locate the QCT (queue control table) of the device. The QCT must be located to place an I/O event field in the QCT event stack and to dispatch the OU (operational unit) task. Dispatching the OU task is done by issuing a Send Count instruction to the SRC (send receive counter) in the QCT.

The I/O register table contains a halfword (2-byte) entry for each I/O resolved address register. Each halfword entry contains an address offset into the segment to locate a QCT. Each device must have a QCT assigned. If a device uses multiple I/O resolved address registers, there are likewise multiple entries in the I/O register table. All of the multiple entries for a device contain the same offset, pointing to the same QCT. For example, if an I/O device is assigned a command register number of hex 10 (decimal 16) and a data register number of hex 11 (decimal 17), then halfword locations 16 and 17 of the I/O register table would contain the same offset so they would both point to the same QCT.

Programming Notes:

1. The I/O register table is aligned on a halfword address boundary and must be in the same V=R segment as all queue control tables. The address of the I/O register table is in the control address table (described in Chapter 2).
2. If an I/O resolved address register is not assigned to a device, then the corresponding entry in the table must contain hex FFFF. The event handler will then change the event to an error event and post it to the channel IOM.
3. The I/O register table must not cross a segment boundary.
4. I/O resolved address register 2 (hex 02) contains the address of the channel OU task QCT.

I/O Command Responses

INPUT/OUTPUT STATUS FIELDS

The types of I/O status information defined are:

- BSTAT (basic status)
- FSTAT (functional status)
- DSTAT (device status)

The read sense command (issued to the operational unit task) returns at least 2 bytes of DSTAT. Either additional read sense commands, or additional (more than two) DSTAT bytes, or both, can be defined to allow program access to all or additional DSTAT information.

The status information is contained in the description for each adapter or device. Only a general description is given here because the information is device dependent.

Basic Status

The BSTAT (basic status) consists of 2 bytes of adapter response data. The adapter response is provided to the channel along with the command end or command end/fetch next command indication. Status is stored in the I/O event stack by channel hardware and is moved to the OU (operational unit) status field of the ORE (operation request element) (operational unit) by the OU task (see Figure 7-7). Only the first byte of BSTAT is required; the second byte is optional.

BSTAT bytes are as follows:

Byte	Bit	Description
0	0-1	Reserved for the channel.
	2	Halt: 0 *No* device halt condition detected by the channel. 1 Device halt condition detected by the channel.
	3	Channel Error: 0 *No* error detected during channel transfers. 1 Error detected during channel transfers.
	4	I/O Exception: 0 *No* device exception condition detected. 1 Device exception condition detected. **Note:** I/O exceptions can be suppressed. An exception is suppressed when the condition that causes the exception does not inhibit setting the command complete bit (bit 7). A suppressed condition sets the I/O exception bit on. Suppression is a device option and may or may not be programmable. Examples of I/O exceptions that might be suppressed are last card on card units and incorrect length on tape units.

Bytes (Hex)	Bits	Description
0	5	Command Reject: 0 Command was acceptable for execution. 1 Operational unit is *not* designed for, or is in a state that prevents, command execution.
	6	I/O Error: 0 *No* error detected by the operational unit during execution of a function operation block. 1 Error detected by the operational unit during execution of a function operation block. *A read sense command must be issued to receive the device status bytes to determine the error condition.*
	7	Command Complete: 0 Command specified by an operation block has *not* begun, or not successfully completed execution. 1 Command specified by an operational block has successfully completed execution.
1		Optional and need not be supplied by the operational unit. This byte is set to zeros by the channel if not supplied by the operational unit.

Functional Status

The FSTAT (functional status) consists of from 1 to 4 bytes of operational unit information that can be required by the program for normal device operation, such as returning a record length from a tape unit.

The FSTAT is transferred into the response portion of the command response field (see Figure 7-7) of the FOB (function operation block) using the OU (operational unit) I/O command register.

The FSTAT is device-dependent and may be command-dependent. The FSTAT is also optional.

If the IOM (I/O manager) needs to interrogate FSTAT before the next operation block is executed, I/O exception must be set and command end must be indicated to the channel by the operational unit.

Device Status

The DSTAT (device status) consists of any number of bytes defined by a device for its status.

The DSTAT is transferred to the data field of the Read Sense command, using the operational unit primary data register. The bytes contain information required for proper device maintenance.

The DSTAT bytes common to all devices are defined as follows:

Byte	Description
0	Same as BSTAT byte 0 of previous FOB (function operation block) command executed by the adapter.
1	If BSTAT byte 1 is used by the operational unit, DSTAT byte 1 is the same as the BSTAT byte for the previous FOB. If BSTAT byte 1 is not used by the operational unit, DSTAT byte 1 is device dependent.

All remaining DSTAT bytes are device-dependent and optional.

Intervention-Required Signal

The I/O units that require operator intervention include one or more commands that indicate when the condition is cleared. Intervention-required conditions include:

- Printers
  - End of form
  - Forms jam

- Card Machines
  - Stacker full
  - Hopper empty

- Tape Units
  - Tape not mounted

An intervention-required condition is cleared when the operational unit can execute functional commands, such as, print, read card, read block, and so forth. The I/O error or I/O exception status is returned for all such functional commands when an intervention-required condition is present.

To test if the intervention-required condition has ended, a special command type is used. For example, a printer could use a return-ready command. This command notifies the program when an intervention-required condition, such as end of form, is cleared and the printer made ready. The I/O unit (printer and attachment) does not complete the command until the intervention-required (new forms loaded) condition is cleared and the printer made ready (by pressing the start key).

Device conditions other than intervention-required allow immediate execution of the return ready with I/O exception or I/O error status bits set. All printer commands other than return-ready immediately execute with the I/O exception or I/O error status bit set.

FUNCTION OPERATION BLOCK TIME-OUT

The FOB (function operation block) time-out capability of the system provides a mechanism for testing active I/O operations to determine if a channel end or device end interrupt has been outstanding for more than the period of time specified at system specialization.

The FOB time-out mechanism is implemented in the channel IOM (I/O manager) routines for all I/O devices except disks which have an implementation-dependent time-out mechanism. Fields included in each device QCT (queue control table) to maintain the information for FOB timing are:

Field	**Description**
Control	Byte 1, bit 2- FOB in progress
FOB Timer Count	Bytes hex 20 and 21-time elapsed for current FOB in n-second increments
FOB Timer Limit	Bytes hex 22 and 23- timing limit for FOB in in n-second increments

Operation

The following sequence describes the operation of the FOB time-out mechanism:

1. The device IOM program loads a value appropriate for the FOB to be executed into the QCT prior to sending the ORE (operation request element) to the OU (operational unit) task. The limit byte in the QCT provides a range of FOB time-out values with 1 to 255 timing intervals.

2. The OU task processes the ORE received from the IOM, loads the appropriate command and data registers, sets the FOB-in-progress bit in the QCT, and initiates the start-device sequence to the channel hardware.

3. The channel IOM, concurrent with the processes described above, tests all device QCTs, on n-second intervals for an active FOB-in-progress bit. The actions taken by the channel IOM when it detects an FOB-in-progress are shown below.

FOB-In-Progress	FOB Timing-In-Progress	Channel IOM Action
1	0	Resets count byte; sets FOB timing-in-progress bit
1	1	1. Increment count byte. 2. If count byte = limit byte, then send time-out MSG to device IOM; else EXIT.

4. The device OU task resets the FOB-in-progress and the FOB timing-in-progress bits in the QCT when the device posts a CE (command end) or a CE/FNC (fetch next command).

I/O EXAMPLE

Figure 7-5 (parts 1 through 14) depicts the sequence of events in an I/O operation under the assumption that the OB (operational block) contained in the ORE (operation request element) is an FOB (function operation block).

Figure 7-6 (parts 1 through 13) depicts the sequence of events in an I/O operation which is similar to that of Figure 7-5, but which makes use of the SENDMW instruction.

Note: The dotted arrows represent an action of the processor and the solid arrows represent a pointer.

With no I/O operations taking place, the IOM (I/O manager) task 1 is on the RDQ (task dispatching queue) ready to run and the OU task 2 is in the wait list of the OUQ (operational unit queue).

The QCT (queue control table) for this OU task was initialized at IPL (initial program load) time with the entries shown.

When the IOM task becomes the top priority task on the TDQ, the I/O operation starts.

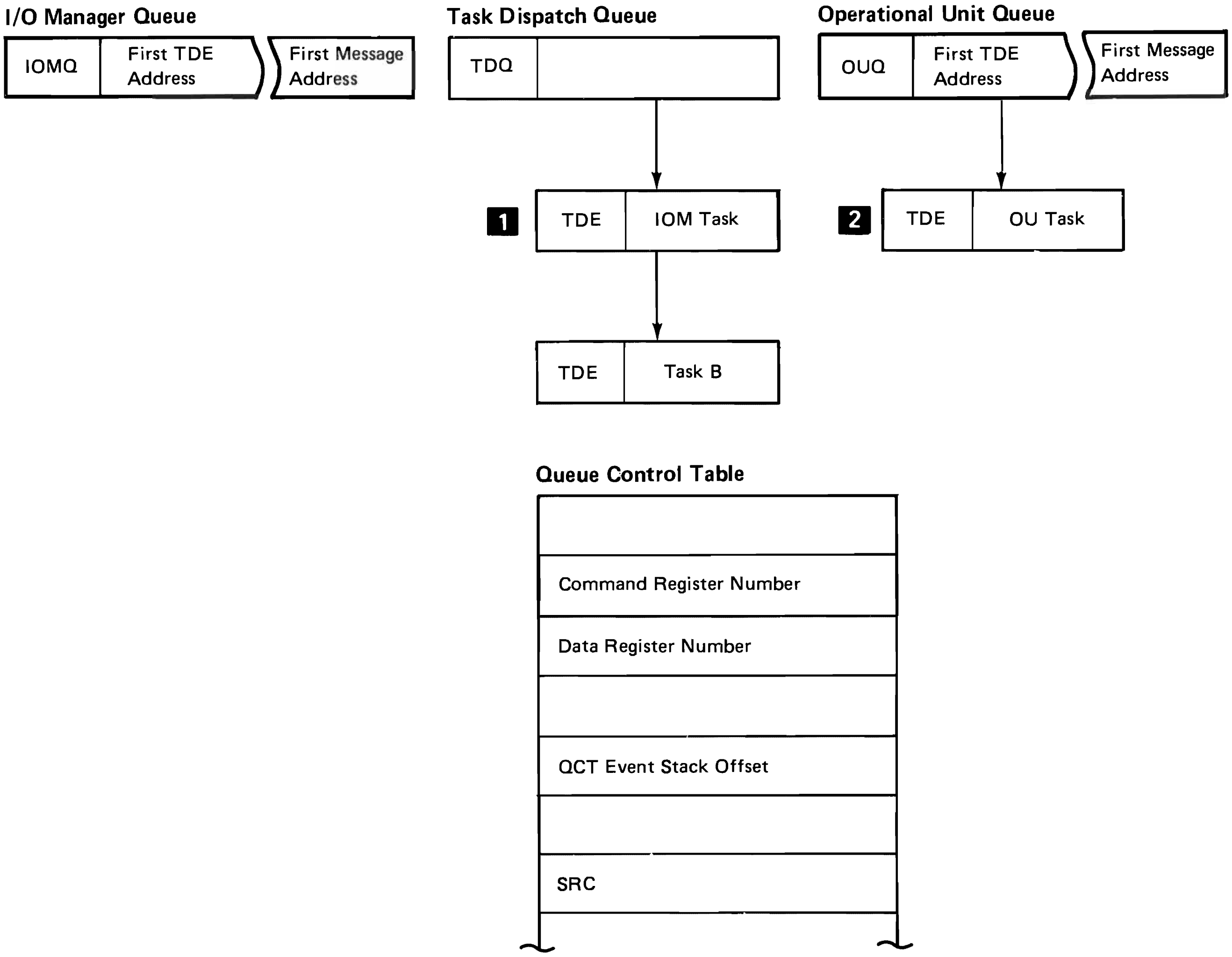


Figure 7-5 (Part 1 of 14). Sequence of I/O Operations

3 The IOM task issues a Send Message instruction to the OUQ.

4 The Send Message instruction puts an ORE on the message list.

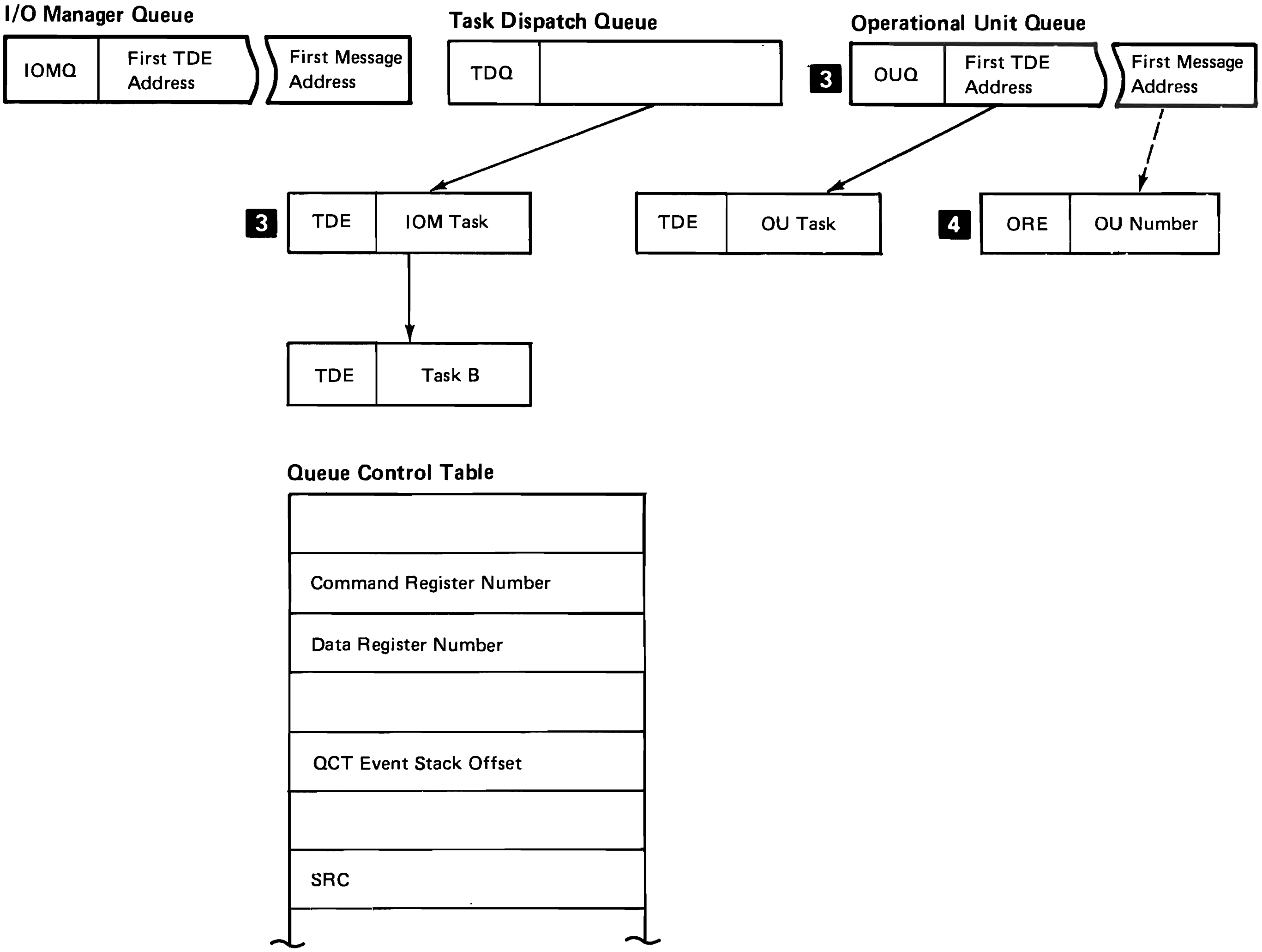


Figure 7-5 (Part 2 of 14). **Sequence of I/O Operations**

5 Because the OU task is on the wait list of the OUQ, the Send Message instruction puts the OU task on the task dispatch queue in priority sequence. When the OU task is dispatched, the OU task base registers are loaded with task control information resident in the OU TDE.

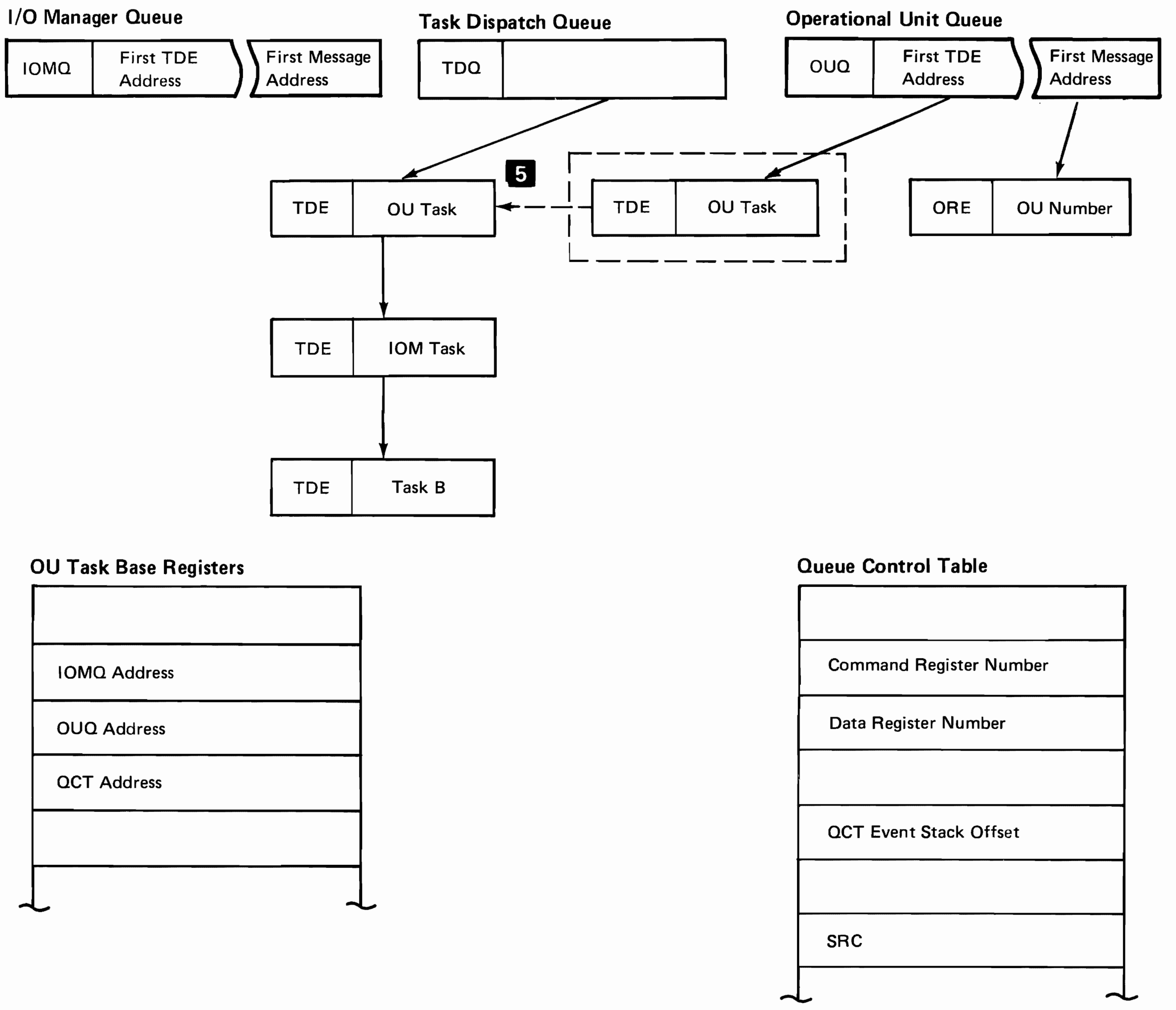


Figure 7-5 (Part 3 of 14). Sequence of I/O Operations

The OU task issues a Receive Message instruction 6 dequeuing the ORE from the OUQ 7.

The OUQ address, IOMQ address, and the QCT address are contained in the base register space 8 of the OU task.

9 The addresses of the executing ORE and the current OB (operation block) are stored in OU task base registers.

10 The OU task locates the QCT via an OU task base register entry.

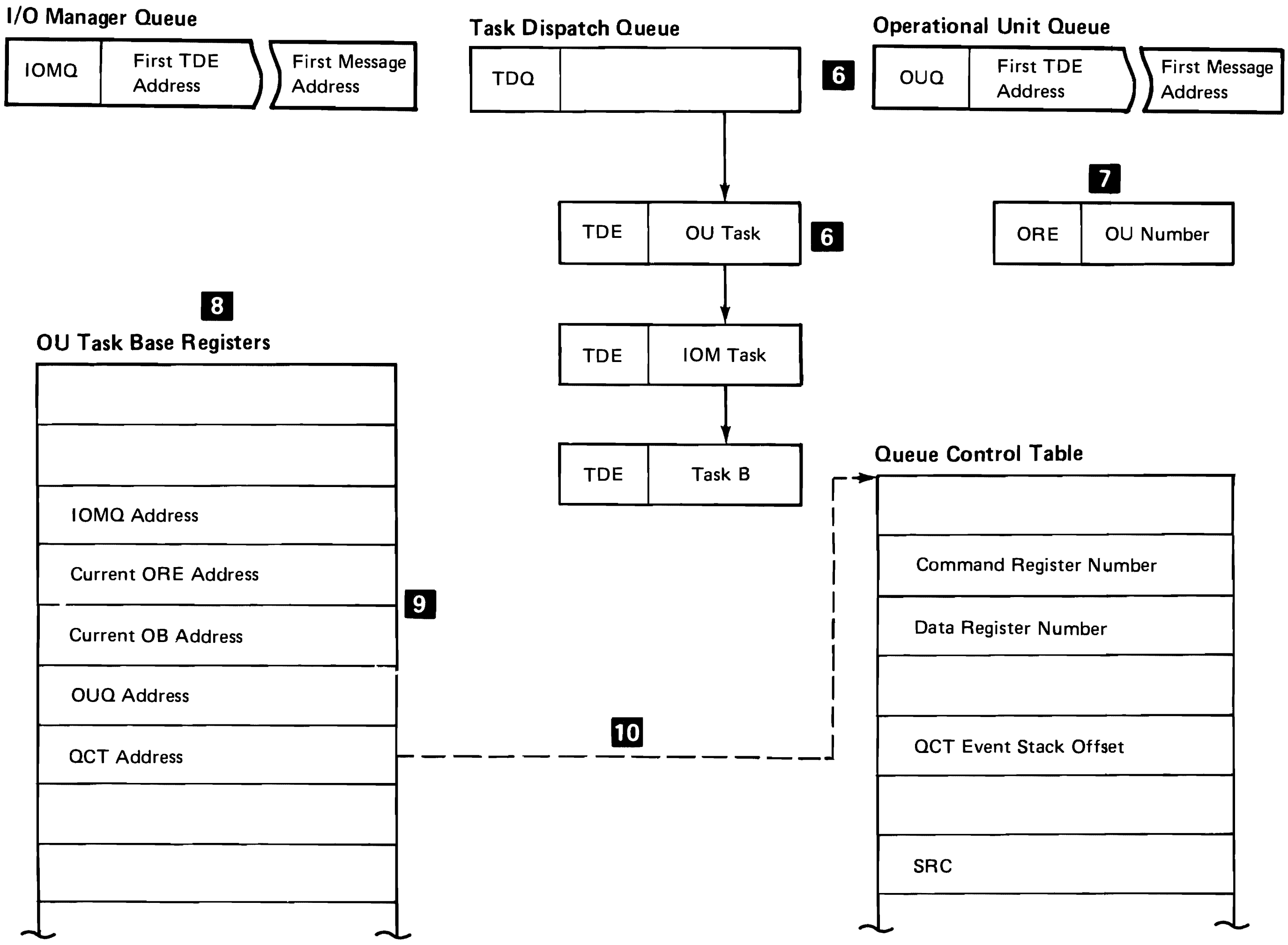


Figure 7-5 (Part 4 of 14). Sequence of I/O Operations

The OU task uses the command register number 11 to locate and load an I/O RAR (resolved address register) 12 with the resolved address of the command/response field (byte hex 18 of the ORE).

13 If required, an I/O RAR is located by the data register number and loaded with the address in bytes 2-7 of the operation block.

14 The OU task requests a start device sequence of the channel hardware. The device can now transfer additional command and data information without direct CPU involvement.

15 The OU task issues a Receive Count instruction to the SRC (send receive counter) in the QCT. The SRC is initialized with a count of zero and a limit of one.

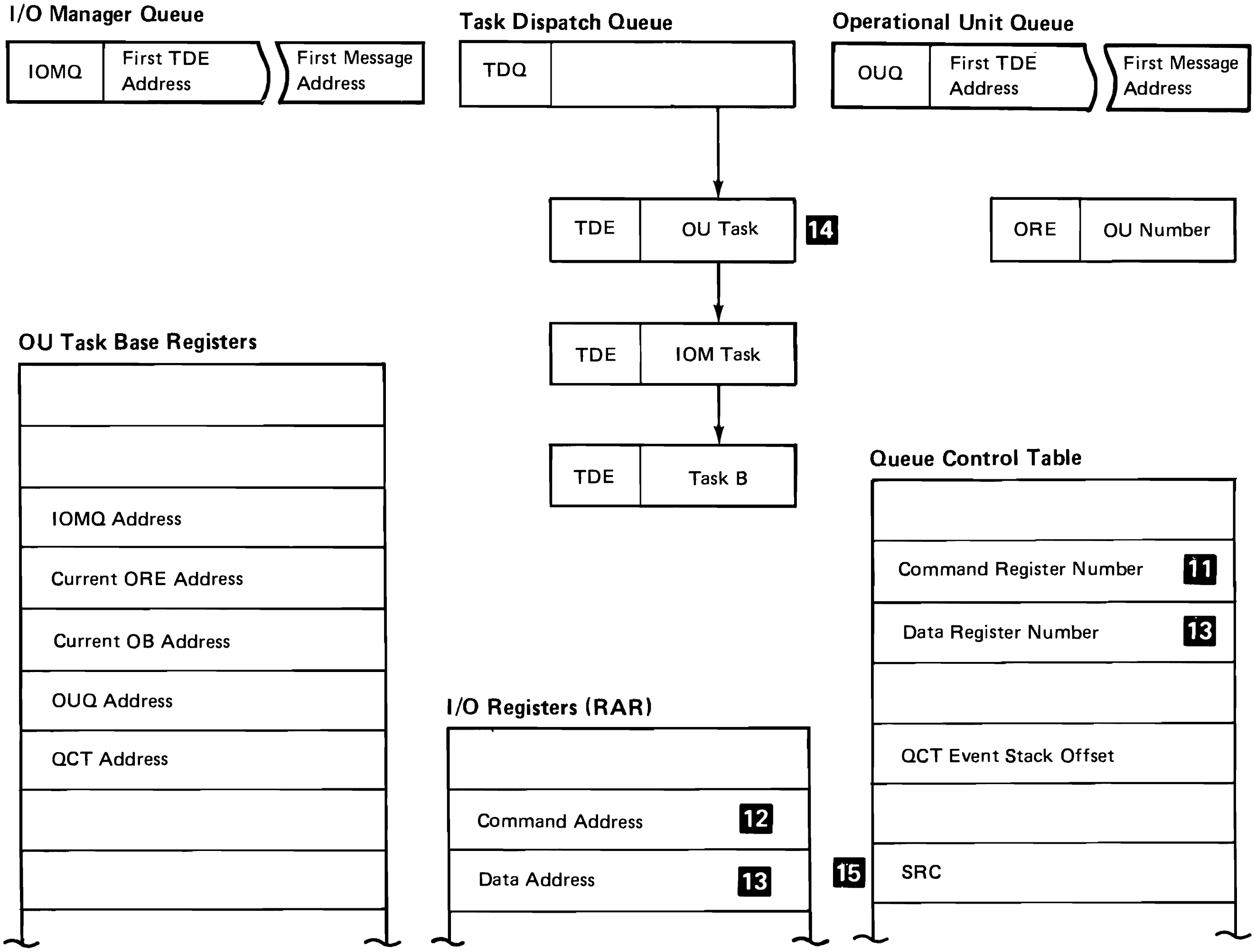


Figure 7-5 (Part 5 of 14). Sequence of I/O Operations

16 Because the SRC count is zero and less than the limit of one, the OU TDE is chained to the SRC, waiting for a command completion request from the I/O device.

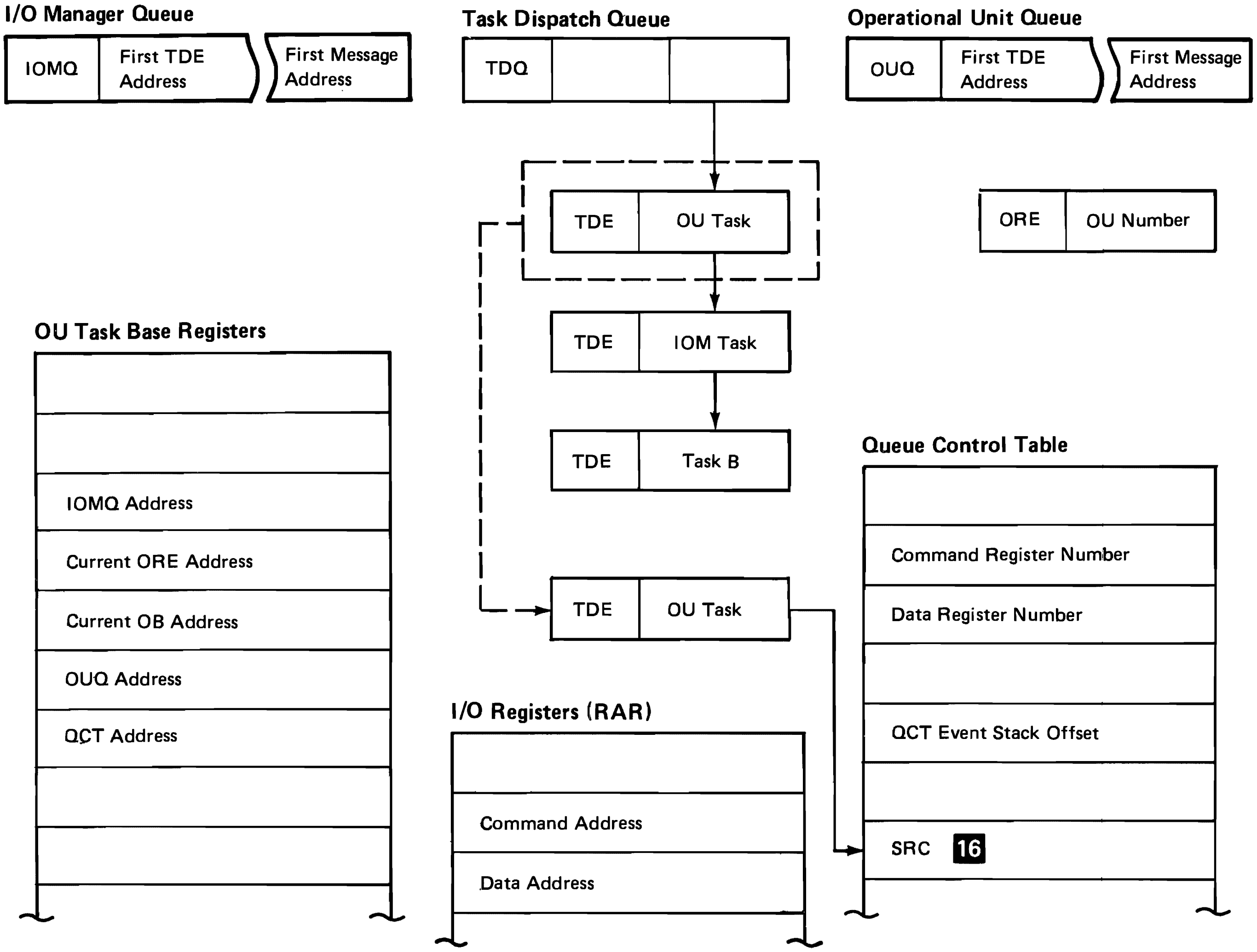


Figure 7-5 (Part 6 of 14). Sequence of I/O Operations

17 Control passes (task switch) to the task with the highest TDE priority. In this example, IOM task has priority so it resumes execution at the point following the Send Message instruction of the ORE to the OUQ.

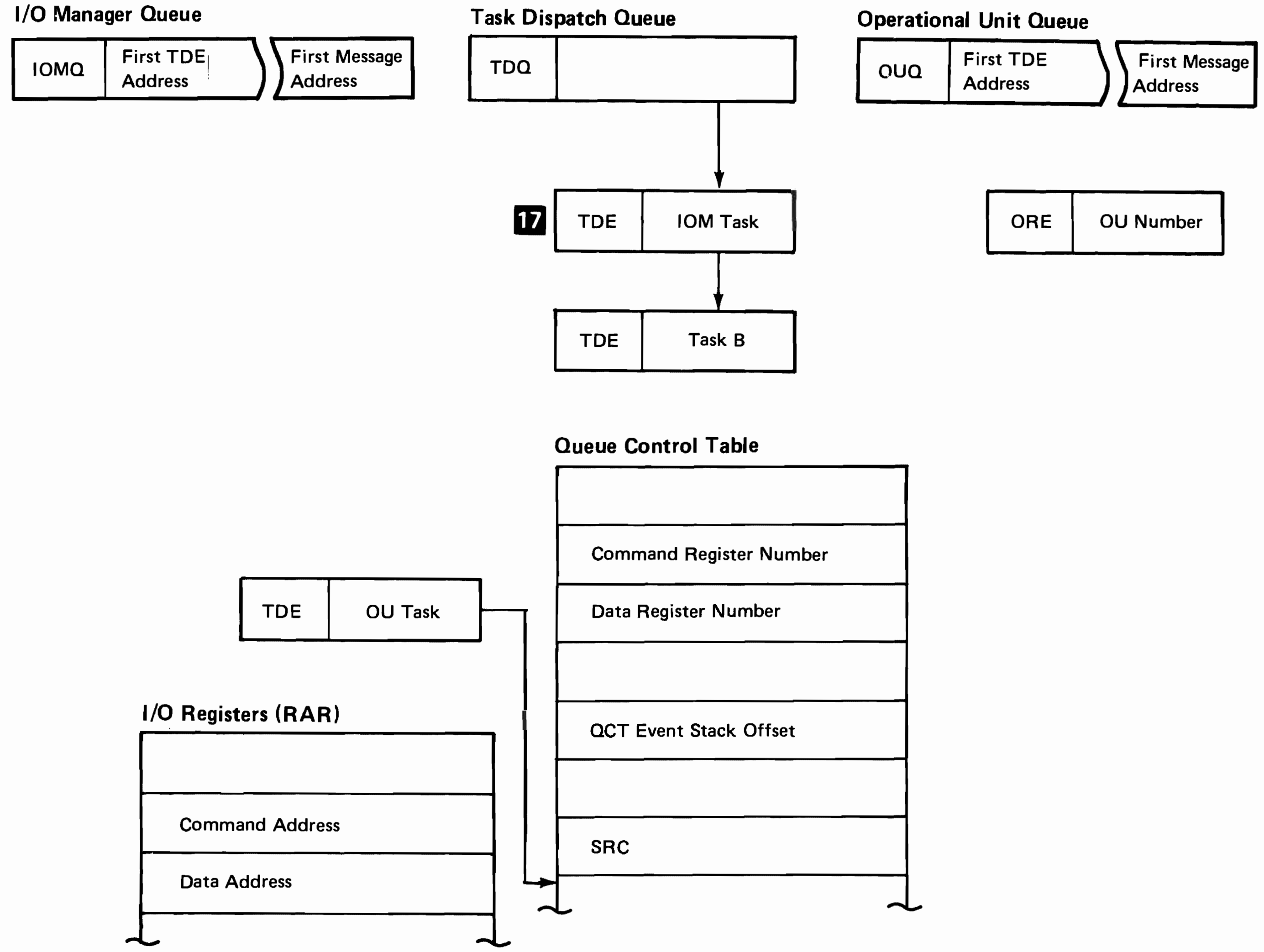


Figure 7-5 (Part 7 of 14). Sequence of I/O Operations

18 When the IOM task TDE reaches a point where it has to wait for the completion of the I/O command, the IOM task TDE issues a Receive Message instruction to the IOM queue.

19 Because there are no messages on the IOMQ message (msg list), the IOM task TDE is queued to the IOMQ wait list.

20 Control passes to the task with the highest priority. In this example, task B would begin execution.

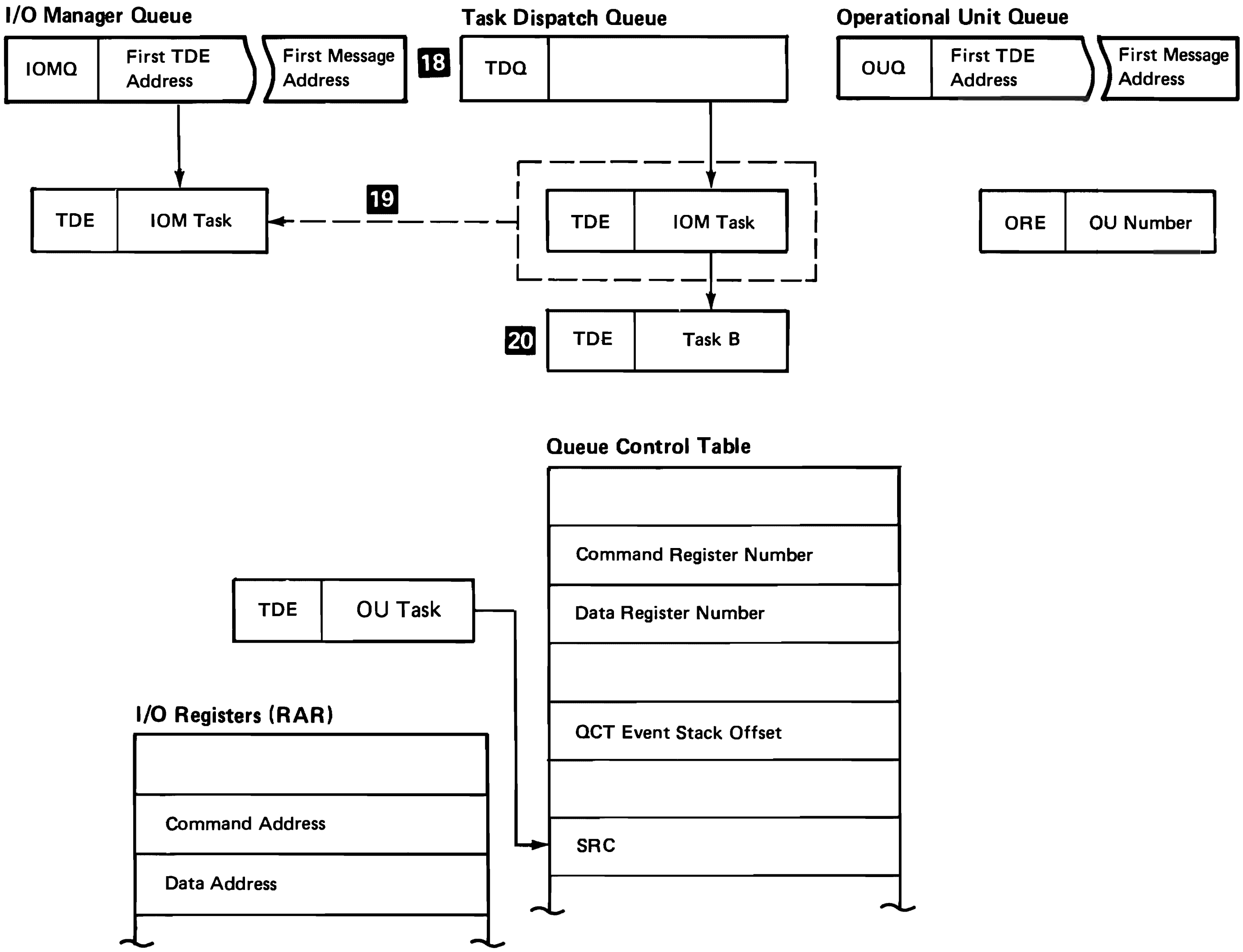


Figure 7-5 (Part 8 of 14). Sequence of I/O Operations

21 Upon completing the command, the device supplies command completion information to the channel. The channel uses this information to form a 4-byte I/O event field (function event type in this case) and place this field on the I/O event stack (see Figure 7-7). Channel hardware now signals an I/O channel event to the processor.

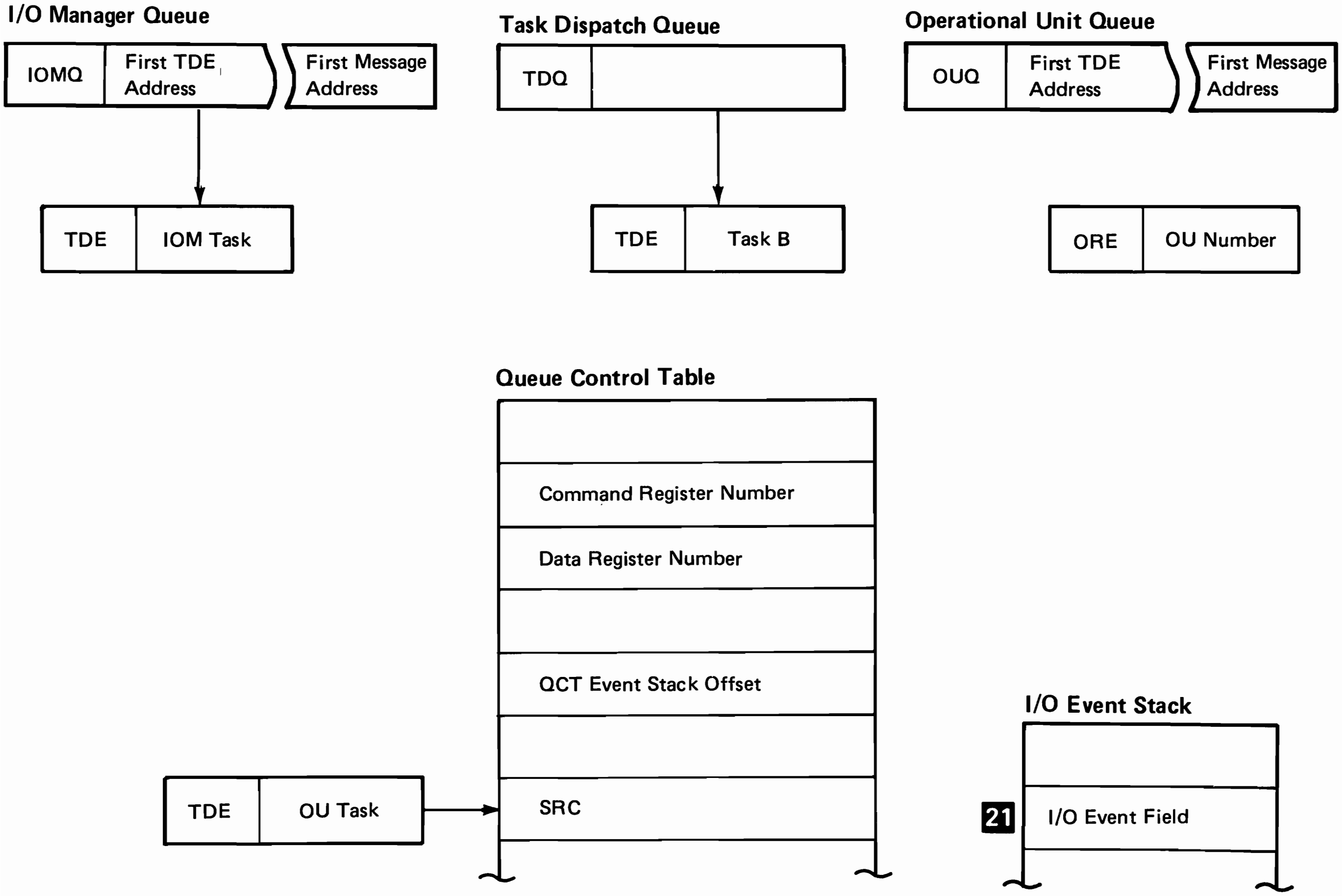


Figure 7-5 (Part 9 of 14). Sequence of I/O Operations

A channel HMC (horizontal microcode) routine, called the I/O event handler, executes when the processor accepts the I/O channel event. The event handler does not execute as a task (no task switch occurs).

22 The present task is temporarily suspended.

23 The I/O event handler accesses an event field in the I/O event stack.

24 The event handler accesses an I/O register table entry using the I/O register number in the I/O event field as an offset into the table.

25 The I/O register table entry (QCT offset) is used to locate an offset pointer in the OU QCT.

26 The QCT entry (QCT event stack offset) is used to locate an entry point in the OU QCT event stack.

27 The event handler moves the event field from the I/O event stack (see Figure 7-7) to the QCT event stack and increments the event count in the QCT.

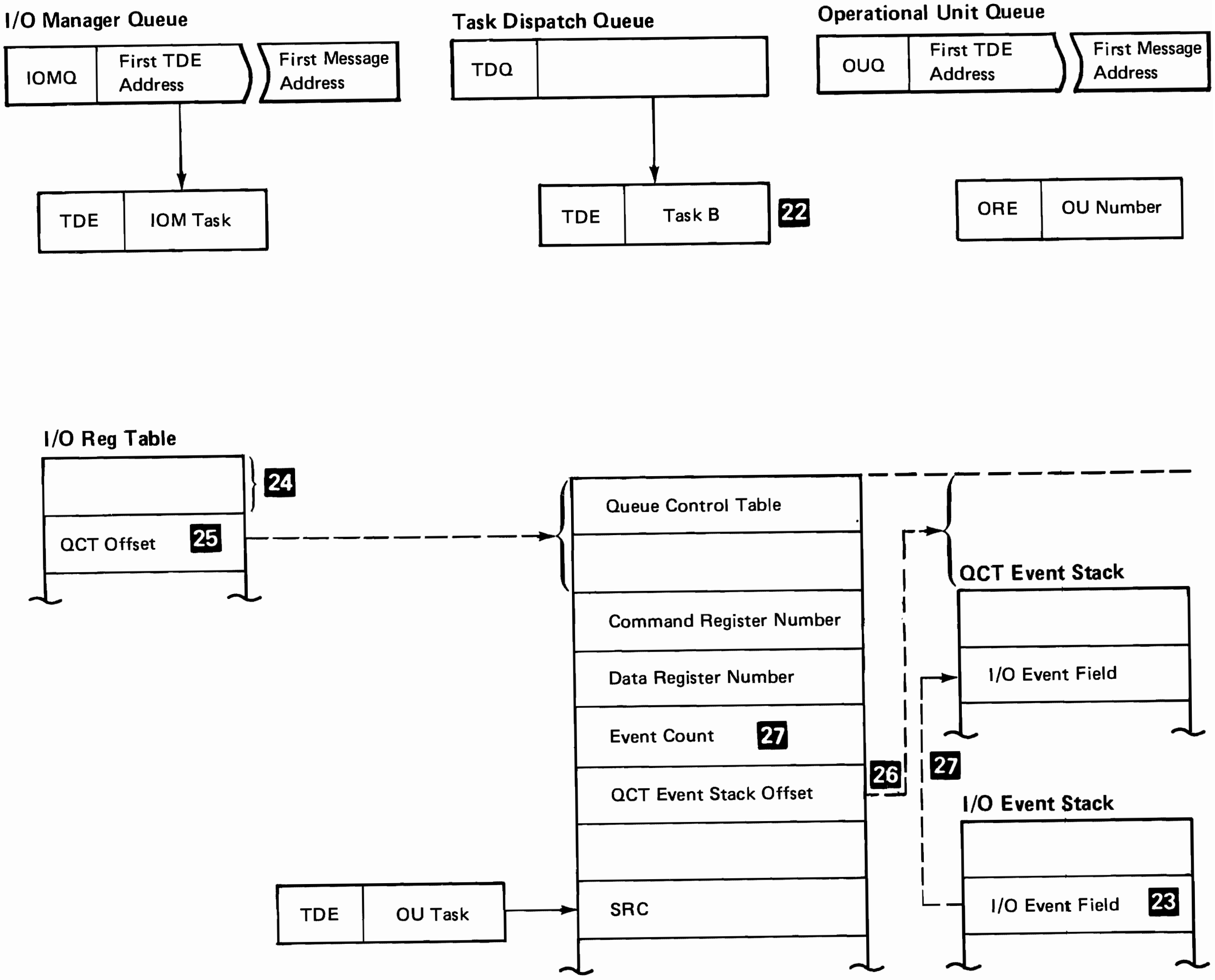


Figure 7-5 (Part 10 of 14). Sequence of I/O Operations

28 A Send Count instruction is issued by the event handler to the QCT SRC header. The Send Count instruction increments the SRC count.

29 The TDE of the OU Task is placed on the TDQ in priority sequence.

The event handler repeats the sequence (in this example, numbered 23-29) until all entries are removed from the I/O event stack.

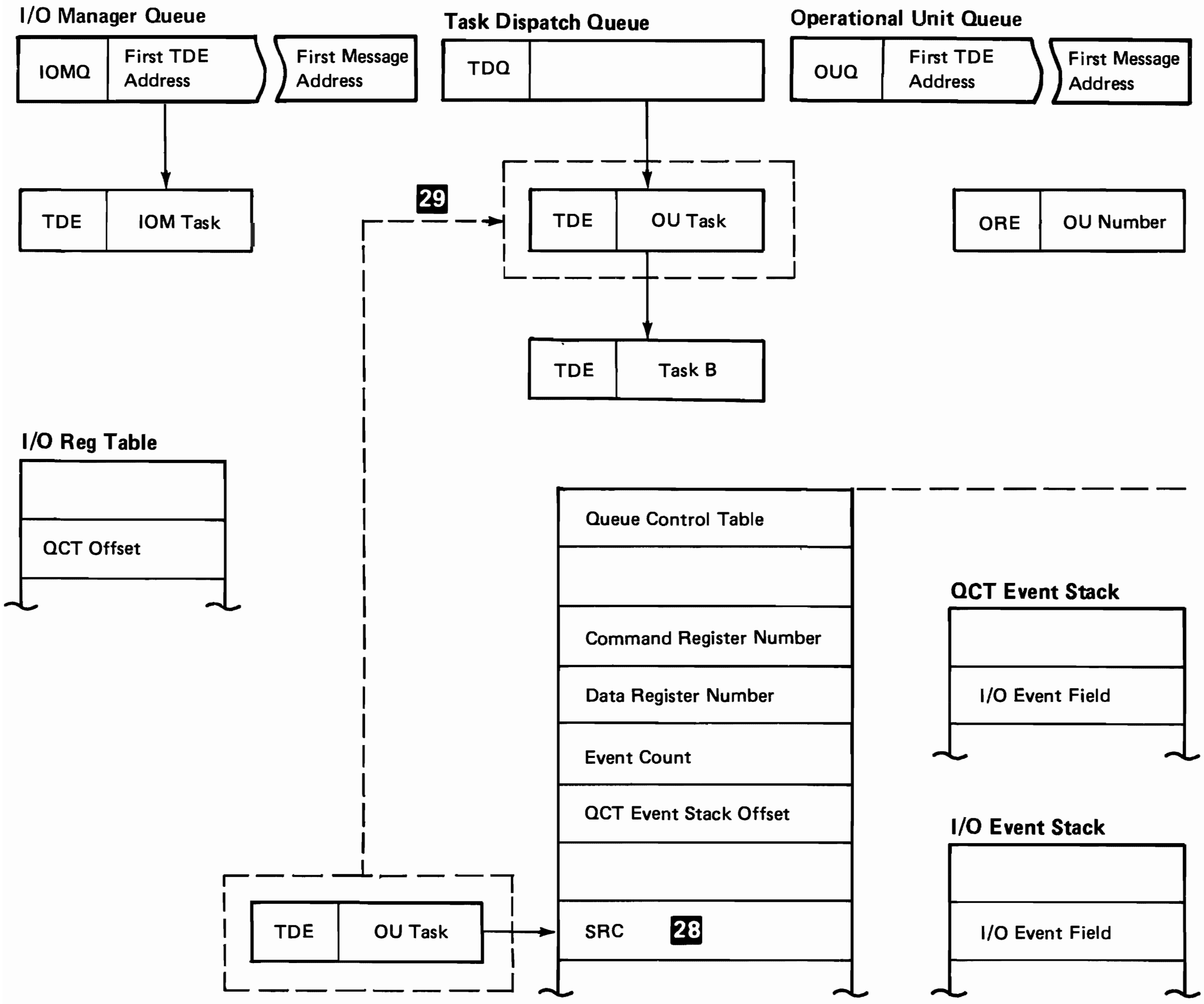


Figure 7-5 (Part 11 of 14). Sequence of I/O Operations

30 Control Passes to the TDE with the highest priority. In this example, the OU task begins execution.

31 The OU task issues a Receive Count instruction to the QCT SRC. This implicitly tests for waiting events in the QCT event stack.

32 The OU task locates the I/O event field and checks the channel command byte of the event field for the function event type.

33 When the function event type is a command completion indication, that is, if command end or command end/fetch next command is the function type, then the BSTAT field of the I/O event is moved to the OU status field of the ORE. (See Figure 7-7.)

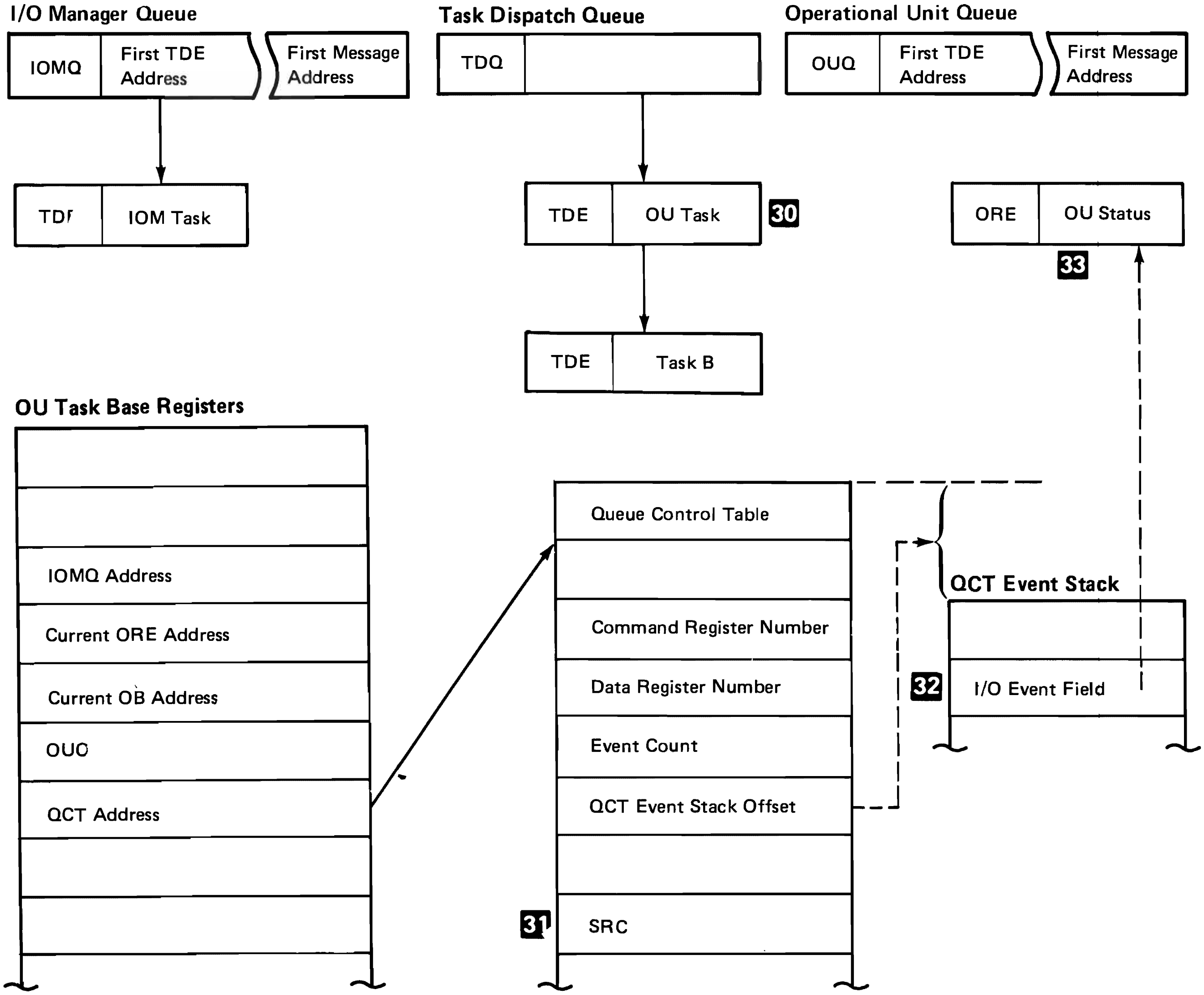


Figure 7-5 (Part 12 of 14). Sequence of I/O Operations

34 If command end is the command completion indication, the OU task issues a Send Message instruction to the IOMQ, placing the ORE on the IOMQ.

35 The OU task then issues a Receive Count instruction to the QCT SRC. The OU task now resides on the QCT SRC wait list, waiting for an IOM work request to be placed on the QCT event stack.

36 If (instead of 34 and 35) command end/fetch next command is the command completion indication, the OU task checks the current OB (operation block) for the last OB in ORE. When the current OB is not the last OB, the task processes the next OB. If the current OB is the last OB, the task issues a Send Message instruction to IOMQ, placing the current ORE on the IOMQ message list.

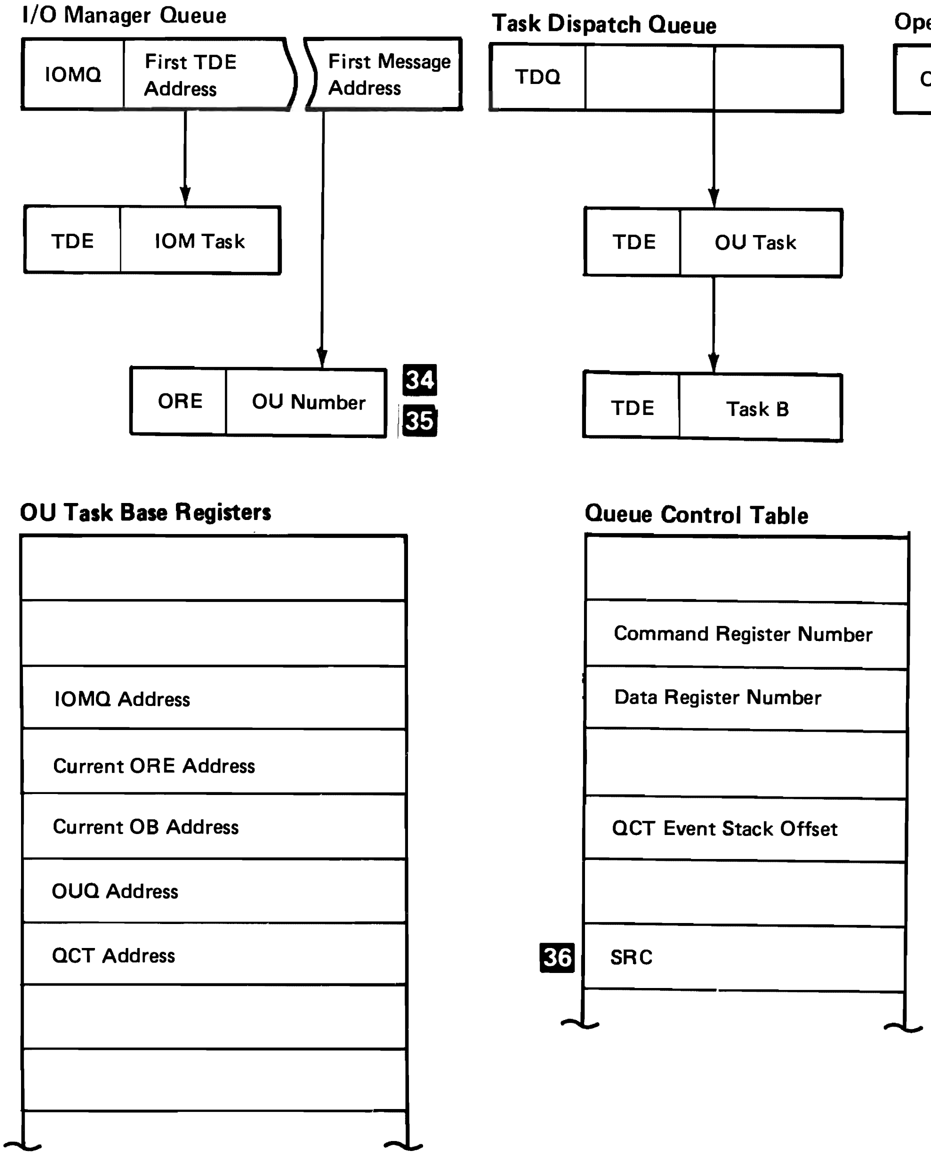


Figure 7-5 (Part 13 of 14). Sequence of I/O Operations

37 Because the TDE of the IOM task is on the IOMQ wait list, the Send Message instruction to the IOMQ places the IOM task on the TDQ in priority sequence.

38 The OU task now issues a Receive Message instruction to the OUQ. With no messages (OREs) queued to the OUQ message list, the OU task is dequeued from the TDQ and placed on the OUQ wait list.

39 The IOM task resumes execution (task switch) if it is of higher priority than task B.

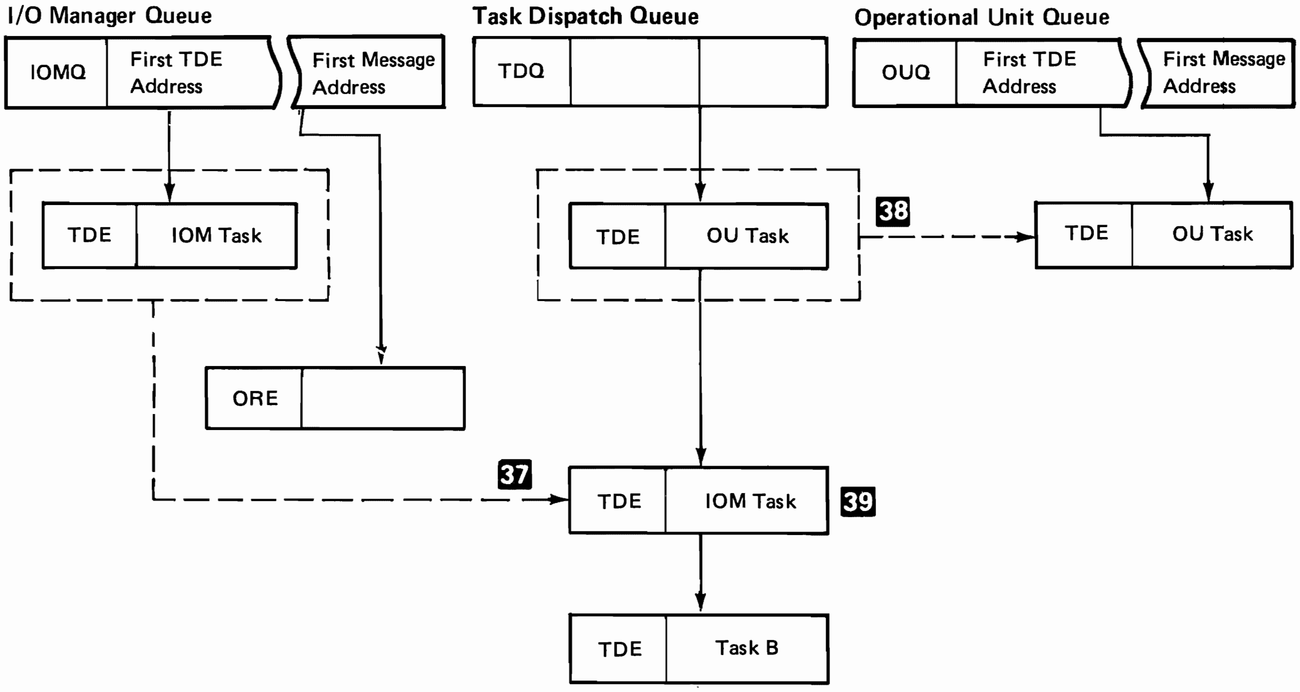


Figure 7-5 (Part 14 of 14). Sequence of I/O Operations

With no I/O operations taking place, the IOM (I/O manager) task 1 is on the TDQ (task dispatch queue) ready to run, and the OU task 2 is on the wait list of the OUQ (operational unit queue).

The QCT (queue control table) for this OU task was initialized at IPL (initial program load) time with the entries shown.

When the IOM task becomes the top priority task on the TDQ, the I/O operation starts.

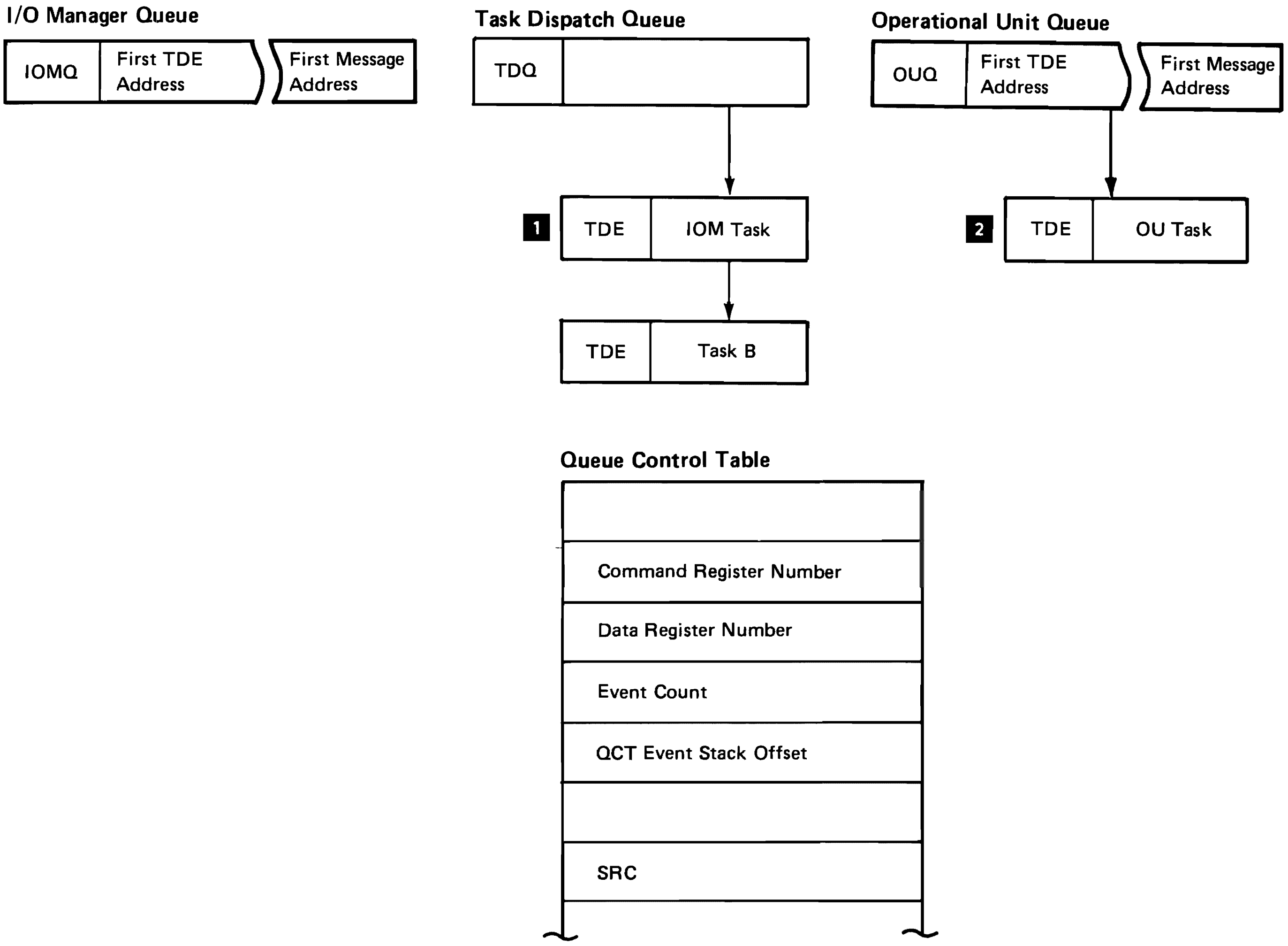


Figure 7-6 (Part 1 of 13). Sequence of I/O Operations with SENDMW Instruction

3 The IOM task issues a Send Message and Wait instruction to the OUQ.

4 The Send Message and Wait instruction puts an ORE on the message list of the OUQ and removes the current TDE from the TDQ.

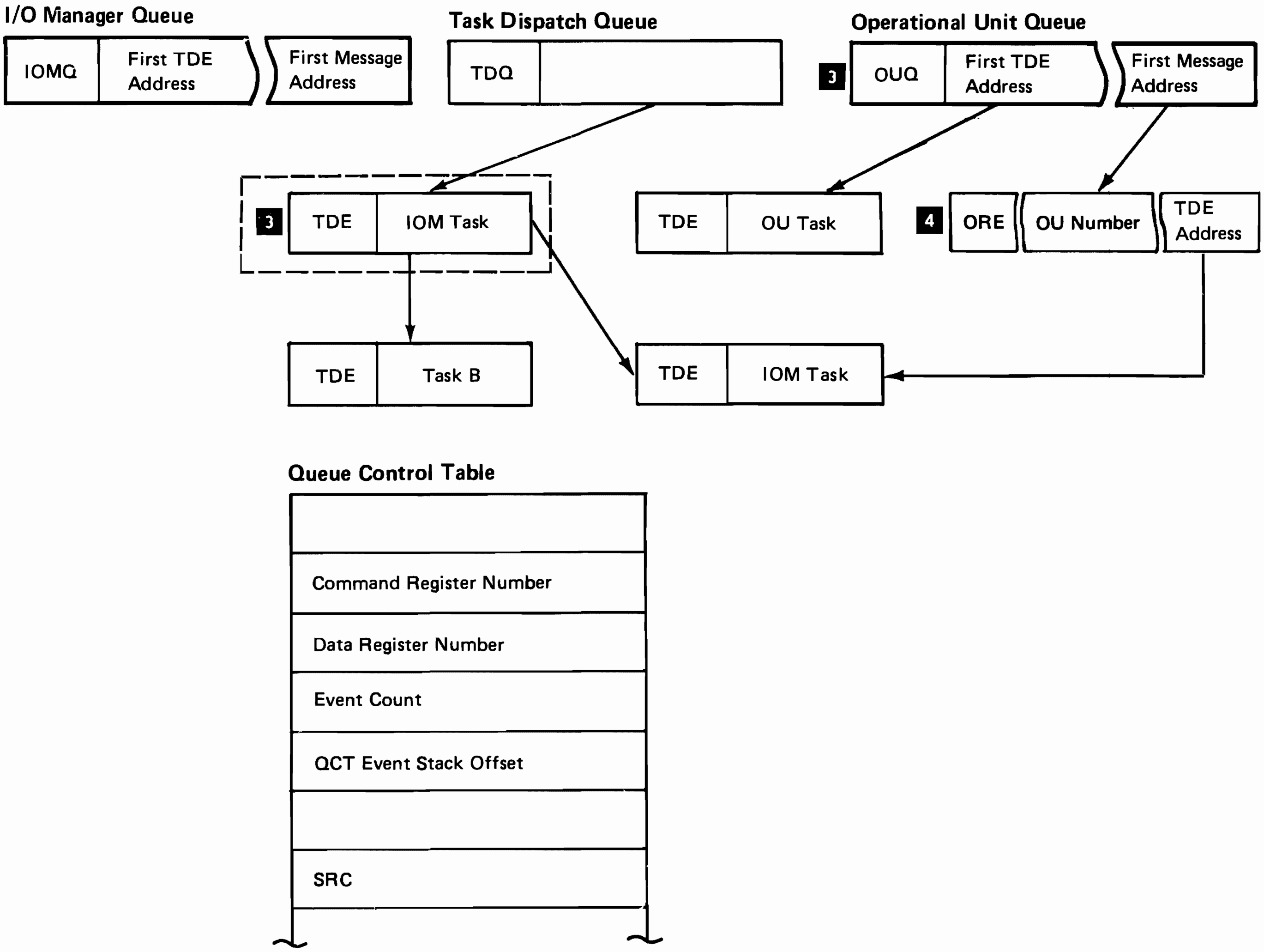


Figure 7-6 (Part 2 of 13). Sequence of I/O Operations with SENDMW Instruction

5 Because the OU task is on the wait list of the OUQ, the Send Message and Wait instruction puts the OU task on the task dispatch queue in priority sequence. When the OU task is dispatched, the OU task base registers are loaded with task control information resident in the OU TDE.

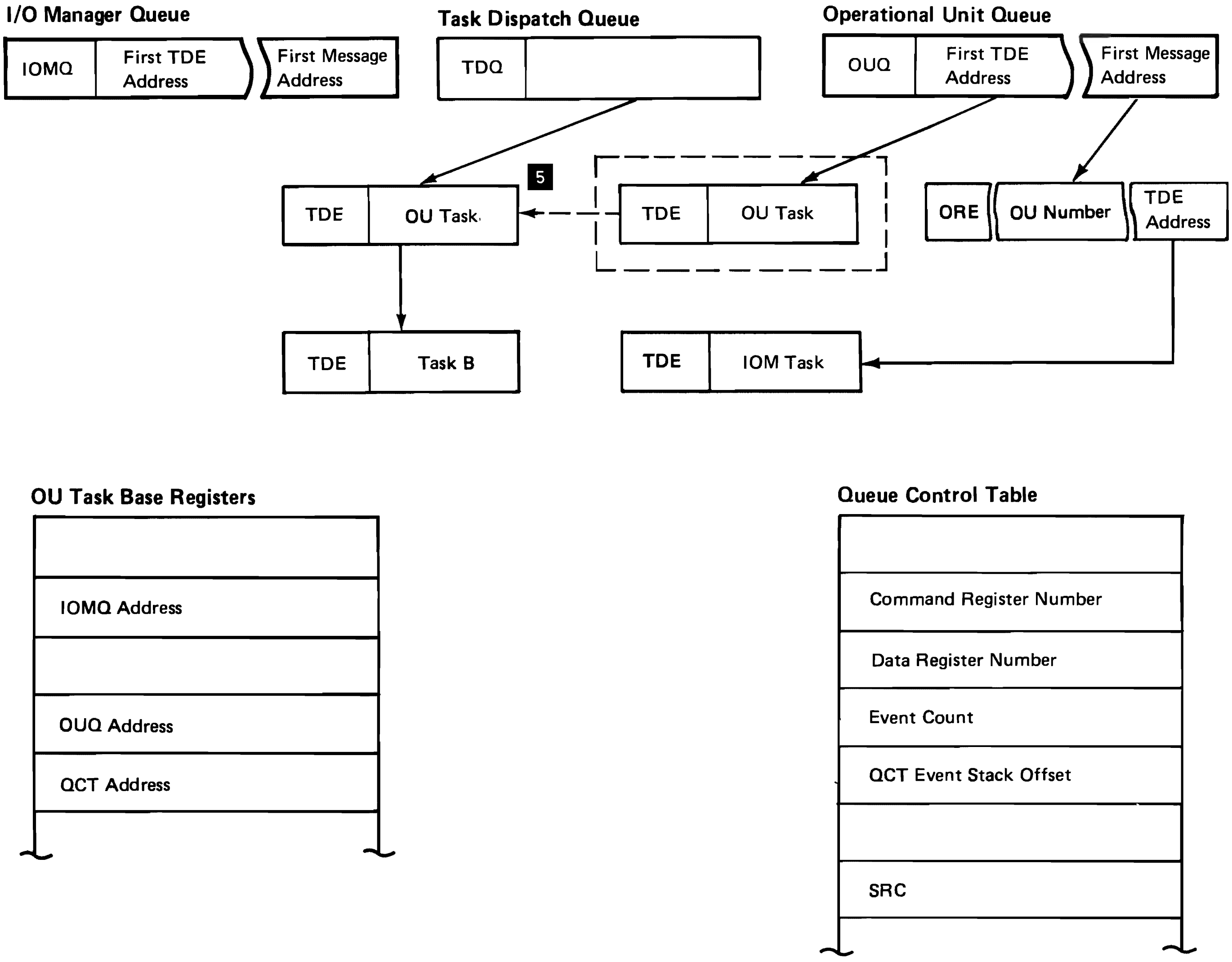


Figure 7-6 (Part 3 of 13). Sequence of I/O Operations with SENDMW Instruction

The OU task issues a Receive Message instruction 6 dequeing the ORE from the OUQ 7.

The OUQ address, IOMQ address, and the QCT address are contained in the base register space 8 of the OU task.

9 The addresses of the executing ORE and the current OB (operation block) are stored in the OU task base registers.

10 The OU task locates the QCT via an OU task base register entry.

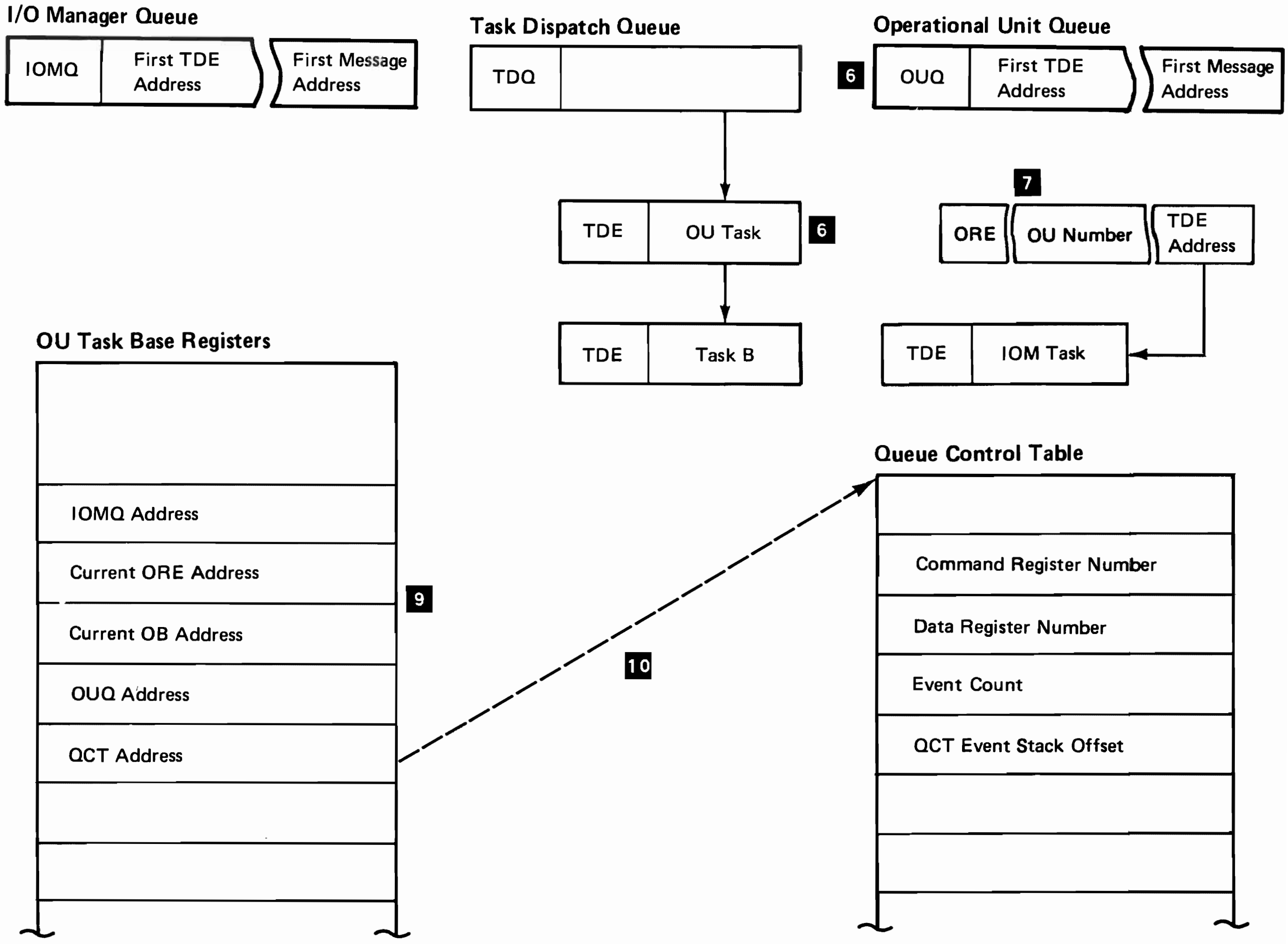


Figure 7-6 (Part 4 of 13). Sequence of I/O Operations with SENDMW Instruction

The OU task uses the command register number 11 to locate and load an I/O RAR (resolved address register) 12 with the resolved address of the command/response field (byte hex 18 of the ORE).

13 If required an I/O RAR is located by the data register number and loaded with the address in bytes 2-7 of the operation block.

14 The OU task requests a start device sequence of the channel hardware. The device can now transfer additional command and data information without direct CPU involvement.

15 The OU task issues a Receive Count instruction to the SRC (send receive counter) in the QCT. The SRC is initialized with a count of zero and a limit of one.

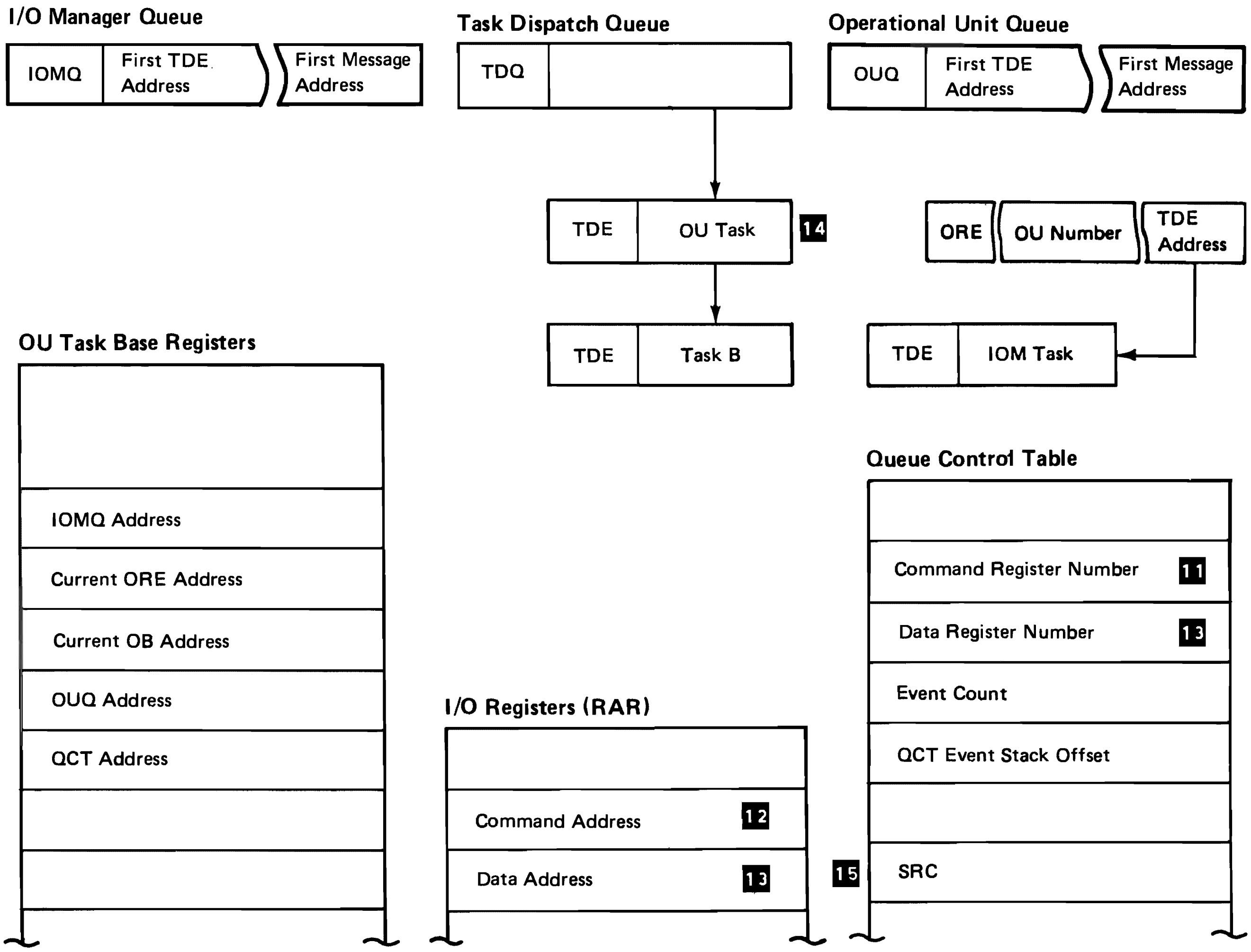


Figure 7-6 (Part 5 of 13). Sequence of I/O operations with SENDMW Instruction

16 Because the SRC count is zero and less than the limit of one, the OU TDE is chained to the SRC, waiting for a command completion request from the I/O device.

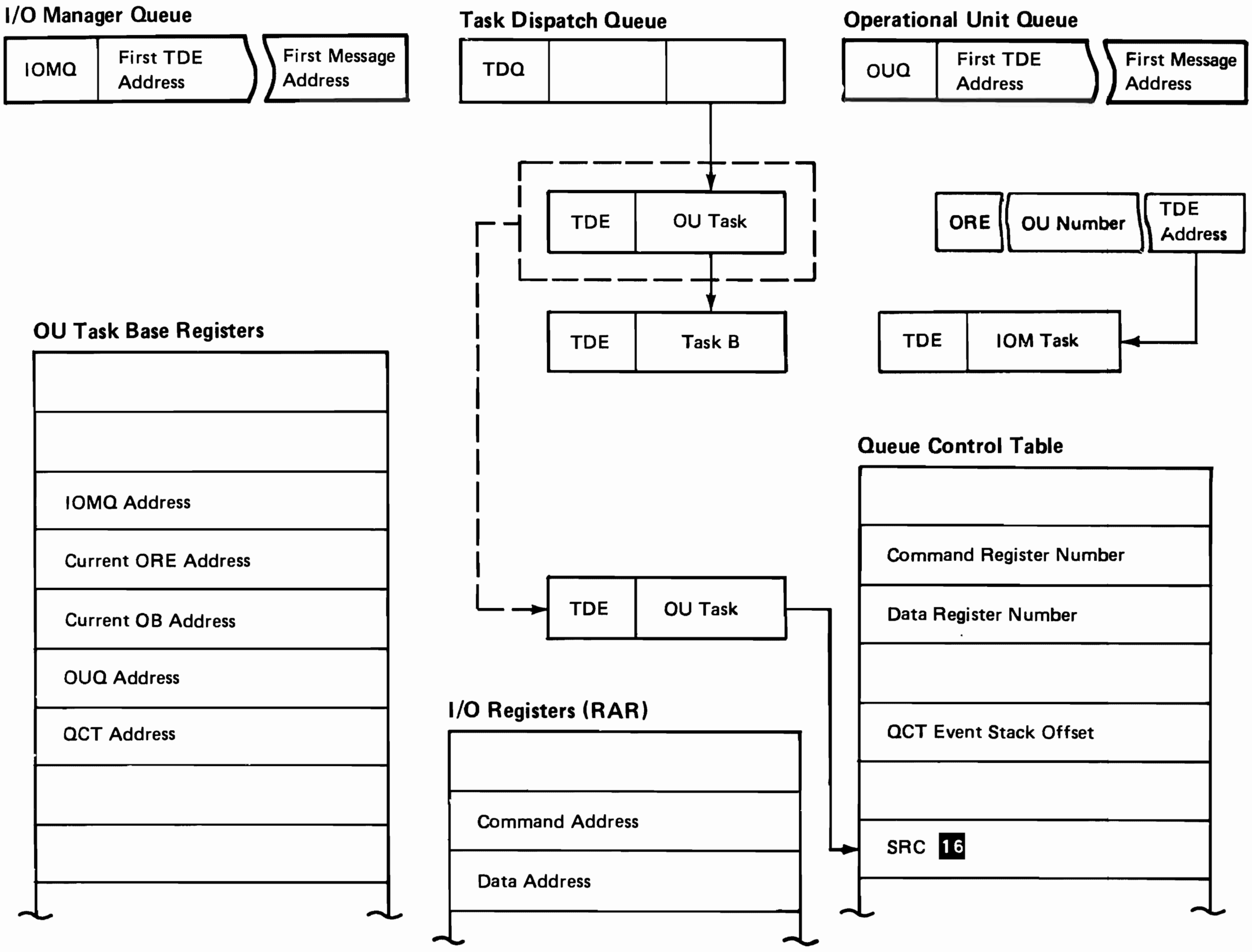


Figure 7-6 (Part 6 of 13). Sequence of I/O Operations with SENDMW Instruction

17 Control passes (task switch) to the task with the highest TDE priority. In this example, task B would begin execution.

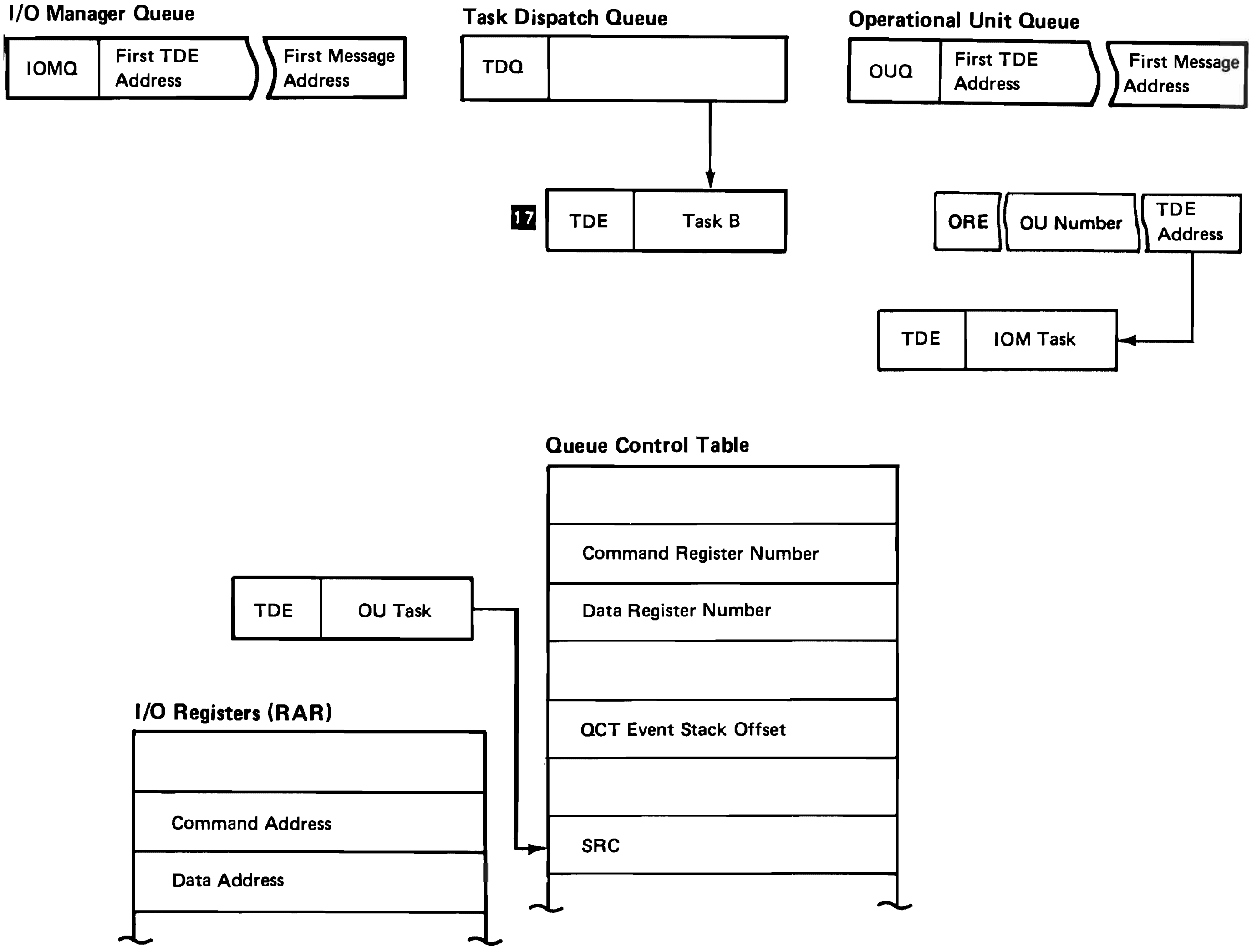


Figure 7-6 (Part 7 of 13). Sequence of I/O Operations with SENDMW Instruction

18 Upon completing the command, the device supplies command completion information to the channel. The channel uses this information to form a 4-byte I/O event field (function event type in this case) and places this field on the I/O event stack (see Figure 7-7). Channel hardware now signals an I/O channel event to the processor.

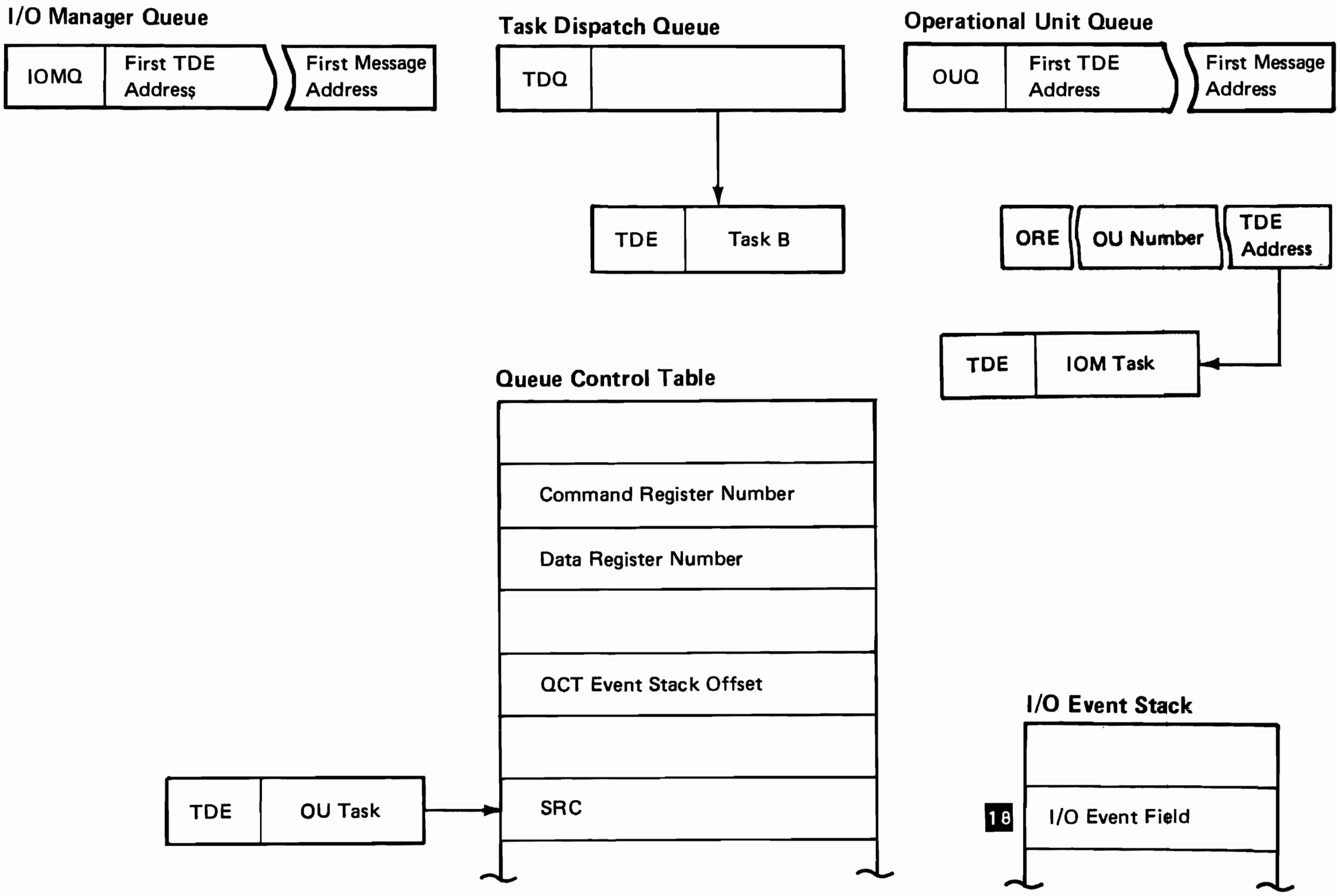


Figure 7-6 (Part 8 of 13). Sequence of I/O Operations with SENDMW Instructions

A channel HMC (horizontal microcode) routine, called the I/O event handler, executes when the processor accepts the I/O event. The event handler does not execute as a task (no task switch occurs).

19 The present task is temporarily suspended.

20 The I/O event handler accesses an event field in the I/O event stack.

21 The event handler accesses an I/O register table entry using the I/O register number in the I/O event field as an offset into the table.

22 The I/O register table entry (QCT offset) is used to locate an offset pointer in the OU QCT.

23 The QCT entry (QCT event stack offset) is used to locate an entry point in the OU QCT event stack.

24 The event handler moves the event field from the I/O event stack (see Figure 7-7) to the QCT event stack and increments the event count in the QCT.

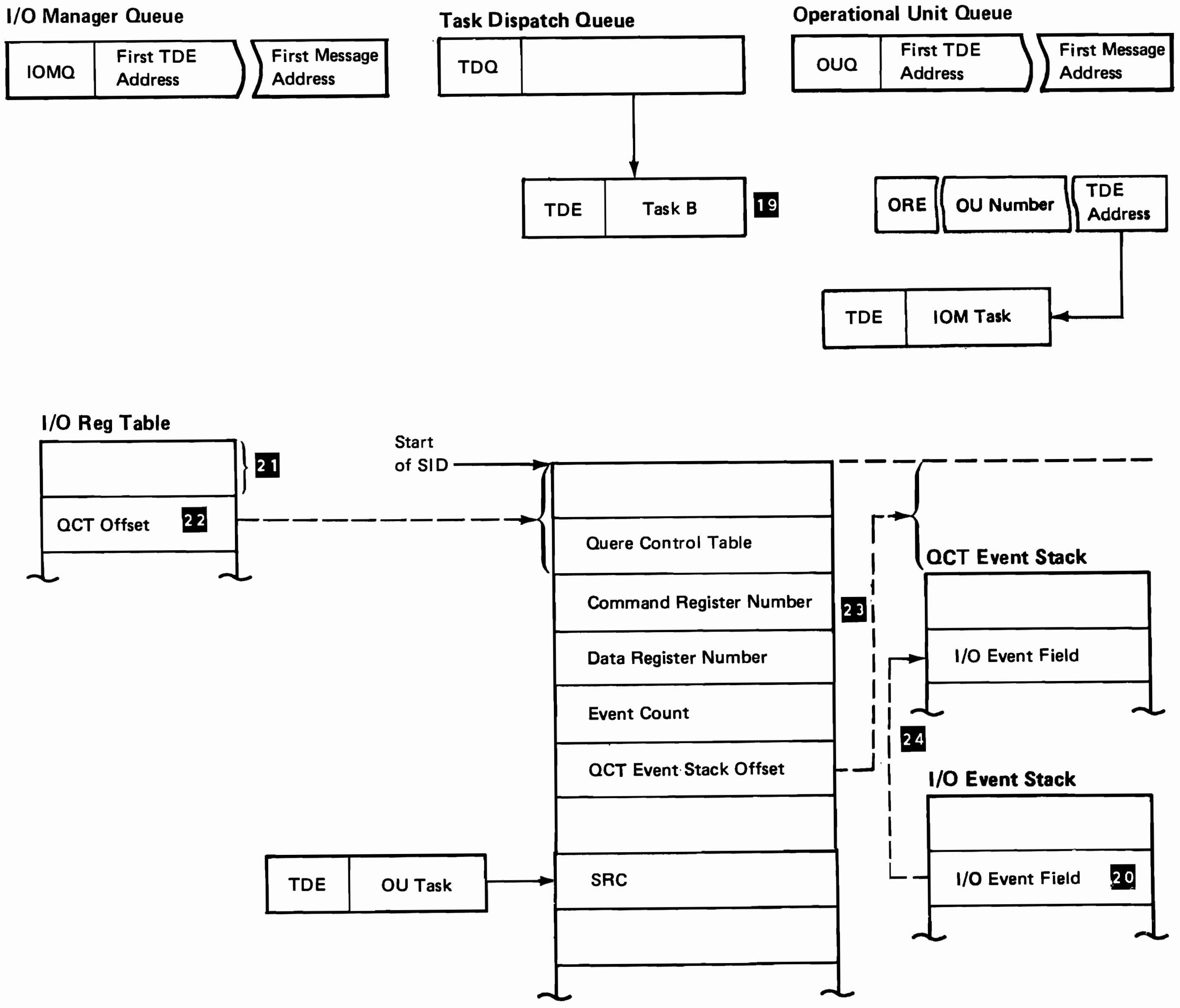


Figure 7-6 (Part 9 of 13). Sequence of I/O Operations with SENDMW Instruction

25 A Send Count instruction is issued by the event handler to the QCT SRC header. The Send Count instruction increments to the SRC count.

26 The TDE of the OU Task is placed on the TDQ in priority sequence.

The event handler repeats the sequence (in this example, numbered 20-26) until all entries are removed from the I/O event stack.

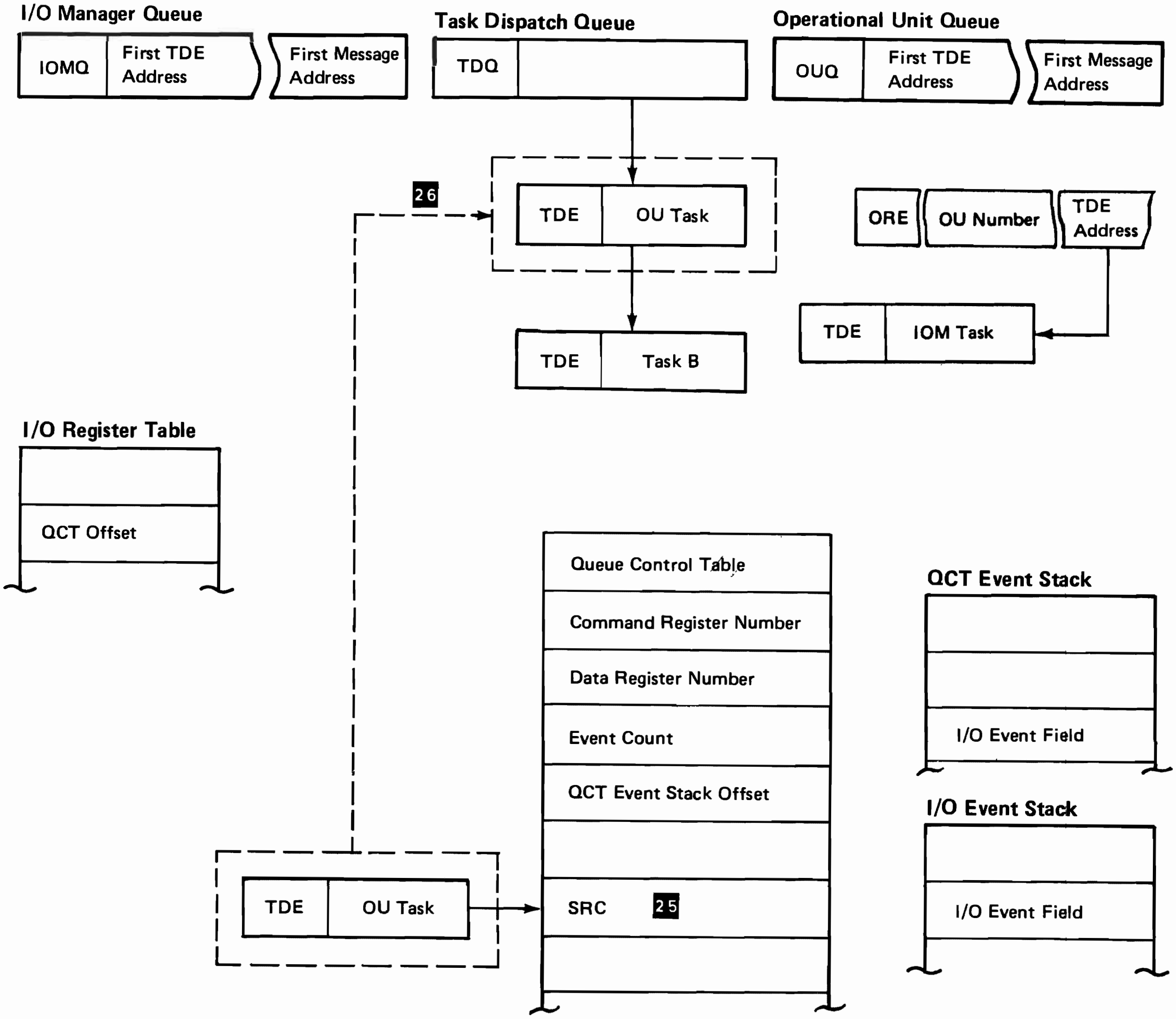


Figure 7-6 (Part 10 of 13). Sequence of I/O operations with SENDMW Instruction

27 Control passes to the TDE with the highest priority. In this example, the OU task begins execution.

28 The OU task issues a Receive Count instruction to the QCT SRC. This implicitly tests for waiting events in the QCT event stack.

29 The OU task locates the I/O event field and checks the channel command byte of the event field for the function event type.

30 When the function event type is a command completion indication, that is, if command end or command end/fetch next command is the function type, then the BSTAT field of the I/O event is moved to the OU status field of the ORE. (See Figure 7-7.)

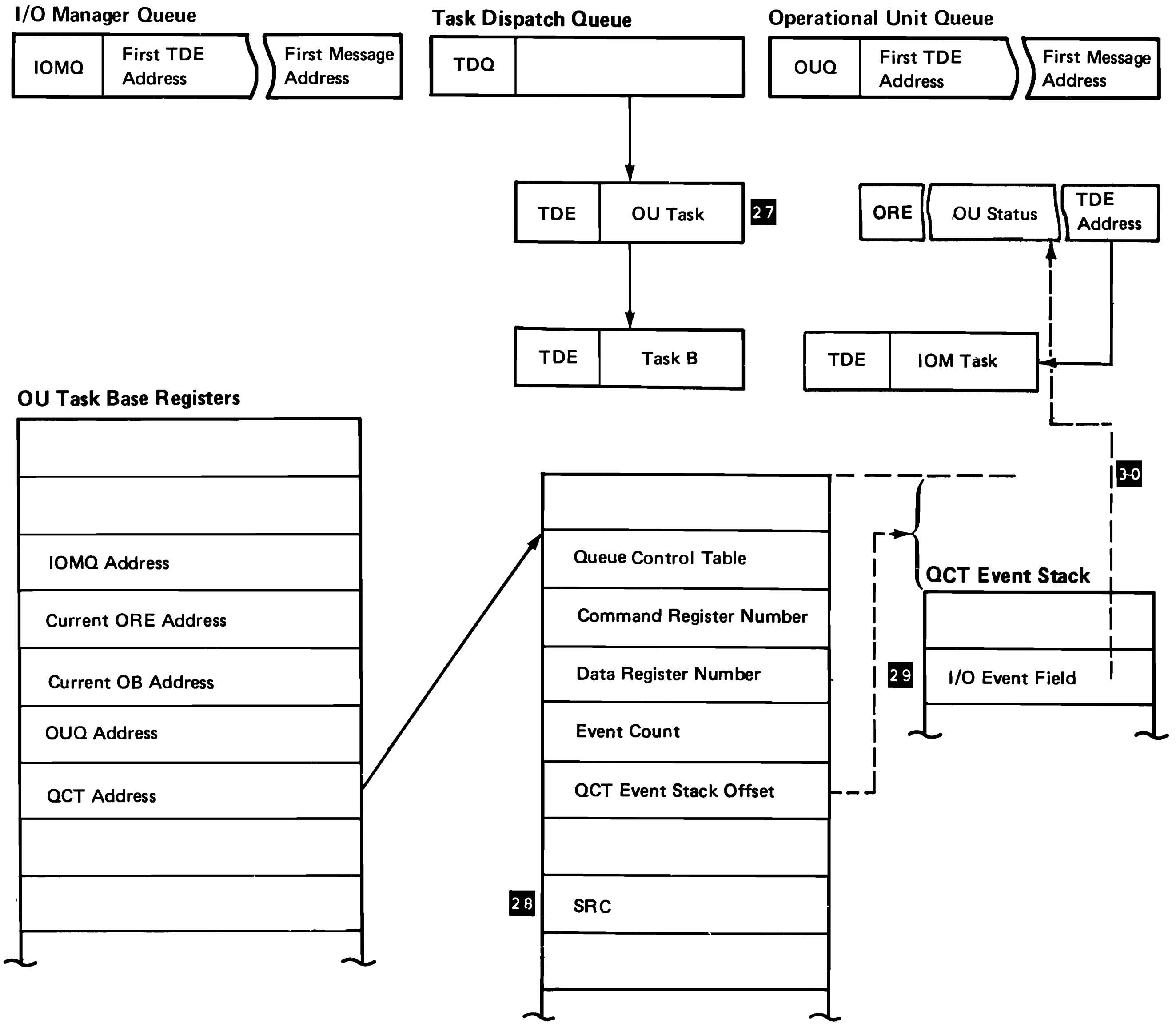


Figure 7-6 (Part 11 of 13). Sequence of I/O Operations with SENDMW Instruction

31 If command end is the command completion indication, the OU task issues a Send Message instruction to the IOMQ. Since bit 6 of the description byte of the SRM is on, the message is not enqueued on the IOMQ. Instead, the IOM task TDE is enqueued to the TDQ.

32 The OU task then issues a Receive Count instruction to the QCT SRC. The OU task now resides on the QCT SRC wait list, waiting for an IOM work request to be placed on the ACT event stack.

33 If (instead of 31 and 32) comand end/fetch next command is the command completion indication, the OU task checks the current OB (operation block) for the last OB in the ORE. When the current OB is not the last OB, the task processes the next OB. If the current OB is the last OB, the task issues a Send Message instruction to IOMQ as in 31 above.

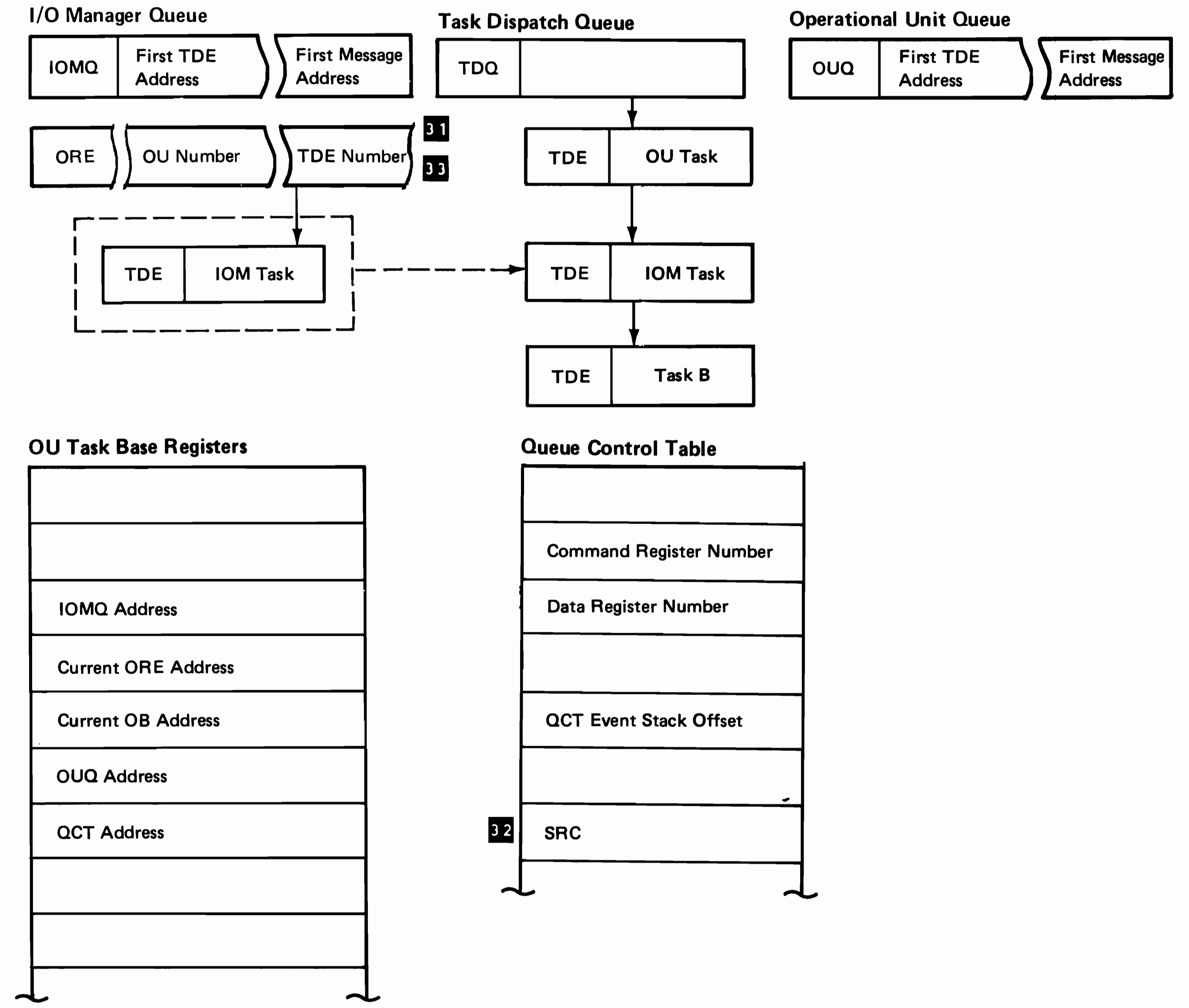


Figure 7-6 (Part 12 of 13). Sequence of I/O Operations with SENDMW Instruction

34 The OU task now issues a Receive Message instruction to the OUQ. With no messages (OREs) queued to the OUQ message list, the OU task is dequeued from the TDQ and placed on the OUQ wait list.

35 The IOM task resumes execution (task switch) if it is of higher priority than task B.

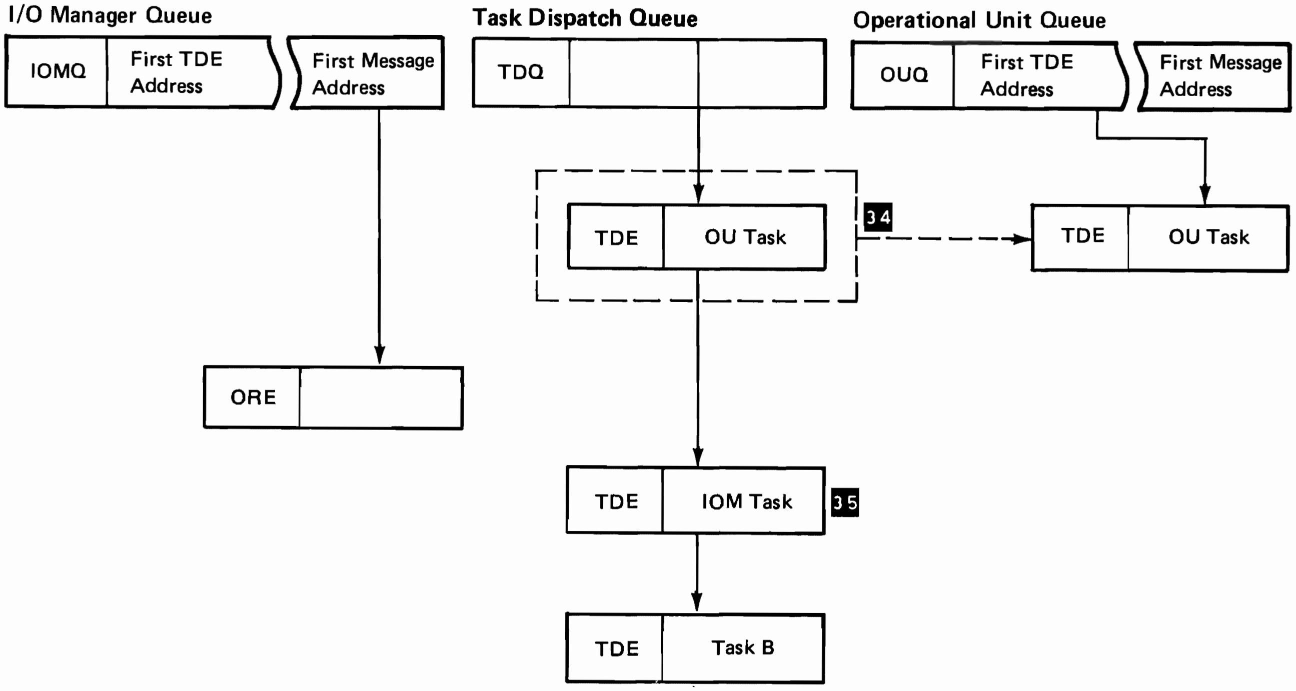


Figure 7-6 (Part 13 of 13). Sequence of I/O Operations with SENDMW Instruction

I/O Errors

Extensive error checking is provided within the I/O structure to ensure correct operation of each component and to maintain the integrity of device data. I/O errors are OU (operation unit) errors when reported to the OU IOM (I/O manager) or channel errors when reported to the channel IOM.

OPERATIONAL UNIT ERRORS

Operational unit errors, that is, operation program errors and device errors, report to the operational unit IOM via the 2-byte operational unit status field of the current ORE (operation request element). See Figure 7-7.

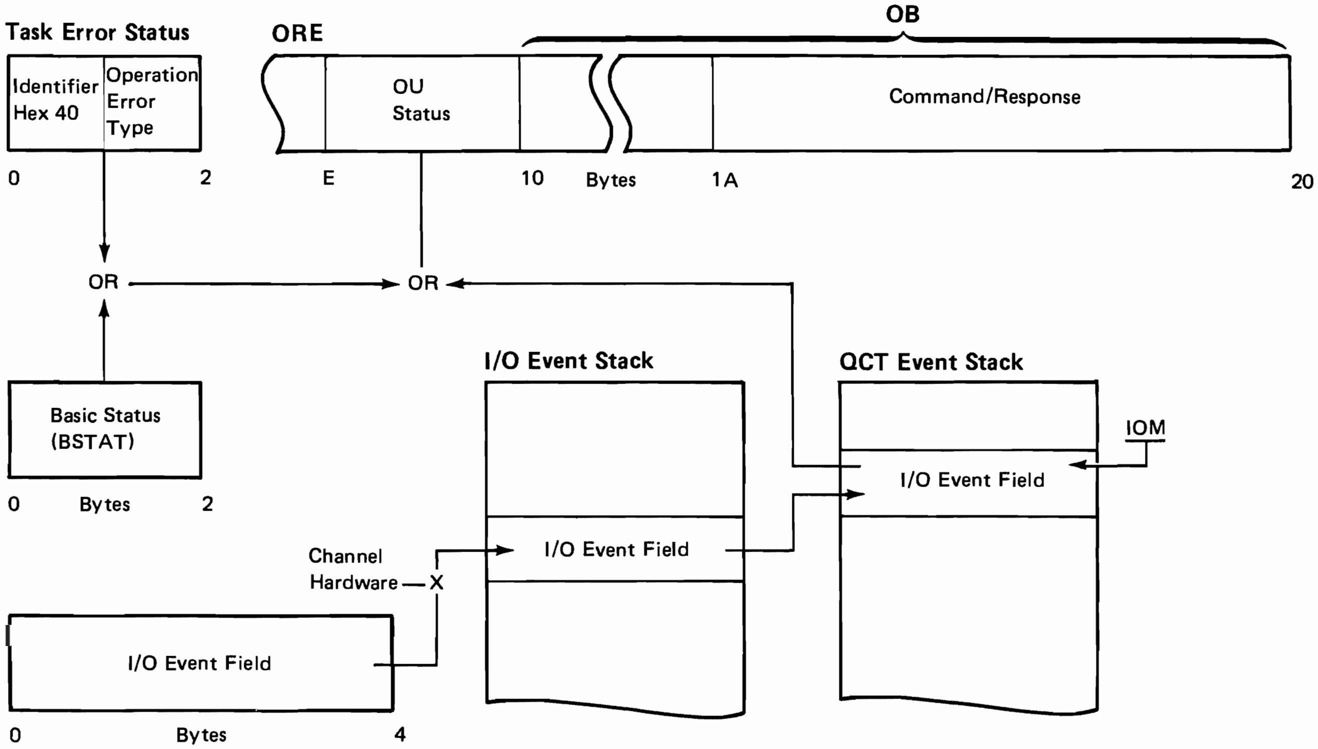


Figure 7-7. Status Fields

Operation Program Errors

Operation program errors occur during the execution of OBs (operation blocks) by the OU (operational unit) task. The type of operation program error is indicated in a 2-byte field called the task error status field.

Task Error Status Field

The operation program error type contains an error code indicating the type of operation program error detected by the OU task during processing of an operation block. Refer to the *Channel Theory-Maintenance* manual.

Device Errors

Device errors occur during the execution of an I/O command contained in a function operation block. A device error causes the device to supply a command end completion indication to its OU task and to provide status associated with the error condition in a 2-byte basic status field. (See *Basic Status* earlier in this chapter.)

Additional status information is available to the IOM in the form of device status fields. (See *Device Status* earlier in this chapter.)

Operational Unit Error Recovery

Operational unit recovery procedures must be initiated by an IOM (I/O manager) whenever the IOM is notified of either a device error, an operation program error, or a channel-detected error. The associated operational unit task resides in the receive-wait state on the QCT-SRC (queue control table, send/receive counter) queue and must be cleared for further operational unit activity.

The IOM must proceed as follows:

1. Lock the QCT.

2. Test FOB (function operation block) in-use bit.
   a. If set, form a command-end function event, place the function event on the QCT event stack, and issue a send count to the QCT-SRC. This sequence, in effect, redispatches the OU task and supplies it with an IOM-formed work request.
   b. If not set, test for OU (operational unit) task in receive-wait state on the QCT-SRC queue.

   If the OU task is on the QCT-SRC queue, then form a fetch next command function event. Place the function event on QCT event stack and issue a send count to the QCT-SRC. If the OU task is not on the QCT-SRC, then continue.

3. Unlock the QCT.

Operation Program Error Recording

Defined operation program errors common to all IOMs are shown in Figure 7-8. The table defines the error code 1, error name 2, error class 3, record type 4, retry limit 6, and the priority 7 in which the error should be decoded from the status information. The method of handling temporary retryable errors 5 is also defined. They may be counted in a storage data register counter and not logged (x in count-only column). All errors that cause an entry into retry may be logged (x in log all column) or they may be thresholded (threshold value in threshold limit column) and counted in a storage data register counter. An x in the count-retryable column indicates that the total number of retries should be counted in a storage data register counter for this error.

The error recovery action required and the error log format for the error are also referenced.

Error Definition

Error Code 1	Error Name 2	Error Class 3	Record Type 4			Temporary Retry Errors 5				Retry Limit 6	Prior-ity 7
			Perm-anent Record	Temp-orary Record	Temp-orary Retry	Count Only	Log All	Thresh-hold Limit	Count Retry		
9998	Operation error—Channel busy				X			2		5	1
9999	Operation program error				X		X			1	2

Figure 7-8. Operation Program Errors

The description of each error in terms of OU-status bytes and bits is shown in Figure 7-9.

Error Definition

Error Code	Error Description	OU Status		FOB Bytes		Recovery Action		Error Log Format
		Byte 0	Byte 1	C D	E F	Oper-ation	Pro-gram	
1	Operation error—Channel busy	40	07				1	
1	Operation program error	40	XX				2	

1 2 3 4 5 6 7 8

Legend: Byte = 40 means high-order hex digit = 4
low-order hex digit = 0
Byte = xx means may be any value
Recovery = Error recovery procedures

Figure 7-9. Error Descriptions

Error Recovery Procedures

Repeat the command up to 5 times. If the condition persists after 5 unsuccessful retries, log the error. IOM then must initiate error recovery (described in *Operational Unit Error Recovery* earlier in this chapter).

The error log format for the OU-IOM is the same for all operation program errors logged. See the *Channel Theory-Maintenance* manual for the format description.

CHANNEL ERRORS

Channel errors are an error class encountered during the operation of a device on the channel interface. A channel error causes the device to cease all operation, and an error event field (refer to *I/O Event Stack*, earlier in this chapter) containing all available status information is passed to the channel IOM (I/O manager) task. Channel errors fall into three subclasses: channel hardware error, I/O event handler error, and a special class of device error report using the post event function. The post event function is used by a device to communicate error status to its IOM via the channel IOM.

The event field format for each subclass is as follows with each type of channel error described in detail in the *Channel Theory-Maintenance* manual.

Channel Hardware Error

d1dddddd	I/O Register Number	Channel Priority Code	ppppss00
0 Bytes	1	2	3

Legend: d = Device dependent
p = Primary channel error code
s = Secondary channel error code

Event Handler Error

0001
or
1111

a1aaaaaa	aaaaaaaa	aaaaaaaa	bbbb
0 Bytes	1	2	3

Legend: a = Depends on the event type processed, see the *Channel Theory-Maintenance* manual for specifics.
b = Event handler error code

Post Event

Post Event	I/O Register	Operational Unit Reporting	ddddddyz
0 Bytes	1	2	3

Legend: d = Device-dependent information provided by the device
y (bit 6) = Post event type
0 Error condition detected in adapter and/or device which cannot be reported via a normal BSTAT.
1 Attention request to OU-IOM.
z (bit 7) = OU type
0 Single OU reporting, that is, OU number in byte 2 of post event defines a unique reporting OU.
1 Multiple OUs reporting, that is, OU number in byte 2 of post event is reporting for all OUs attached to a particular I/O port.

Channel Error Reporting

The active device at the time of the error has its operation suspended until the error is resolved. The error must be posted to the channel IOM and notification given to the operational unit IOM that a channel error has occurred and the device halted. The I/O event handler moves the error event field data from the I/O event stack (see Figure 7-7) to the QCT (queue control table) event stack of the channel OU (operational unit). A Send Count instruction to the SRC (send/receive counter) of the channel OU-QCT then signals the channel OU task. The channel OU task sends the ORE (operation request element) containing the error event to the IOMQ for the channel IOM.

The channel IOM notifies the operational unit IOM that a channel error has occurred and the device is halted. The operational units IOM now proceeds as described under *Operational Unit Error Recovery* earlier in this chapter.

Channel Error Recovery

Four FOB (function operation block) commands allow the channel IOM access to the channel hardware to perform I/O error recovery operations.

The channel IOM communicates with the channel hardware via an OU and IOM queue pair. An OU (operational unit) of hex 00 is used for FOB commands directed to the first hardware channel on the system.

The channel FOB commands are decoded and executed by the OU task servicing the OU queue used by the channel error IOM.

- Start Channel (hex 11)

  The start channel FOB command can be used at IMPL or following a secondary channel error condition that caused the channel hardware to stop. The start channel command causes the channel OU task to reset the secondary error stop bit in register E0 (hardware register), which causes the channel hardware to log the 4-byte event field in the hardware into the I/O event stack. The secondary error stop bit is then set to force the channel hardware to stop on the next secondary error condition.

  The channel OU task next issues a Receive Count instruction to the SRC in the channel OU task control table. When the I/O event handler services the I/O event stack and issues a Send Count instruction to the QCT of the channel OU task, the OU task removes the event from the QCT event stack and places the 4-byte field into FOB bytes hex C-F. Following the read event operation, the command-complete bit in the OU status field of the ORE is set and the FOB is completed.

  Note: The event field obtained during the start channel FOB is normally the event field maintained in the channel hardware when a secondary error is detected. However, because the event fields are removed from the event stack on a last in, first out manner, any event fields posted by I/O devices between the time the channel hardware starts and the channel OU task is dispatched can be returned prior to the secondary error event field.

- Start Device (hex 12)

  The start device FOB command causes a start device channel sequence. The start device channel sequence is normally used by an OU task, during the processing of an FOB block, to notify an I/O device that a command is available for execution. Depending on the device implementation of the channel sequence, the command information in the FOB may be transferred to the device during the start device sequence.

  The device address to be used during the channel sequence is provided in byte hex C of the FOB. The command complete bit in the OU status field of the ORE is set upon successful completion of the FOB.

- Halt Device (hex 21)

  The halt device command is similar to the start device command in that an I/O device attached to the channel is selected by a broadcast of the device address provided in byte hex C of the FOB. The halt device command issues a halt condition to the selected device to cause termination of any active command.

  Completion of the halt device command is indicated by the command-complete bit in the OU status field of the ORE being set.

- Read Event (hex 22)

  The read event command is used by the channel error IOM to obtain events from the channel IOM QCT event stack. The read event causes the OU task servicing the channel IOM to issue a Receive Count instruction to the SRC in the queue control table. If no events are on the QCT event stack, a task switch occurs with the OU task waiting for a Send Count instruction from the I/O event handler.

  If event fields are in the QCT event stack, or when the OU task is dispatched by a Send Count instruction to the SRC, an event field is removed from the QCT event stack in a first in, first out manner and placed in FOB bytes hex C-F.

  The read event command is completed by setting the command complete bit in the OU status field of the ORE.

Channel Error Recording

Error Definition

Errors defined for the channel are the error code, error name, error class, record type, retry limit, and the priority in which the error should be decoded from the status information. The method of handling temporary retryable errors is also defined. They may be counted in a storage data register counter and not logged. All errors that cause an entry into retry may be logged or they may be thresholded and counted in an SDR counter. See the *Channel Theory-Maintenance* manual for specific error codes.

Error Recovery Procedures

1. Repeat the command up to five times. If the condition persists after five unsuccessful retries, perform a Terminate Immediately instruction. Operator panel light-emitting diode readout = hex 0832.

2. Perform a Terminate Immediately instruction.

3. Issue an AOB (address operation block) to read IORAR 0 (I/O resolved address register 0). If offset of IORAR is 512 bytes from the beginning address, or if retry fails, perform a Terminate Immediately Instruction. Operator panel light-emitting diode readout = hex 0831. Otherwise, issue a start channel FOB (function operation block).

4. Log the error. There is no recovery since the I/O adapter is not uniquely known.

5. Send a message containing the post event field (bytes hex C-F of read event field FOB) to IOM (I/O manager) servicing the OU (operational unit) whose number is contained in byte hex E of the read event FOB. In the event that byte hex E contains an OU number that is currently inactive or invalid, then log the error.

6. Log the error and send message containing error event field (bytes hex C-F of read event FOB) to the IOM servicing the OU having the IORAR contained in byte hex E of the read event FOB.

7. Log the error and send message containing error event field (bytes hex C-F of read event FOB) to the IOM servicing the OU having the IORAR contained in byte hex D of the read event FOB.

8. Log the error and send message containing error event field (bytes hex C-F of read event FOB) to the IOM servicing the OU whose IORAR and channel priority code match respectively bytes hex D and E of the read event FOB.

9. Log the error and send message containing error event field (bytes hex C-F of read event FOB) to the IOM servicing the OU having the IORAR contained in byte hex E of the read event FOB.

Error Log Format:

The error log format for the channel has the same format for all errors logged. See the *Channel Theory-Maintenance* manual for the formats and error descriptions.

DEVICE HALT

During the execution of an ORE (operation request element) by an OU (operational unit), the device IOM (I/O manager) task can terminate execution of the ORE. This is accomplished by the device IOM task requesting the channel IOM task to perform a halt device function. The following occurs:

- The channel IOM sends a message (ORE) to the channel OUQ (operational unit queue). The ORE contains an FOB with a command byte of halt device (hex 21) and with the OU number of the device to be halted in byte hex C.

- The channel OU task services the channel OUQ and passes the command to the channel hardware, which issues a halt to the selected device.

- The channel OU task accepts the device's response to the channel as a device command completion indication and sends an ORE containing completion indication to the IOMQ of the channel IOM.

The device OU task has not received indication of the termination of the ORE. The OU task resides on the wait list of the QCT-SRC of the operational unit in the recieve wait state. To clear this state requires action by both the channel IOM and device IOM tasks:

- The channel IOM forms an I/O event field (see Figure 7-7) of the function event type (command end) as follows:

Hex 02	Hex 00	Hex 20	Hex 00

0 Bytes 1 2 3

- The I/O event field is placed on the operational unit QCT event stack, and a Send Count instruction is issued to the QCT-SRC of the operational unit.

- The device IOM forms an I/O event field of the function event type (fetch next command) and places it on the OU-QCT event stack. A Send Count instruction is issued to the QCT-SRC of the OU.

- The OU task is now placed on the TDQ in priority sequence.

Chapter 8. Virtual Storage Addressing

The following addresses are virtual:

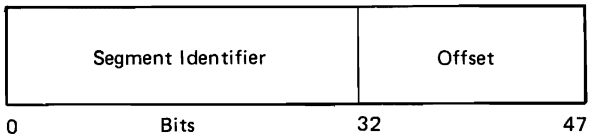
- All addresses used by the processor in executing instructions or fetching data operands.

- All storage addresses that are explicitly specified by an IMP (internal microprogramming) instruction and are used by the processor.

- The address(es) indicated to the processor on an exception or as the result of executing an instruction.

- All storage addresses explicitly specified in I/O messages.

The complete virtual address of any byte of storage is a 48-bit address as shown below.



The 48-bit virtual address is translated by the processor into a real storage address using the VAT (virtual address translator) facility described in the following paragraphs.

Virtual Address Translator Overview

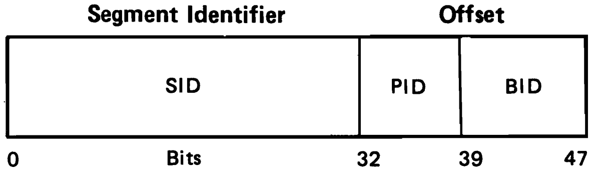
The VAT facilities:

- Translate virtual storage addresses to real storage addresses; or, when that translation cannot be completed,
- Interrupt the execution of IMP instructions, which allows:
  - Invocation of storage management functions, which
  - Alters the contents of real storage, which allows
  - Continuation of processing

TRANSLATION PROCESS

During translation, two units of information are recognized–segments and pages. A *segment* is a block of sequential virtual addresses spanning up to 65 536 (2^{16}) bytes. A *page* is a block of sequential virtual addresses and contiguous storage locations containing 512 bytes beginning at a virtual address that is a multiple of the page size. All pages in storage are the same size.

The 48-bit virtual address logically is divided into two parts. Bits 0-31 are used as an SID (segment identifier). Bits 32-47 are used to provide an offset to data within the segment. For translation to main storage addresses, bits 32-38 are used as a PID (page identifier). The remaining bits of offset are used as a BID (byte identifier) within a page. See the following diagram.



Translation is achieved by means of translation tables. Each table entry describes a block of consecutive real storage locations. Each such block is called a *page frame*. Each page frame contains a *page* of instructions or data.

The method used by the virtual address translator to translate a virtual address to a real storage address depends on the value of the virtual address. Virtual addresses, when they are within the SID (segment identifier) range shown below, are converted to real storage addresses by selecting the appropriate bits.

System Unit	Models	V=R Address SID Range (Hex)	Selected Bits
5381	3, 4, 5	0000 0100 - 0000 011F	27-47
	6, 7, 8	0000 0100 - 0000 01FF	24-47
5382	All	0000 0100 - 0000 02FF	22-47

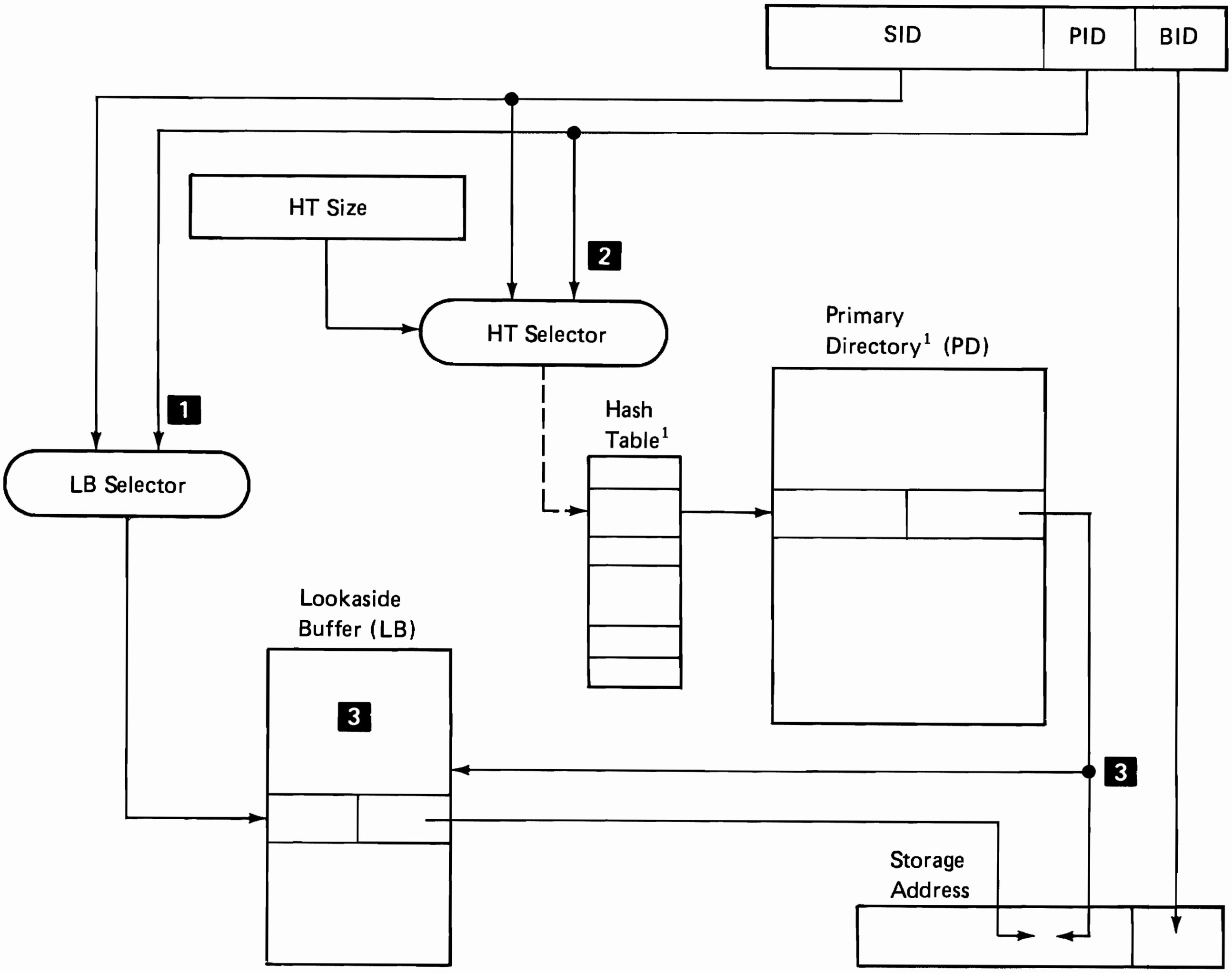
This is referred to as virtual = real addressing. Those virtual addresses not in the virtual = real addressing range are referred to as a virtual = virtual addresses, and are translated to real addresses by means of the PD (primary directory). If the resultant real address is too large for a particular available main storage size, an addressing exception results.

The assignment of storage occurs in page-size blocks; the storage locations are assigned contiguously within a page. Two pages need not be adjacent in storage (unless they are virtual = real) even though assigned a set of sequential virtual addresses.

The SID and PID portion of a 48-bit virtual address to be translated by means of the PD are used to select an entry from the PD. The PD entry, whose format is described later in this chapter, contains information that specifies one of the following actions:

- If the PD entry describes a page frame of storage that contains the page whose SID and PID match that of the address to be translated, the storage address is formed from this PD entry.
- If the PD entry does not describe such a page frame of storage, advance to and examine another PD entry.
- If there are no more PD entries to examine, signal an address translation exception.

Virtual-to-Real Address Translation



[1] In Storage

1 Information extracted from the virtual address is used to search the LB.

2 Information extracted from the virtual address is used to search the PD.

3 If no match exists in the LB, the PD in storage is searched to translate the address. If a match exists, the information is used to form an entry in the LB.

Programming Note: The primary directory and hash table for storage are in a virtual = real segment in storage, and cān be accessed by the IMP instructions.

Virtual Address Translator Components

Address translation is performed by means of the HT (hash table), the PD (primary directory), and the high-speed LB (lookaside buffer).

The HT and the PD reside in storage and can be accessed by the IMP instructions. Their structure and functional characteristics are described in this section. Also discussed are the functional characteristics of the LB, which does not reside in storage and whose contents cannot be accessed as data by the IMP instructions.

CONTROL INFORMATION

The address of the first HT entry and the address of the first entry in the PD directory are in the control address table. The sizes (the number of entries-1) of the HT and the PD are also contained in the control address table. These fields are used by the processor during IMPL. They can be accessed or modified at any time by IMP instructions, but any changes do not affect the address translation process.

HASH TABLE

An entry fetched from the HT provides an index into the PD. The number of hash table entries varies, as the following chart shows:

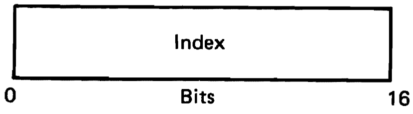
System Unit	Model	Number of Entries (In Powers-of-2 Increments)
5381	3, 4, 5	256 - 32 768
	6, 7, 8	256 - 65 536
5382	All	256 - 65 536

The number of entries is controlled by the HT size field. Generally the hash table should contain at least two entries for every PD entry in order to control the length of PD entry chains. The number of HT entries must be a power of 2. Each entry has 16 bits of data.

The HT entries occupy contiguous storage beginning at the address specified by the HT address field. The hash table must be aligned on a segment boundary. For Models 3, 4, and 5, the table must be within one virtual virtual = real segment. For the the 5381 System Unit, Models 6, 7, and 8, and for all 5382 System Unit models, the hash table may cross a segment boundary but must be within two virtual = real segments.

An entry value of zeros indicates an end-of-chain condition.

Hash Table Entry Format



Hash Table Lookup

The processor accesses the HT in storage as part of the address translation process. The SID and PID portion of the virtual address are used to select an entry from the HT. The value of the selected HT entry is used to select a PD entry.

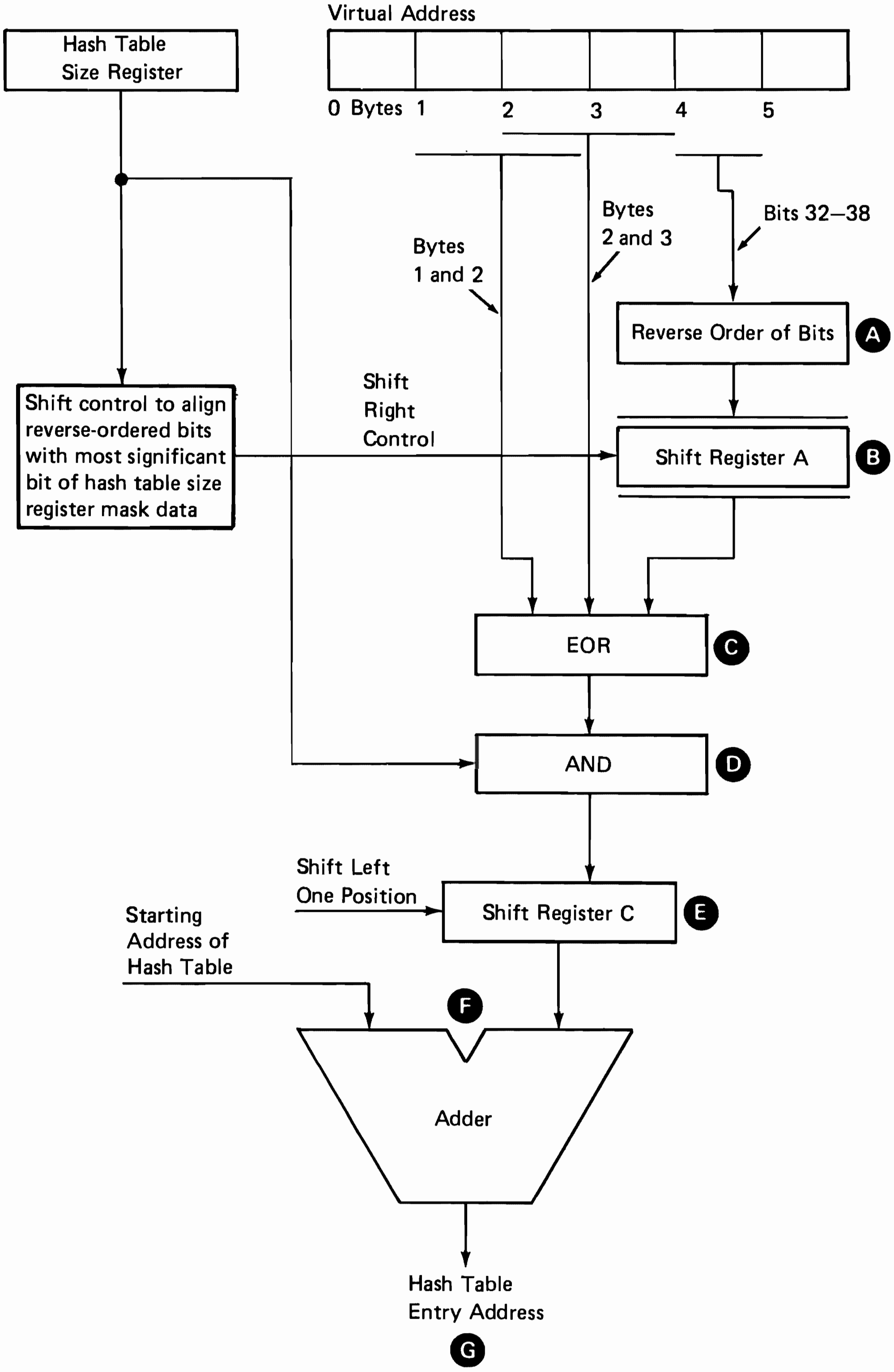
A 16-bit HT entry index value is generated from the virtual address by address compression (hashing) of the 39-bit field formed by linking the SID to the PID. The number of significant bits in the result is controlled by the hash table size field. The hash table entry index value is used to select a hash table entry. Hashing is shown in the following diagram and is described in the following text:

The PID bits (32-38 of the virtual address) Ⓐ are reversed (38-32) Ⓑ then shifted right once for each zero bit in the HT size register, plus one.

Bytes 1 and 2 of the SID are Ⓒ exclusively ORed with bytes 2 and 3 of the SID and then exclusively ORed with the result of the reversed and shifted PID bits. The shifted data is then Ⓓ ANDed with the halfword HT size register (from the control address table). This ANDing causes the hash on the left to be truncated so that the result has the same size specified by the HT size register. The result Ⓔ is shifted left 1 bit position.

The result is Ⓕ added to the beginning address of the hash table. The sum Ⓖ is used as an entry into the hash table.

The virtual address of the selected HT entry is obtained by adding bits 0-15 of the index to bit 31-46 of the virtual address of the HT (as given in the HT address field). Bit 0 of the HT index is ignored (on Models 3, 4, and 5) and treated as 0, and low-order bit 47 is forced to 0.


Virtual Address
Hash Table Size Register
0 Bytes 1
2
3
4
5
Bytes 1 and 2
Bytes 2 and 3
Bits 32–38
Reverse Order of Bits
A
Shift Right Control
Shift control to align reverse-ordered bits with most significant bit of hash table size register mask data
Shift Register A
B
EOR
C
AND
D
Shift Left One Position
Shift Register C
E
Starting Address of Hash Table
F
Adder
Hash Table Entry Address
G

As part of the hash table lookup process, the index is tested for a value of all-zeros (end-of-chain). If nonzero, the index field is used to access an entry in the primary directory. The virtual address of the PD entry is obtained by adding the 16-bit hash table entry to bits 28-43 of the virtual address of the primary directory. The low-order 4 bits (44-47) are forced to zeros. The control address table contains the address of the primary directory. Bits 0-3 of the PD index are ignored and treated as zeros (Models 3, 4, and 5 only).

A VMC program can use the HVVA (Hash and Verify Virtual Address) instruction and the LHTEA (Load Hash Table Entry Address) instruction to access a HT entry in the same way that the processor accesses HT entries. See Chapter 10 for a discussion of this instruction.

PRIMARY DIRECTORY

One PD (primary directory) entry is provided for each frame of main storage installed on the system. A PD entry fetched from the PD (primary directory) indicates the virtual address of the page stored in the block of storage represented by the PD entry and the status of the page. Linkage to other PD entries is also provided. Each PD entry contains 16 bytes of data. There can be 1 to 65 536 entries, in power-of-2 increments.

The PD entries occupy contiguous storage beginning at the address specified by the PD address field. For Models 3, 4, and 5 the primary directory must be SID-aligned and may not cross a segment boundary. For Models 6, 7, and 8 the primary directory may cross segment boundaries but must not exceed 16 virtual = real segments.

Primary Directory Entry Format

SID	PID	Status	Index	PINCNT	Status	Not Used	Usage Code		Not Used
0 Bytes	4	5	6	8	9	A	B	C	10

The fields are allocated as follows:

Bytes (Hex)	Bits	Description
0-3		**SID:** Segment identifier of the page stored in this block of storage. The SID field is compared against bits 0-31 of the virtual address to be translated.
4		**PID:** Page identifier of the page stored in this block of storage. The PID field is compared against bits 32-38 of the virtual address to be translated. Bit 39 must be zero.
5		**Status:** Information about the page.
	0	Valid: This bit can be set by the VMC and should be reset by only the Invalidate Primary Directory Entry and Examine Primary Directory Entry instructions. 0 Page described by this PD entry is *not* available for access. An address translation exception is recognized and the operation being attempted is nullified. 1 Page described by this PD entry is available for access. Address translation proceeds, using the PD entry.
	1	Reference: This bit is set whenever the corresponding non V=R page frame is accessed by the processor and the corresponding entry is not in the lookaside buffer, or when an I/O operation requires use of the address. This bit should only be reset by the Read Reference and Change and Reset Reference instruction, and in some cases, by an Examine Primary Directory Entry instruction. 0 Page has *not* been referred to. 1 Page has been referred to.
	2	Change: This bit is updated (ORed with the change bit in the lookaside buffer) when the corresponding lookaside buffer entry is removed by the processor updating the lookaside buffer; by an Invalidate Primary Directory Entry instruction, a Read Reference and Change and Reset Reference instruction, or in some cases, an Examine Primary Directory Entry instruction; or when an I/O operation is started which will store into the associated non V=R page frame. This bit is reset by the VMC. **Note:** The lookaside buffer change bit is set whenever the processor stores data in to the associated non V=R page frame. 0 Page has *not* been changed. 1 Page has been changed.
	3-4	I/O Used by the processor when the page is being used by the I/O. The bits are both set and reset by the processor.
	5-7	Reserved: Must be zeros.
6-7	0-15	**Index:** Index for the next PD entry in this chain of PD entries. The value of the index field is used by the processor to access the next PD entry in a chain of entries. An all-zero value indicates an end-of-chain condition.
8		**PINCNT:** A 1-byte use counter for pinning (holding) pages in storage. A nonzero value indicates that the page is in use and should not be removed from storage. The counter can be updated by either the IMP task or the processor.

Bytes (Hex)	Bits	Description
9		**Status:** Provides page status information, and is defined and maintained by the vertical microcode.
	0	Not used.
	1	Purge indicator set/reset by EPDE.
	2	Not used.
	3	Write pending.
	4	Access pending.
	5	Faulty page.
	6	Nucleus.
	7	Virtual=Real page.
A		**Not used.**
B		**Usage Code Byte:** Maintained by the vertical microcode.
	0-1	Not used.
	2	Usage code.
	3-7	Not used.
C-F		**Not used.**

Programming Note: The processor sets the reference and change bits. The Read Reference and Change and Reset Reference instruction resets the reference bit. The program must reset the change bit.

Primary Directory Lookup

The processor accesses the PD in storage as part of the address translation process.

The SID and PID portion of the virtual address are used to select a PD entry as previously described in this chapter under *Translation Process*. The PD entry fetched from the primary directory indicates the virtual address and status of the page described. The SID of the virtual address to be translated is compared to the SID of the page stored in the page frame of storage described by the fetched PD entry. Bits 0-31 of the PD entry are compared to the SID of the virtual address to be translated.

Bits 32-38 of the PD entry are compared to the PID of the virtual address to be translated. If a match occurs and bit 40 is 1, this page is available for access. The storage page address may be formed as described in *Storage Address Formation* in this chapter. If a match occurs and bit 40 is 0, an address translation exception occurs, and the operation being attempted is nullified.

If no match occurs, the index field is tested for 0. If this field is 0, an address translation exception occurs; if the field is not 0, the index field is used to access another PD entry. The virtual address of the PD entry is obtained by multiplying the index field by 16 and adding the resultant 20-bit offset to the address of the PD. On Models 3, 4, and 5, the high-order 4 bits of this 20-bit offset must be zeros. The PD address must be SID-aligned and is obtained from the PD address field of the control address table.

When a virtual address is used in an I/O operation, the PINCNT field in the PD entry for that address is incremented by the processor. When the I/O operation is completed, the processor decrements the PINCNT field. This ensures that an IMP task does not invalidate the PD entry in the middle of an I/O operation.

When the next PD entry has been fetched, all tests and steps described for the first PD entry of the chain are performed. The lookup operation continues from entry to entry until encountering either a PD entry with a matching virtual address or a nonmatching virtual address and zero index value, indicating an end of chain.

Storage Address Formation

When a PD entry is found that contains a virtual address matching the address to be translated and the page is available for access (bit 40 = 1), the storage address is formed. The processor uses bits 0-15 of the index value pointing at the current PD entry as the page frame identifier and 9 bits of the BID (byte identifier) from the virtual address to be translated in forming the storage address. The BID is concatenated to the right of the page frame identifier to provide 25 bits of storage address.

LOOKASIDE BUFFER

To enhance performance, the VAT facility is implemented so that some of the information specified in the primary directory in storage is also maintained by the processor in a special buffer called the LB (lookaside buffer). The processor refers to a PD entry in main storage for the initial access to the entry, then maintains the information in the lookaside buffer. All subsequent translations involving PD entries from the same real storage page frame can use the information recorded in the lookaside buffer.

The presence of the lookaside buffer affects the translation process in that a modification of the contents of a PD entry in storage does not have an immediate effect on the translation. Also, changes to the reference and change status of a page are not immediately visible in the PD entry in storage if address translation is accomplished using the lookaside buffer.

The LB entries are not explicitly addressed by IMP instructions, nor can information be explicitly entered into the lookaside buffer by executing IMP instructions. How the reference and change bits can be read out and how entries can be removed by executing IMP instructions is described in the next topic. Entries are implicitly added to and removed from the lookaside buffer by the translation process explained in *Translation Process* earlier in this chapter. A copy of a PD entry is placed in the lookaside buffer only when the valid status bit of the PD entry is 1. An address translation exception is recognized when an attempt is made to use an invalid PD entry for translation.

When a copy of a PD entry exists in the lookaside buffer, the PD entry is said to be active. The LB entry copy of a PD entry can be implicitly removed from the lookaside buffer by the processor to fulfill subsequent translations involving other PD entries. Once the LB entry copy is removed from the lookaside buffer, the PD entry is said to be inactive. No status bit is provided to show the active-inactive state of a PD entry. When an active PD entry becomes inactive, the reference and change status bit of the page is updated in the PD entry to reflect the most recent active status of the page.

Reference and change recording takes place for any storage access made by the processor and I/O operations unless the I/O operation uses virtual = real addresses. Hence, references to a storage location associated with I/O operations are included.

The change bit is not turned on for an attempt to store if the storage reference is not permitted, regardless of whether the IMP instruction responsible for the reference is suppressed or terminated. In particular, a processor reference causing an addressing or address translation exception does not cause the change bit to be turned on.

VAT Component Maintenance

The VAT (virtual address translator) components in storage–the control information, the HT (hash table), and the PD (primary directory)–can be accessed and modified by IMP instructions. Control information modification is discussed in *Control Information* under *Virtual Address Translator Components* in this chapter. This paragraph describes the effects of any manipulation of an HT or PD entry by IMP instructions and the relationship of changes in the primary directory to changes in the LB (lookaside buffer).

MODIFICATION OF HASH TABLE ENTRIES

The effects of changes to an HT entry depend on the status of any associated PD entries; that is, a PD entry indexed by the HT entry or any PD entries connected to that PD entry by chaining. See the next topic for a description of the status of PD entries.

A change to an HT entry associated with inactive PD entries takes place immediately. A change to an HT entry associated with active PD entries can take effect for implicit translation any time after the instant of change (when the entry becomes inactive).

Programming Note: Manipulation of an HT entry associated with active PD entries can produce unpredictable results. Prior to changes, all associated PD entries should be made inactive and invalid. See the next topic.

MODIFICATION OF PRIMARY DIRECTORY ENTRIES

Entries in the storage PD can be accessed and modified by IMP instructions. The effects of any manipulation by an IMP instruction of a PD entry and the recording of its contents in the LB (lookaside buffer) depend on whether the entry is valid and whether a copy of the entry exists in the LB; that is, whether the PD entry is active.

When an inactive, invalid PD entry is made valid, the change to valid takes place immediately. Also, when an inactive, valid PD entry is made invalid, the change to invalid takes place immediately.

A change to an active PD entry, one for which a copy exists in LB, can take effect for implicit translation any time after the instant of the change (when the entry becomes inactive). An Invalidate Primary Directory Entry instruction is used to invalidate a PD entry (see Chapter 10).

REFERENCE AND CHANGE RECORDING

Reference recording provides information for use in selecting storage blocks for page replacement. Change recording provides information as to which pages have to be saved in backing storage when they are replaced in storage by new pages. Both reference and change recording are done by the processor as part of virtual address translation.

PD entry bit 41, the reference bit, is set each time the entry goes from the inactive state to the active state. This occurs whenever the entry is inactive when a location in the page contained in the corresponding page frame is referred to for either the storing or fetching of information. The PD entry bit 42, the change bit, is set each time information is stored in the corresponding page contained in that page frame.

Change recording in the primary directory is accurate only for inactive PD entries. After the initial reference to a page, address translation is performed by means of references to the LB. Change recording takes place in the LB without corresponding updates of the change bit in the PD entry in storage until such time as that PD entry becomes inactive.

An active PD entry becomes inactive when it is purged from the lookaside buffer. It can be made inactive implicitly by the processor as a result of translations involving other PD entries, and explicitly by executing an Invalidate Primary Directory Entry instruction, a Read Reference and Change and Reset Reference instruction, or in some cases, an Examine Primary Directory Entry instruction (see Chapter 10).

The current state of the reference and change bits can be obtained for any PD entry by executing a Read Reference and Change and Reset Reference instruction (see Chapter 10).

Chapter 9. Machine Support Functions

This section provides detailed descriptions of certain facilities of VMC (vertical microcode) that enhance the efficiency, utility, and programmability of the machine. Included are the facilities for IMPL (initial microprogram load), monitoring, timers, machine control, and machine check.

Initial Microprogram Load

Initial microprogram load (IMPL) provides for the initiation of processing when the contents of storage are not suitable for processing. Invoking the IMPL function causes information to be read from a selected input device (usually the disk file), into preassigned storage locations. The information read in is the minimum amount required to execute an IMP task. The IMPL function can be invoked whenever system power is up.

The IMPL function is started by the hardware/HMC (horizontal microcode) and is completed by the IMP. This document addresses only the HMC portion and the transition to the VMC portion.

IMPL in hardware/HMC performs three functions: (1) some basic hardware tests and initialization; (2) loading of control storage and a small portion of main storage; and (3) HMC initialization. Several initialization/configuration parameters are required by these functions and are stored on the IMPL device. Function 1 requires main storage and control storage configuration parameters. Function 2 requires the size of the VMC load to be stored in main storage. Function 3 requires the following parameters to be stored at known locations within the VMC load (see the control address table described in Figure 2-2): valid addresses for the PD (primary directory), the HT (hash table), and the HMC overlay area and size parameters for the PD and the HT.

As part of the hardware test, main storage is functionally tested. Any page frame found to be failing will be flagged invalid in the main storage defective frame table. Previously defective pages, as logged in the main storage history table (cylinder 0, head 0, sector 3 of the file) are flagged in the main storage defective frame table. A swap is made, if necessary, to ensure that the VMC nucleus area contains no defective frames. The main storage status word is updated showing whether a swap was performed. If a defect-free VMC nucleus area cannot be attained, the system halts and the sequence indicator lights on the CE/Op panel are lit. The main storage defective frame table contains 512 bytes in Models 3, 4, and 5; 1024 bytes in Models 6 and 7; and, 2048 bytes in Model 8. Each bit represents one main storage page frame (hex 0 bit = good, hex 1 bit = defective). The address of the main storage defective frame table is found in the control address table (see Figure 2-2) at SID (segment identifier) hex 0000 0100 offset hex 0002-0007.

The transition after the HMC initialization to the VMC code (part of the VMC load) is accomplished by the HMC task dispatcher code switching to the VMC IMPL task code. Required to accomplish this are two addresses in the control address table and a preinitialized TDQ (task dispatching queue) and TDE (task dispatching element). The two addresses needed are for the TDQ and any current TDE address. The preinitialized TDQ contains a pointer to the preinitialized TDE belonging to the VMC IMPL task. This TDE is preinitialized as follows: no pending exceptions; ILC (instruction length count) equal to zero and CC equal to zero (hex 08); initial values for all 16 IMP base registers (base register 0 must point to the VMC IMP task code space); and an IAR (instruction address register) value pointing to the first VMC instruction to be executed.

Program Event Monitoring

A program event is recognized whenever the task dispatcher is enabled and the microprocessor determines that the initial byte of an instruction is located within a particular range of virtual addresses. The range is specified by the PEM (program event monitor) start address and PEM stop address fields of the TDE (task dispatching element). Bytes C-D (control mode), bit 6 of the TDE selectively enables or disables the PEM range check performed while fetching each successive IMP instruction.

Once a task has been dispatched, alteration of bit 6 is not detected until the task is dispatched at a later time. The PEM range is not checked if the instruction is altered by an Execute instruction.

A PEM exception is presented as follows:

- If bit 8 of the TDE exception mask field is 1, the instruction is nullified and the exception SVL (supervisor linkage) mechanism is invoked.

- If bit 8 is 0, it is set to 1; no exception is generated. The instruction is completed and the next instruction is fetched and checked for being within the PEM range.

This allows the PEM IMP exception handler to cause a nullified instruction to be completed without altering the PEM start and stop addresses in the TDE.

Programming Note: A Dispatch Task Dispatching Queue instruction can be used to cause bit 6 of the TDE control mode field to be reinspected or to reload the PEM registers from the TDE.

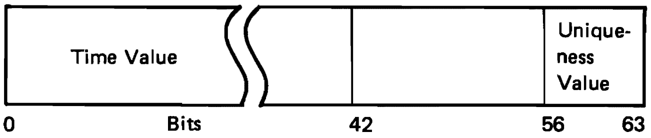
Internal Microprogramming Timer Support

The processor provides these support timing functions: a time-of-day clock, a clock comparator, and two interval timers.

TIME-OF-DAY CLOCK

The time-of-day clock provides date and time. The cycle of the clock is approximately 143 years.

The time-of-day clock is a binary counter with a format as shown in the following figure. The bit positions of the clock are numbered 0 to 63, corresponding to the bit positions of an unsigned binary doubleword. Time is measured by incrementing the value of the clock, following the rules for unsigned binary arithmetic.



The clock is incremented by adding a 1 in bit position 41 every 1024 microseconds. When incrementing of the clock causes a carry out of bit position 0, the carry is ignored and counting continues from zero. No exception condition is generated as a result of the overflow.

The clock can be inspected by means of the instruction, Store Time-of-Day Clock, which causes the bits corresponding to the bits being updated to be stored. In order to ensure that successive executions do not provide the same clock value, the Store Time-of-Day Clock instruction causes a one bit to be added to bit position 63 every time the instruction is executed. Any carry from bit position 56 is ignored. Thus, the rightmost 8 bits of the stored value contain a number that is used to provide uniqueness and is not a part of the actual clock value.

The clock can be set to a specific value by means of the Set Time-of-Day Clock instruction, which causes bits corresponding to the bits being updated to be replaced with the operand designated by the instruction. If a Store Time-of-Day Clock instruction is issued before the Set Time-of-Day Clock instruction, an unpredictable result is stored.

CLOCK COMPARATOR

The clock comparator provides a means of determining when the TOD (time-of-day) clock has passed a specified value. The clock comparator has the same format as the TOD clock, and only those bits that correspond to the clock bits being incremented participate in the compare.

The clock comparator can be inspected by means of the Store Clock Comparator instruction and can be set to a specific value by means of the Set Clock Comparator instruction. The address of the location of the target send/receive counter, when the time-of-day clock value is equal to or greater than the specified value, is contained in the control address table shown in Figure 2-2. The results of a compare are unpredictable if a Set Clock Comparator instruction is issued before the TOD clock is set. If the value specified in the Set Clock Comparator instruction is less than the current value in the TOD clock, the value is loaded in the clock comparator and a send count is issued immediately after the Set Clock Comparator instruction.

INTERVAL TIMERS

Two interval timers provide the means for measuring elapsed time and determining when a prespecified amount of time has elapsed. The first interval timer is known as the task interval timer and is used by the processor for task timing. (See *Task Dispatcher Enable/Disable Functions* in Chapter 5.) The second interval timer is for general use.

Each interval timer is a binary counter with a format that is the same as that of the time-of-day clock and is decremented by subtracting 1 from bit position 41 every 1024 microseconds. Both interval timers and the time-of-day clock are stepped at the same rate.

The mechanism used to indicate that an interval timer has been decremented from a positive number (including zero) to a negative number is different for each interval timer. For the task interval timer, an exception is recognized. For the second interval timer, an SRC (send/receive counter) identified in the control address table (see Figure 2-2) is the target for a SENDC operation.

The interval timers can be inspected by means of the Store Interval Timer instruction and can be set to a specific value by means of the Set Interval Timer instruction. When the second interval timer is specified, the Set Interval Timer instruction indicates whether the time interval is to be repetitive. A repetitive time interval can be specified such that the value in the interval timer is reset to the value contained in the repetitive interval timer doubleword when the prior interval is decremented through zero. See Figure 2-2 for the location of the repetitive interval timer doubleword. This doubleword must begin on a doubleword boundary, and be resident in storage or a machine check will occur when it is used. The repetitive interval timer doubleword must be set prior to issuing a Set Interval Timer instruction or a previous value can be used if repetitive timing is specified.

If an untimed task issues a Set Interval Timer instruction to the task interval timer, a specification exception is presented. A Store Interval Timer instruction issued by an untimed task to the task interval timer stores unpredictable results.

Programming Note: After the indication has been given that an interval timer has been decremented through zero, the interval timer continues to decrement. Thus, a Store Interval Timer instruction can store a negative number, because the interval timer format is the same as an unsigned binary doubleword, which is represented as a large positive number.

System Control

The system console and the operator/service panel provide external control or alteration of the processor.

The system console:

- Displays requested machine status
- Provides operator-to-machine (or service personnel-to-machine) communication
- Provides controls required by the operator (or service personnel) to intervene in normal programmed operation

The operator/service panel and the SCA (system control adapter) provide:

- Means for the control and indication of power
- System status lights
- Operator control, such as,
  - IMPL (initial microprogram load)
  - Alternative IMPL
  - CPU Start
  - CPU Stop
- Controls power to devices such as the printer for concurrent maintenance

Some controls are for the use of service personnel only.

The SCA has a direct interface to the processor. This interface is described in the *System Control Adapter Theory-Maintenance* manual.

The SCA functions as follows:

- When possible, it presents menus to service personnel on the system console.
- It accepts responses from the operator/service panel or keyboard.
- It uses the queue structure that is part of the processor.

The SCA diagnoses system problems. The operator/service panel and the SCA assist in maintaining the dedicated portion of the system.

The diagnostic task, using routines written into and executed by the SCA, enables concurrent maintenance to be performed on a portion of the system.

SYSTEM CONTROL ADAPTER

The system control queue is the operational unit queue of the SCA (system control adapter). OU (operational unit) number 1 is the value of the OU field of an ORE (operation request element) that selects the SCA. The command field for the SCA is as follows:

Command Field (Hex)	**Description**
01	Write data-RAM2
02	Read data-RAM2
04	Reset SCA
0C	Start up
12	Read rotary switches
19	Write IOC LSR/data store
1C	Diagnostic write
21	Write control-RAM1
2A	Read IOC LSR/data store
2C	Diagnostic head
34	Execute-RAM1
44	Terminate routine

The diagnostic task provides functions for system maintenance using routines that are written into and executed by the SCA. Some of the functions are:

- Timing tests
- Power down of individual devices
- Test patterns
- Instruction test/address stop

Machine Check

The machine-check function provides a mechanism for handling detected machine malfunctions that can occur in hardware or HMC. A description of the malfunctions handled by the machine check function are given under *Machine Check Logout* later in this chapter. A machine check is reported as either a soft machine check report (error corrected) or a hard machine check report (error not corrected).

Soft and hard machine checks (called IMP machine checks) are reported to the IMP by the PMCH (processor machine check handler). The status data for machine checks are logged into the MCLB (machine check log buffer) by the PMCH. See *Machine Check Log Buffer* for the description of the format and contents of the MCLB. The address of the MCLB as specified in the control address table (see Figure 2-2) must be fullword aligned. If not properly aligned, a second machine check will cause the processor to enter check stop mode.

Once the machine check has been reported by the PMCH, the data in the MCLB is used to determine the response. After this response has been taken, the MCLB is cleared to zeros and the machine check mode is reset, thereby clearing the status of the MCLB.

MACHINE CHECK HANDLING

Machine checks are reported to IMP whenever:

- A malfunction is detected below the IMP instruction set.

- An exception condition occurs and the task dispatcher is disabled. See Chapter 5.

- A Terminate Immediately instruction is issued and the machine is not in machine check mode. See Chapter 10.

- An error exists for some VMC objects that are referenced by an IMP instruction (for example, when the TDE [task dispatching element] or TDQ [task dispatching queue] are not aligned to a fullword). See Chapters 5 and 6.

- Any of the following instructions are executed when the task dispatcher is disabled. See Chapters 5 and 6.
  - Receive Message
  - Receive Count
  - Dispatch Task Dispatching Queue
  - Supervisor Linkage:
    Implicit SVL
    Explicit SVL
    Exception SVL

Check Stop

In some situations, it is either impossible or undesirable to continue processor operation when a machine check occurs. When these situations arise, the processor stops all processing and goes to the check stop state. See Chapter 4 for the definition of processor states.

In the check stop state, the processor executes no instructions, the interval timers and TOD clock are not updated, and channel operations are suspended.

Check Stop Initiated by HMC

There are two sources for a check stop by HMC. The first is a machine error occurring while an instruction is being retried by the HMC because of an earlier error. The second is a hard machine error that cannot be reported because the MCLB (area in storage where machine checks are logged by the PMCH) contains a machine check report from a previous error. In each case, the check stop is caused by machine errors occurring faster than they can be processed. In these situations, all processor operations stop (including microprocessor, virtual address translator, and channel), the SCA (system control adapter) is informed, and the SCA displays the state of the processor on the machine CE/Op panel sequence indicators. IMPL (initial microprogram load) is required to remove the system from the check stop state.

Check Stop Initiated by IMP

An IMP procedure can put the processor into the check stop state, when an IMP procedure has determined that error conditions exist such that the IMP processing is no longer feasible or desirable. In this instance, the Terminate Immediately instruction is issued by an IMP procedure (see Chapter 10).

Machine Check Mode

The processor enters the machine check mode whenever a machine malfunction or an IMP machine check is detected. In this mode, the IMP execution characteristics of the processor are altered such that an IMP procedure can be activated without the presence of a tasking structure (current TDE). Rather than performing a task dispatching or SVL (supervisor linkage) function to activate an IMP procedure, the machine check function branches to a routine whose addressability is at offset hex 40 in the control address table (see Figure 2-2). The following text (*Processor Machine Check Handler*) defines the interface used to pass control to the IMP procedure when the processor is in machine check mode. Machine check mode causes an implicit disabling of the task dispatcher before control is passed to an IMP procedure. The restrictions on the machine when the task dispatcher is disabled are defined in Chapter 5. If the restrictions are violated by an IMP procedure, a second machine check occurs, causing the processor machine check function to put the machine into the check stop state. It is the responsibility of the activated IMP procedure to enable the task dispatcher, if desired. The reenabling of the task dispatcher can be performed via the Enable Task Dispatching instruction (reference Chapter 10).

It is also the responsibility of the activated IMP procedure to reset the machine check mode if desired. This function can be performed via the Reset Machine Check Mode instruction (reference Chapter 10).

PROCESSOR MACHINE CHECK HANDLER

The PMCH (processor machine check handler) is a processor HMC routine (built-in function) that:

- Retries hardware-signaled errors
- Loads the MCLB (machine check log buffer) with machine check status information (see *Machine Check Log Buffer*, later in this chapter)
- Disables task dispatching
- Branches to the IMP procedure whose address is at offset hex 40 in the control address table
- Initiates the termination of processing for some error conditions

When the processor encounters a hardware malfunction, it pauses from 1 to 2 milliseconds before trapping to the PMCH. This pause allows any intermittent electrical noise to subside. During this time, the processor hardware determines if the PMCH is being executed at the time of the machine error; if so, the processor enters the check stop state.

When a processor error occurs, the PMCH determines if the error can be retried. A retryable error is an error that occurs in an IMP instruction before source data has been changed, or an error that occurred in an IMP instruction that can be executed again without changing the final results.

If the instruction is successfully retried, the PMCH is activated again to report successful recovery of a machine error. This is a soft machine check report. If the machine is in machine check mode, the soft machine check is not reported and the next sequential IMP instruction is executed. If the machine is not in machine check mode, the PMCH loads the error information into the MCLB whose addressability is at offset hex 38 in the control address table (reference Figure 2-2). The task dispatcher is disabled and an exit is made to the IMP procedure whose addressability is at offset hex 40 in the control address table.

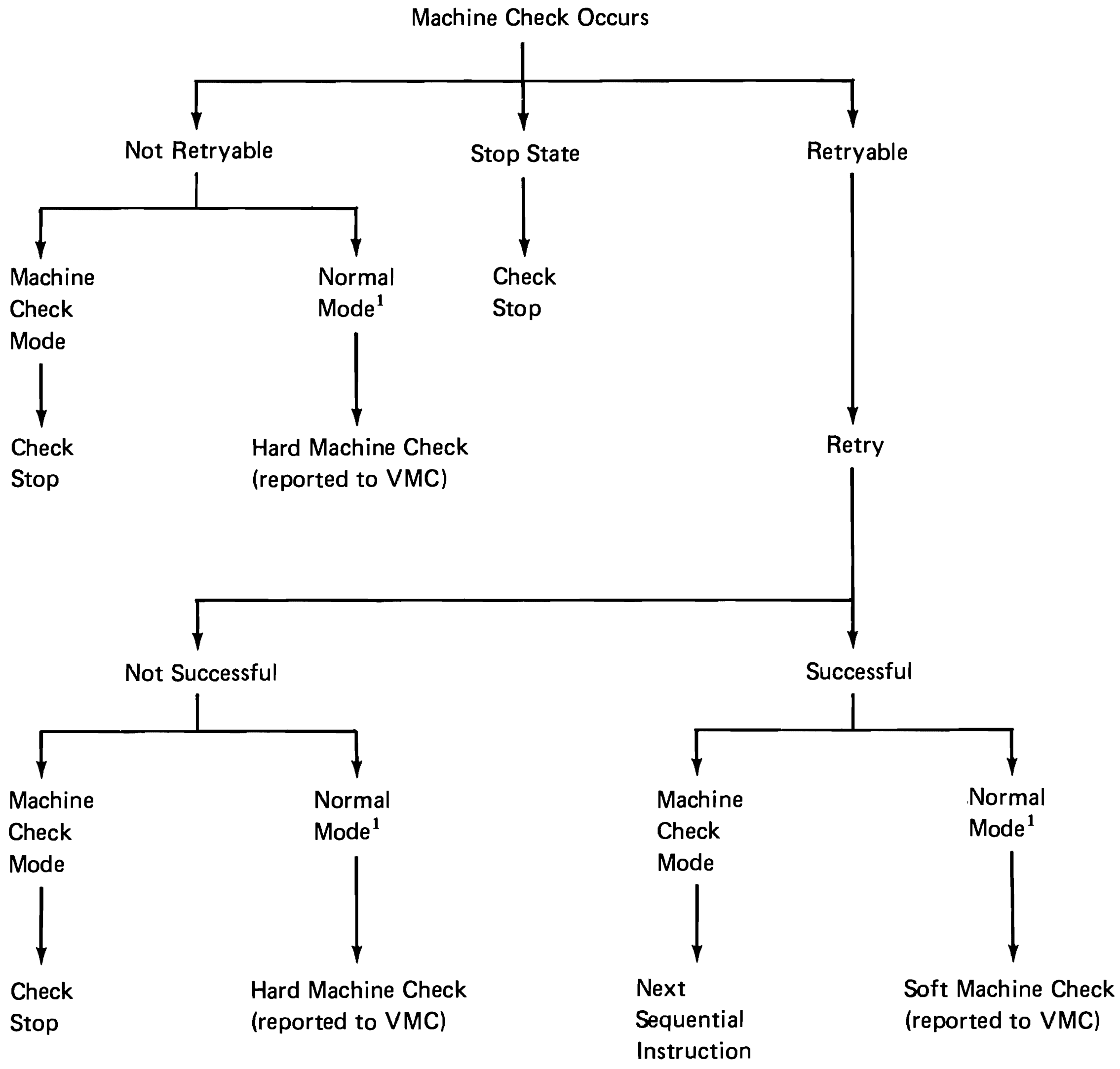
If the instruction retry is unsuccessful or impossible, and the PMCH determines that the machine is not in machine check mode, the PMCH moves the machine check error status information into the MCLB. The task dispatcher is disabled and control is passed to the IMP procedure whose addressability is at offset hex 40 in the control address table. If the machine is in machine check mode, the PMCH puts the processor in the check stop state. At this time, the MCLB contains the earliest hard machine check processor status and the earliest soft or hard machine check task status.

Before the PMCH passes control to the IMP procedure, the PMCH:

- Puts the machine in machine check mode
- Fills the log buffer with the following machine check information:
  - Processor status
  - Task status
- Disables the task dispatcher
- Stops the task interval timer

Machine Check Process Procedures and States

Following is a diagram of the processor machine check procedure and the various states that the machine can be put in by the PMCH.



[1] Normal mode includes the run and wait states when the processor is not already handling a previous machine check.

Stop State Machine Check

If a machine check occurs when the processor is in the stop state, the processor enters the check stop state.

Wait State Machine Check

Two phases of processor activity are possible when it is in the wait state; it can either be active, servicing I/O and timer events, or it can be idle, not servicing I/O or timer events.

If the processor is active and a machine check occurs, the processor exits the wait state, logs the processor and task status into MCLB and reports a hard machine check.

Programming Note: The IMP machine check handler programmers should note that, in this situation, the processor is in the operational state with no TDE present on the TDQ.

The processor maintains internal status indicating a hard machine check while in the wait state. This status is used by the processor in the following manner when the Enable Task Dispatching instruction is executed:

- If there is no TDE on the TDQ, the processor returns to the wait state without storing task status into any TDE.

- If there is a TDE on the TDQ, the processor switches in the new task without first storing task status into any TDE.

If the processor is in the wait state and the operation being executed is retryable (MCLB byte hex E bit 0 = 1) when a machine check occurs, the PMCH logs the processor status and returns to the wait state. The PMCH also sets an internal flag called *soft log required;* thus, when the processor is reactivated (from the wait to run state), the PMCH will regain control. The PMCH will log the currently activated task into the task status and report the processor and task status to the IMP machine check handler.

If the processor is in the wait state and the operation being executed is not retryable (MCLB byte hex 14 bit 0 = 0) when a machine check occurs, the PMCH logs the processor status. If the MCLB was not busy the task status is also logged. However, if the buffer is already busy, no log of the task status occurs. Control is passed to the IMP machine check handler to report this machine check immediately after logging. The PMCH resets the wait state before control is transferred to the IMP machine check handler.

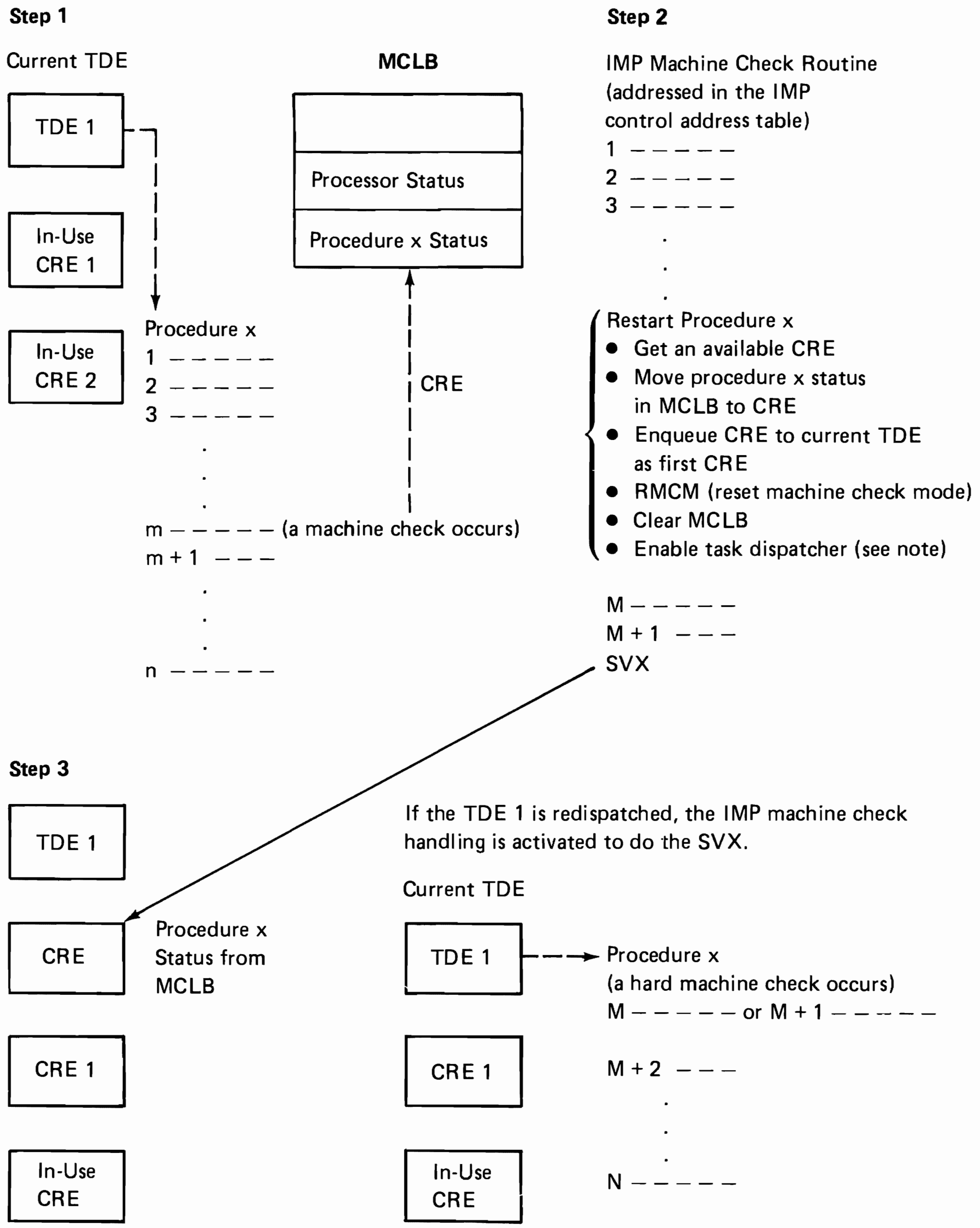
If the processor is in the wait state and the operation being executed is retryable but has been unsuccessfully retried by the PMCH, the PMCH logs the task status, resets the wait state, and transfers control to the IMP machine check handler.

Programming Note: The task status may not be valid or consistent with the processor status for wait state machine checks.

Restart Task if Machine Check Is in Run State

The task status section of the MCLB provides a mechanism for restarting the IMP procedure that was executing at the time of the machine check, if the processor was in the run state. The IMP machine check procedure can determine the state of the machine at the time of the machine check by referencing byte hex 15 bit 4 of the MCLB (see *Processor Status* in this chapter). If this flag is set to zero, the processor was in the run state when the machine check occurred. In this instance, the current TDE that is addressed at offset hex 48 in the control address table is the task that was active at the time of the machine check. If the task is to be restarted following a machine check, the task information in the MCLB must be moved to a CRE; that CRE is marked as the first CRE to get control following an SVX instruction. The IMP machine check procedure uses the SVX instruction to pass control to the IMP procedure that was active at the time of the machine check.

The following illustrates the steps that should be performed by the IMP machine check processor to restart the IMP procedure that incurred a machine check:



Restart Procedure x

- Get an available CRE
- Move procedure x status in MCLB to CRE
- Enqueue CRE to current TDE as first CRE
- RMCM (reset machine check mode)
- Clear MCLB
- Enable task dispatcher (see note)

If the TDE 1 is redispatched, the IMP machine check handling is activated to do the SVX.

Note: The Enable Task Dispatcher instruction does one of two things:

1. Continues processing without a task switch, thereby using processor status associated with the IMP machine check handling procedure; or
2. Performs a task switch, causing the processor status associated with the IMP machine check handling procedure to be stored in the dispatcher's CRE section of the TDE 1. See Chapter 10 for the Enable Task Dispatcher and Reset Machine Check instructions.

MACHINE CHECK LOG BUFFER

Machine check reports are found in the MCLB (machine check log buffer) whose addressability is at offset hex 38 in the control address table (see Figure 2-2). Figure 9-1 illustrates the format of the MCLB. The MCLB information contains the type of error that occurred, an indication of whether it was recovered or not, and the status of the hardware and processor at time of error along with the status of the procedure executing at the time of the error. This information is divided into two categories of status: processor status and task status.

Task status contains the status associated with the task whose address is in the current TDE location, offset hex 48 in the control address table. The information in the task status of the MCLB is valid only when the processor is in the run state. If the processor is in the wait state, the processor status associated with the current TDE is invalid because it reflects the processor status in the wait state at the time of the machine check.

Machine Check Log

When a machine check occurs, the data is logged out to the MCLB (machine check log buffer). Depending on the prior state of the MCLB, the following condition will be logged:

- Both the processor status and the task status;
- The task status only; or
- Neither processor status or task status.

The format of the MCLB is described in the text that follows.

Processor Status

The processor status field of the MCLB contains the information needed to determine the type and severity of the machine check. It also contains information that indicates the state of the HMC at the time of the error. This field contains 44 bytes.

	Bytes	0	1	2	3
Processor Status	0	Error Type	VMC Flags	MCHK Designation (primary)	MCHK Designation (secondary)
	4	Hardware Code			
	8	Main Storage Error Code			
	C	Retry Indicator	S-Register	G-Register	Operation Code Extender
	10	Operation Code	Current MCLB Status	MCSAR	
	14	HMC Flags			
	18	HMC Exceptions		Overlay Index A/B	
	1C	IMP Exceptions		Condition Code	Reserved
	20	Lookaside Buffer Miss Control Address		Link Register	
	24	Address Register		Reserved	Instruction Length
	28	Exception Register		Overlay Index C/D	
Task Status (CRE)	2C	CRE Flags	Base Register Specification	Instruction Length	Condition Code
	30	Address Register			
	34	Base Register (0–F) (16 x 6 = 96 bytes)			
	90			Failing V=V Address (all 5382 System Unit models)	
	94	These bytes are reserved for all other models.			
	98	IMP Exceptions			

Figure 9-1. Machine Check Log Buffer

Processor Status (continued)

Bytes (Hex)	Bits	Descriptions
0		**Error Type:**
	0	HMC-detected errors: See *HMC Flags* (bytes 14-17) for more information on specific errors. Also see *Lookaside Buffer-Miss Control Address* (bytes 20-21) and *Link Register* (bytes 22-23) for additional information.
	1	Channel checks: Secondary channel errors. These are main storage or VAT (virtual address translator) errors reported via the microcode. See *Hardware Code* and *Main Storage Error Code* (bytes 4-B) for more information.
	2	Microprocessor hardware-detected error: See *Hardware Code* and *Main Storage Error Code* (bytes 4-B) for more information.
	3	FIB (fill instruction buffer) error: Error occurred while trying to fill the instruction buffer.
	4	System damage (HMC procedure): Machine checks occurred during an operation that cannot be isolated to a specific task. See *Hardware Code* (bytes 4-6) for more specific information.
	5	Instruction-processing damage (IMP procedure): Errors that can be isolated to a particular task. See *Hardware Code* (bytes 4-6) for more specific information.
	6	Recovery report: If this bit is set and the secondary designation (byte 3 of MCLB) indicates a value of hex 2A, successful recovery has been made from an initial error. This bit can also be set as a result of a successful recovery report attempted due to a second recoverable error while handling the initial error. In this case, the second error report is suppressed with this condition being identified by the bit being set and the secondary designation (byte 3 of MCLB) indicating a value of hex 2B. See *Hardware Code* for additional information about the error.
	7	Timer errors: Error that occurred in any element of timing. *Programming Note*: All the timers being used (including the TOD) must be reinitialized following a timer error.
1		**VMC Flags:**
	0	0 *Not* VMC machine check. 1 VMC machine check.
	1-7	Reserved: May be any value. **Note:** HMC initializes this byte to hex 00.

Bytes (Hex)	Bits	Description
2		**MCHK Designation–Primary**: See the following diagram for the designation number. Each designation number uniquely identifies the facility which detected the machine check, the state of the processor, the functions performed by the PMCH (processor machine check handler), and the exit from the PMCH. As an example, if the designation number found in the MCLB was hex 21, the general facility that reported this machine check was microcode. If this column is followed down to where the first X appears, the processor was in the run state when the machine check occurred. The next X indicates that the retry indicator was 0. The next X indicates that the MCLB was not busy. The next group of Xs indicate what should have been logged as well as other functions that should have been set or reset such DTD = 0 (disable task dispatcher). Further down are the exits from PMCH to checkstop, IMP machine check handler (startup), or go to wait state. In this case the machine check would have reported to IMP via startup.

Hex Code	
00-16	Hardware facility
1B-24	Microcode facility
26-29	Channel facility
2A-2B	Recovery-Soft
2C-32	Timer facility
36-37	Channel facility

	FACILITY --->	HARDWARE																				
	ENTRY POINT --->	MHDWRE																				
P	DESIGNATION NUMBER ---> (hex)	I*	S*	__	00	01	__	__	04	05	06	__	__	08	09	0A	CB	0C	0D	0E	0F	__
RS	IMPL	X																				
OT	STOP		X		X	X																
CA	WAIT								X	X	X							X	X	X	X	
ET	RUN (IMPLIED)													X	X	X	X					
SU																						
SS	CAV		0						0	0	0			0	0	0	1	1	1	1	1	
O																						
R	SCA XFER IN PROC		X																			
	RETRY INDICATOR = 0																X	X	X			
	RETRY INDICATOR = 1																			X	X	
	FIB ERRCR																					
	NOT BUSY		X		X				X					X			X	X		X		
**	BUSY SOFT		X		X					X					X				X		X	
	BUSY HARD		X			X					X					X			X		X	
	LOG PROCESSOR STATUS				X				X	X				X	X		X	X		X		
	LOG RETRY INDICATOR				X				X	X				X	X					X		
	LOG CRE DATA				X				X	N				X	N					X		
	SET RETRY INDICATOR = 0														X					X		
	SET RETRY INDICATOR = 1																X	X	X			
	DTD = 0								X	X				X	X					X		
R	MCLB BUSY = S																					
E	MCLB BUSY = H				X				X	X				X	X					X		
S	RESET WAIT STATE								X										X	X		
P	SOFT LOG REQUIRED = 0								X	X				X						X		
O	SOFT LOG REQUIRED = 1																X	X	X			
N	RESET AUTO ERROR IN VAT																					
S																						
E	RESET CHAN ERROR IN VAT																					
	RESET/RESTART EVERYTHING								X					X			X	X	X	X		
	SUBTRACT IL FROM IAR (ECC)																X		X			
	IL = 0																X		X			
	STOP TASK INTERVAL TIMER								X					X			X		X	X		
	SET IAR TO IMCH (IL = 0)								X					X						X		
	RESERVED																					
	SET WAIT STATE MCHK FLAG								X	X	X								X	X	X	
	UHOO CODE TO SCA	X	X		X	X				X	X				X	X					X	
	CHECKSTOP STATE	X			X	X				X	X				X	X					X	
E	RETURN TO SCA OR FIX UP		X																			
X	START UP W/AT (ESRT.ESRTAT)								X					X						X		
I	START UP W/NT (ESRT.ESRTNT)																X		X			
T	GO TO STOP STATE (ESP1.ESCAO)																					
	GO TO WAIT STATE (TWTL.TWTLNE)																	X				
	RETURN TO CALLER																					

* Designation I = IMPL State - No Logcut
Designation S = SCA Transfer in Process - No Logout

** indicates MCLB STATUS

X = Condition or Response
N = No Response
blank = No Condition or No Response

Processor Machine Check Handler State (Part 1 of 3)

	FACILITY --->	HARDWARE												CHANNEL								
	ENTRY POINT --->	MHDWEE												MCHER								
P	DESIGNATION NUMBER ---> (hex)	10	11	12	13	14	15	16						26	27	28	29	36	37			
RS	IMPL																					
OT	STOP																	X	X			
CA	WAIT													X	X							
ET	RUN (IMPLIED)	X	X	X	X	X	X	X								X	X					
SU	------------------------------																					
SS	CAV	1	1	1	1																	
O	------------------------------																					
R	SCA XFER IN PROC																					
	RETRY INDICATOR = 0		X																			
	RETRY INDICATOR = 1	X		X	X																	
	FIB ERROR					X	X	X														
	NOT BUSY	X				X								X		X		X				
**	BUSY SOFT		X	X			X								X		X		X			
	BUSY HARD		X		X			X							X		X		X			
	LOG PROCESSOR STATUS	X		X		X	X							X		X		X				
	LOG RETRY INDICATOR	X		X		X	X							X		X		X				
	LOG CRE DATA	X		N		X								X		X		X				
	SET RETRY INDICATOR = 0	X		X			X															
	SET RETRY INDICATOR = 1		X																			
	DTD = 0	X		X		X	X							X	X	X	X					
R	MCLB BUSY = S																					
E	MCLB BUSY = H	X		X		X	X							X		X		X				
S	RESET WAIT STATE													X	X							
P	SOFT LOG REQUIRED = 0	X		X		X																
O	SOFT LOG REQUIRED = 1		X																			
N	RESET AUTO ERROR IN VAT					X																
S	------------------------------																					
E	RESET CHAN ERROR IN VAT													X	X	X	X					
	RESET/RESTART EVERYTHING	X	X			X																
	SUBTRACT IL FROM IAR (EOC)		X																			
	IL = 0		X																			
	STOP TASK INTERVAL TIMER	X	X			X								X	X	X	X					
	SET IAR TO IMCH (IL = 0)	X				X								X	X	X	X					
	RESERVED																					
	SET WAIT STATE MCHK FLAG													X	X							
	UHOO CODE TO SCA			X	X		X	X										X	X			
	CHECKSTOP STATE			X	X		X	X										X	X			
E	RETURN TO SCA OR FIX UP																					
X	START UP W/AT (ESRT.ESRTAT)	X				X								X	X	X	X					
I	START UP W/NT (ESRT.ESRTNT)		X																			
T	GO TO STOP STATE (ESP1.ESCA0)																					
	GO TO WAIT STATE (TWTL.TWTLNE)																					
	RETURN TO CALLER																					

* Designation I = IMPL State - No Logout
Designation S = SCA Transfer in Process - No Logout

** indicates MCLP STATUS

X = Condition or Response
N = No Response
blank = No Condition or No Response

Processor Machine Check Handler State (Part 2 of 3)

	FACILITY ——>	MICROCODE											RECOVERY			TIMERS						
	ENTRY POINT ——>	MUCODE											MSOFT			MTIMER						
	DESIGNATION NUMBER ——> (hex)	1B	1C	1D	1E	1F	20	21	22	23	24		2A	2B		2C	2D	2E	2F	30	31	32
PROCESSOR STATUS	IMPL	X																				
	STOP		X	X												X						
	WAIT				X	X	X										X	X	X			
	RUN (IMPLIED)							X	X	X	X		X	X						X	X	X
	CAV		0	0	0	0	0															
	SCA TRANSFER IN PROCESS																					
	RETRY INDICATOR = 0							X														
	RETRY INDICATOR = 1								X													
	FIB ERROR																					
**	MCLB NOT BUSY		X		X			X	X				X				X			X		
	MCLB BUSY SOFT		X			X				X				X				X			X	
	MCLB BUSY HARD			X			X				X			X					X			X
RESPONSE	LOG PROCESSOR STATUS	X	X		X	X		X	X	X			D	D		X	X	X		X	X	
	LOG RETRY INDICATOR		X		X	X		X	X	X						X	X	X		X	X	
	LOG CRE DATA		X		X	N		X	X	N			X				X	N		X	N	
	SET RETRY INDICATOR = 0								X	X			X	X								
	SET RETRY INDICATOR = 1																					
	DISABLE TASK DISPATCHER				X	X		X	X	X			X	N		X	X	X		X	X	
	MCLB BUSY = SOFT												X									
	MCLB BUSY = HARD		X		X	X		X	X	X						X	X	X		X	X	
	RESET WAIT STATE				X	X	X										X					
	SET SOFT LOG REQUIRED = 0				X	X		X	X	X			X	X			X	X		X	X	
	SET SOFT LOG REQUIRED = 1																					
	RESET AUTO ERROR IN VAT															X	X	X	X	X	X	X
	RESET CHAN ERROR IN VAT																					
	RESET/RESTART EVERYTHING				X			X	X								X			X		
	SUBTRACT IL FROM IAR (EOC)																					
	IL = 0																					
	STOP TASK INTERVAL TIMER		X	X	X		X	X	X							X	X	X	X	X	X	X
	SET IAR TO IMCH (IL = 0)				X			X	X				X				X			X		
	RESERVED																					
	SET WAIT STATE MCHK FLAG				X	X	X										X	X	X			
	UHOO CODE TO SCA	X	X	X		X	X			X	X					X		X	X		X	X
EXIT	CHECKSTOP STATE	X	X	X		X	X			X	X					X		X	X		X	X
	RETURN TO SCA OR FIX UP																					
	START UP W/AT (ESRT.ESRTAT)				X			X	X								X			X		
	START UP W/NT (ESRT.ESRTNT)																					
	GO TO STOP STATE (ESP1.ESCA0)																					
	GO TO WAIT STATE (TWTL.TWTLNE)																					
	RETURN TO CALLER												X	X								

** indicates MCLB STATUS

X = Condition or Response
D = Hard-hard double bit error corrected condition
will be ORed into processor status
N = No Response
blank = No Condition or No Response

Processor Machine Check Handler State (Part 3 of 3)

Processor Status (continued)

Bytes (Hex)	Bits	Description
3		**MCHK Designations–Secondary**: Designation number if a second MCHK occurred before the first was handled or indicates the recovery designation number after a recovery log has been executed.
4-6		**Hardware Code**: Specific hardware error. Note that byte 6 has meaning only when byte 5, bit 1 is a 1; and, if byte 6, bit 0 is a 1. Additional information is contained in the *Main Storage Error Code* (bytes hex 7-B).
4	0-7	Reserved: Must be zero.
5	0	Main storage time-out error (Models 3, 4, and 5). Reserved: Must be zero (5381 Models 6, 7, and 8; all 5382 models).
	1	VAT (virtual address translator) machine check.
	2	ALU (arithmetic logic unit) check.
	3	Reserved: Must be zero.
	4	Control storage read data parity check.
	5	ALU output parity check.
	6	IAR (instruction address register) parity check.
	7	Reserved: Must be zero (all 5382 models). Invalid control storage address (all other models).
6	0	Main storage error.
	1	VAT internal parity or VAT error during HMC request (such as no traps, invalid register or page, or store without set change bit set).
	2	VAT error during channel request.
	3	VAT error during automatic operation (FIB/TOD/IT) (fill instruction buffer time/interval timer). An FIB error is detected as VAT hardware machine check. TOD/IT error is detected by microcode.
	4	Fetch/Store command when address compare was made.
	5	Reserved: Must be zero.
	6-7	Address compare buffer select.
7-B		**Main Storage Error Code (Models 3, 4, and 5)**: Additional main storage error information.
7	0	Read data parity check.
	1	Main storage address parity check.
	2	Main storage write data parity check.
	3	Main storage invalid address.
	4	Main storage multibit failure.
	5	Reserved: Must be zero.
	6	Main storage single bit failure (status only–does not cause a machine check).
	7	MSAR (main storage address register) specification: 0 MSAR2. 1 MSAR1.

Processor Status (continued)

Bytes (Hex)	Bits	Description
8	0-6	Page identifier.
	7	Byte identifier (with byte 9).
9	0-7	Byte Identifier: The byte identifier makes up the low order 9 bits of the 21-bit real address of the failing storage address.
A	0-1	00 I/O access: 01 Data access. 10 Data access. 11 Instruction stream access.
	2	0 Fetch from main storage. 1 Store to main storage.
	3	0 V=V. 1 V=R.
	4-7	Frame identifier (with byte B).
B		Frame Identifier: The frame identifier makes up the high-order 14 bits of the 23-bit real address of the failing storage address for non V=R addresses. For V=R addresses byte hex B, bits 1-7 make up the high-order 7 bits of the 23-bit real address, and bytes 8 and 9 make up the low-order 16 bits. On Models 3, 4, and 5, bits 1 and 2 of Byte B are zero.

Bytes (Hex)	Bits	Description
7-B		**Main Storage Error Code (Models 6, 7, and 8):** Additional main storage error information.
7	0	Data parity error–CPU/channel.
	1-4	Main storage error. See the *IBM System/38 Processing Unit MAP Reference*, P/N 2550526, for details.
	5-6	Reserved: Must be zero.
	7	MSAR (main storage address register) specification: 0 MSAR2. 1 MSAR1.
8	0-6	Page identifier.
	7	Word identifier (with byte 9).
9	0-5	Word identifier.
	6-7	Access type: 00 I/O access. 01 Data access. 10 Data access. 11 Instruction stream access.
A	0	Virtual = Real. 0 Not V=R. 1 V=R.
	1	Fetch/store from main storage. 0 Fetch. 1 Store.
	2-7	Frame identifier (with byte B).

Processor Status (continued)

Bytes (Hex)	Bits	Description
B	0-7	Frame identifier.

Note: Bytes 8, 9, A, and B are used to form a 24-bit real address as follows: If byte A, bit 0 is set:

Address Bits	Bytes and Bits Used	
0-7	B	0-7
8-15	8	0-7
16-21	9	0-5
22, 23	Forced to 00	

If byte A, bit 0 is reset:

Address Bits	Bytes and Bits Used	
0	Forced to 00	
1-6	A	0-7
7-14	B	2-7
15	8	7
16-21	9	0-5
22, 23	Forced to 00	

Bytes (Hex)	Bits	Description
7-B		**Main Storage Error Code (all 5382 Models)** Additional main storage error information.
7	0	Data parity error–CPU/channel.
	1-4	Main storage error. See the *IBM System/38 Processing Unit MAP Reference* P/N 2550526, for details.
	5-7	Reserved: Must be zero.
8	0-1	Access type: 00 I/O access. 01 Data access. 10 Data access. 11 Instruction stream access.
	2	Reserved: Must be zero.
	3	Fetch/store from main storage. 0 Fetch. 1 Store.
	4-6	Failing main storage card number.
	7	Word identifier (with byte 9).
9	0-5	Word identifier.
	6-7	Reserved: Must be zero.
A	0-7	Frame identifier (with byte B).
B		**Frame identifier:** The frame identifier makes up the high-order 16 bits of the 25-bit real address of the failing storage address. Bytes 8 and 9 make up the next 7 bits. The low-order 2 bits are forced to zeroes.
C		**Retry Indicator:**
	0-6	Reserved: Must be zero.
	7	Retry indicator: 0 Successful retry. (If a retry was never attempted, bit 7 always = 0.) 1 Unsuccesful retry.

This page is intentionally left blank.

Processor Status (continued)

Bytes (Hex)	Bits	Description
D		**S-Register**: Processor hardware status register which contains temporary HMC status and condition code.
	0-3	Temporary HMC flags.
	4	Hardware overflow.
	5	High-order result bit of an indirect binary add or subtract.
	6	Carry from the ALU (arithmetic logic unit).
	7	ALU result equal to 0.
E		**G-Register**: Hardware register collection of control latches used by the processor logic and HMC.
	0	Checkpoint address valid. This bit indicates whether the microprogram was checkpointed when a machine check occurred: 0 Instruction *not* retryable. 1 Instruction retryable.
	1	Temporary HMC flag.
	2	Block machine check trap. If this bit is set, a machine check error will not trap the microprogram. This bit is for diagnostic use only.
	3	Temporary HMC flag.
	4	Local storage partition latch. This bit is the high-order address bit for direct addressing.
	5	Temporary HMC flag.
	6	L-register couple control. This bit, when set, indicates the L-register is in the coupled mode.
	7	Stop state indication.

Bytes (Hex) Bits Description

F **Extended Operation Code**: This byte contains the second byte of the instruction if a machine check occurred while executing an instruction with an extended operation code. The low-order 4 bits contain the operation code extender field. This byte is valid only if byte hex 10 is equal to one of the following:

Hex		
	0D	80
	5A	83
	6D	91
	71	AE
	79	BE
		CE

See Chapter 2 for a description of the extended operation code format, and Chapter 10 for the extended operation code assignments.

10 **Operation Code:** This byte indicates the type of operation being performed:

Note: See *Machine Check Special Error Conditions* in this chapter.

00 Built in function.

40 HMC procedure. Any other value not equal to 00, 40, or FF represents the operation code of the IMP instruction currently being executed. See hex byte F for a listof extended operation codes.

Bytes (Hex) Bits Description

11 **Current MCLB Status:**

0 Reserved: May be any value.

1-2 Encoded current MCLB status:

00 Log area not busy.

01 Log area busy with soft MCHK.

10 Log area busy with hard MCHK.

11 Not used.

3-7 Reserved: May be any value.

12-13 **MCSAR (machine check control storage address register):** CSAR address when a hardware-detected failure has occurred (MCLB byte 0, bit 2=1). MCLB hex bytes 12-13 will equal 0 if MCLB hex byte 0, bit 2=0.

14-17 **HMC Flags:** Status of various HMC and VMC facilities as used by HMC. These HMC internal flags are, in general, only modified by HMC routines directly associated with an individual bit or bits.

Bytes (Hex)	Bits	Description
14	0	Flag virtual address mapped in PD (primary directory) when SCA (system control adapter) executes the Set AC (address compare) for VA (virtual address) command: 0 Virtual address mapped in PD. 1 Virtual address *not* mapped in PD.
	1-2	Encoded prior MCHK log busy status: 00 Log *not* busy. 01 Log busy with soft MCHK. 10 Log busy with hard MCHK. 11 Not used. **Note:** These bits represent the status of the log when the MCHK occurred.
	3	Flag interval timer in use: 0 Interval timer *not* in use. 1 Interval timer in use.
	4	Flag TOD (time-of-day clock) in use: 0 TOD *not* in use. 1 TOD in use.
	5	Flag clock comparator in use: 0 Clock comparator *not* in use. 1 Clock comparator in use.
	6	Flag task interval timer in use: 0 Task interval timer *not* in use. 1 Task interval timer in use.
	7	Flag task switch blocked: 0 Task switch blocked. 1 Task switch *not* blocked.
15	0	Flag procedure type executing: 0 IMP procedure. 1 HMC procedure.
	1	Reserved: May be any value.
	2	Flag interval timer repetitive: 0 *Not* repetitive. 1 Repetitive.
	3	Reserved: May be any value.
	4	Flag processor in wait state: 0 *Not* in wait state. 1 In wait state.
	5	IS (instruction step) mask: 0 Allow IS exception. 1 Do *not* allow IS exception.
	6	SCA (system control adapter) routine retryable: 0 *Not* retryable. 1 Retryable.
	7	FIB (fill instruction buffer) window flag: 0 Not FIB retry. 1 FIB retry.

Bytes (Hex)	Bits	Description
16	0-1	Reserved: May be any value.
	2	Soft address compare mode: 0 *Not* soft address compare mode. 1 Soft address compare mode.
	3	SCA exceptions: 0 SCA exception. 1 SCA trap.
	4	Timer MCHK flag: 0 *Not* timer error. 1 MCHK timer error.
	5	MCHK in wait state. 0 MCHK *not* in wait state. 1 MCHK in wait state.
	6	Task-controlled interrupt allowed.
	7	SLVM1 instruction enabled.
17	0-7	**Reserved:** May be any value.

Bytes (Hex)	Bits	Description
18		**HMC Exceptions:**
	0	Flag execute instruction: 0 *Not* execute instruction. 1 Execute instruction.
	1	Flag soft MCHK report pending: 0 *No* soft MCHK report pending. 1 Soft MCHK report pending.
	2	Flag IS (instruction step) mode: 0 *Not* in IS mode. 1 In IS mode.
	3-7	Reserved: May be any value.
19		**Microcode Generated Exceptions:**
	0	Flag task dispatcher call required: 0 *No* task dispatcher call required. 1 Task dispatcher call required.
	1	Flag PEM (program event monitor) mode: 0 *Not* in PEM mode. 1 In PEM mode.

Bytes (Hex)	Bits	Description
19	2-3	Flag AC (address compare) mode:
		00 *Not* in AC mode.
		01 In AC sync.
		10 In AC mode.
		11 In data AC mode.
	4	Event Handler call required:
		0 Normal event processing.
		1 Event handler call required.
	5-7	Reserved: May be any value.
1A-1B		**Overlay Index A/B:** HMC routine that was in the control store overlay area A and B at the time of the machine check.
1C-1D		**IMP Exceptions:** IMP exception code as described under *IMP Exception Codes,* (under *Task Status*), later in this chapter.
1E		**Condition Code:** IMP condition code:
	0-3	Hex 0.
	4-7	Condition code.
1F		**Reserved:** May be any value.
20-21		**LB-Miss Control Address:** Lookaside buffer-miss control address.
22-23		**Link Register:** Control storage link address at the time of failure.
24-25		**Address Register:** These 2 bytes contain the IAR (instruction address register) if an IMP procedure, or the left-justified CSAR (control store address register) if an HMC procedure.
26		**Reserved:** May be any value.
27		**Instruction Length:** IMP instruction length.
28-29		**Exception Register (Models 3, 4, and 5):**
28	0-7	Reserved: Must be zeros.
29	0	I/O channel event: (I/O service required).
	1	I/O channel machine check.
	2	Main storage address compare.
	3	Timer carry occurred.
	4	IMPL (initial microprogram load).
	5	Reserved: May be any value.
	6	SCA (system control adapter) request pending.
	7	Microprocessor exception.

Bytes (Hex)	Bits	Description
28-29		**Exception Register (5381 Models 6, 7, and 8; all 5382 Models):**
28	0-7	Reserved: Must be zeros.
29	0	I/O channel event: I/O service required.
	1	I/O secondary error machine check–Set by the I/O channel logic to indicate that a hard channel error has occurred and the I/O channel has been stopped until error recovery has been completed.
	2	Main storage address compare.
	3	Timer carry occurred.
	4	IMPL (initial microprogram load).
	5	Main storage double-bit error–Data has been corrected in main storage.
	6	SCA (system control adapter) request pending.
	7	Microprocessor exception.
2A-2B		**Overlay Index C/D:** HMC routine that was in the control store overlay area C and D at the time of the machine check.

Task Status

The task status field of the MCLB contains information that indicates the state of the task that was running at the time of the machine check or SCA (system control adapter) request. This field contains 110 bytes.

Bytes (Hex)	Bits	Description
2C		**CRE Flags:** This byte contains special flags used by the HMC logout routine.
	0	0 Valid CRE (call/return element) data has *not* been logged out to this area. 1 Valid CRE data has been logged out to this area.
	1	IMP or HMC task pending: 0 IMP task. 1 HMC task.
	2-7	Reserved: May be any value.
2D		**Base Register Specification:**
	0-3	Number of the first base register to be logged.
	4-7	Number of base registers logged minus 1.
2E		**Instruction Length:** Instruction length, right justified.
2F		**Condition Code:** IMP condition code.

Bytes (Hex)	Bits	Description
30-31		**Address Register**: IAR (instruction address register) if an IMP procedure or the left justified CSAR (control storage address register) if an HMC procedure.
32-91		**Base Registers**: The next 96 bytes contain sixteen 6-byte registers beginning with the base register specified in bits 0-3 of byte hex 2D.
92-97		**Failing V=V Address**: These 6 bytes contain the failing virtual address when a page fault results in a machine check (all 5382 models). **Reserved**: May be any value (all other models).
98		**IMP Exceptions**: Two-byte IMP exception code.
	0-2	Reserved: May be any value.
	3	Soft address compare.
	4	Dispatcher time increment expired (task interval timer).
	5	Monitored ACQ descriptor (SVL [supervisor linkage] receive).
	6	Monitored CRE descriptor (SVL receive).
	7	Monitored TDE (task dispatching element) descriptor (SVL receive wait).

Bytes (Hex)	Bits	Description
99	00	No exception
	02	Invalid descriptor
	04	Busy
	06	Reserved
	08	Allocate page frame
	0A	Monitored SRQ descriptor
	0C	Monitored SRM descriptor
	0E	Monitored TDE descriptor
	10	SRC overflow
	12	Address translation
	14	PEM (program event monitor)
	16	Execute
	18	Specification
	1A	Addressing
	1C	Effective address overflow
	1E	Data
	20	Binary overflow
	22	Binary zero divide
	24	Decimal overflow
	26	Decimal zero divide
	28	Floating-point overflow
	2A	Floating-point underflow
	2C	Floating-point inexact result
	2E	Floating-point zero divide
	30	Operation
	32	Stack
	34	Verify
	36	Chain Conflict
	38	End of Chain
	3A	Edit Digit Count
	3C	Length Conformance
	3E	Edit Mask Syntax
	40	Invalid Segment Group
	42	Floating-point invalid operand
	44	Reserved
	46	Second Chain Search
	48	Conversion
	4A	Invalid floating-point conversion
	4C-74	Reserved
	80	Invalid Segment (note)
	81	Invalid Page (note)
	82	Page Read Error (note)
	83	Invalid Pool State (note)
	84	Invalid Pin Request (note)
	85	Invalid Write Request (note)
	86	Main Store Error (note)
	87-8F	Reserved

Note: Implicit SVL codes. See Chapter 6 for the description of all IMP exceptions.

MACHINE CHECK SPECIAL ERROR CONDITIONS

Virtual Address Translator Machine Checks

Since the decode and execution of IMP instructions is asynchronous with main storage and the VAT (virtual address translator), any MCHK (machine check) occurring on a write to main storage cannot be conclusively isolated to the instruction which caused the MCHK. When this condition is detected, the task which incurred the MCHK must be terminated immediately. The conditions to test for the above MCHK are: byte 5, bit 1 = 1; byte 6, bit 0 = 1; and byte 7 bit 2 = 1.

Machine Check During Translate Instruction

A Translate instruction, due to its special interrupt mode, must always force termination of that task if a machine check occurs while the Translate instruction is retryable. MCLB (machine check log buffer) byte hex 10 = hex CC and byte hex 0E, bit 0 = 0 identify this condition.

Error/Recording Error Definition

Errors defined for the CPU are described in the *Processing Unit Theory-Maintenance* manual.

Stack Handling

An IMP stack is a group of storage areas assigned sequentially within the addressing space. The stack entries provide a means of declaring and using save/work areas in storage with nested programs. These stack entries are handled last-in, first-out.

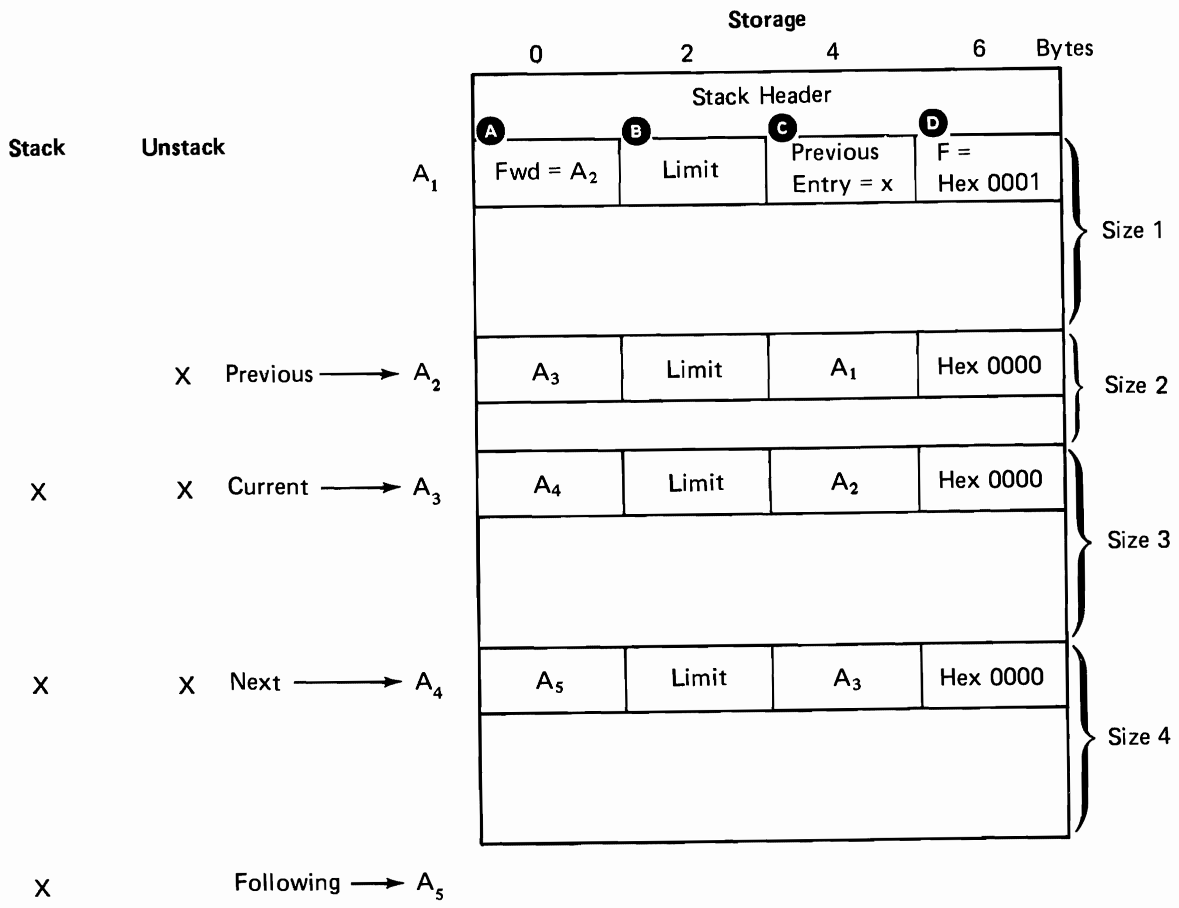
STACK STRUCTURE

An IMP stack is contained within a segment. A software-maintained and used header is found at the front of the stack. The stack entries following this header are variable in length, double-word aligned, and contain an 8-byte area at the front. Two IMP instructions (Stack and Unstack) are used to add and remove stack entries.

The size of a stack entry is presented as the contents of a halfword register in the Stack instruction. The 8-byte area at the beginning of each entry contains 4 halfword fields. The first halfword 🅐 is a virtual address offset (forward pointer) indicating the start of the next stack entry. The second halfword 🅑 is a limit for the stack presented as an upper address offset boundary. The third halfword 🅒 is a virtual address offset (backward pointer) indicating the start of the previous stack entry. The fourth halfword 🅓 is a flag field in which the only IMP-recognized flag is hex bit 15. When set to one, this bit indicates that the stack entry is the first entry on the stack. An unstack operation is not permitted when this flag is set to 1.

The following diagram shows a typical stack. When an IMP stack operation is performed with address A_3 pointing to the entry currently being used, the entry at address A_4 is formed by computing its end address A_5, filling in its 8-byte area, and updating the stack address to A_4 as the new current entry. When the IMP Unstack operation is performed with address A_3 pointing to the entry currently being used (the entry at A_4 has not been created in this case), the stack address is backed up to address A_2 because the flag bit hex 15 is not set to 1.

IMP Stack



HOLD/FREE FUNCTION

The IMP hold/free function is embodied in five IMP instructions and a storage segment containing chained hold records. Each chain represents hold activity for a system object and its hash synonyms. In addition, one chain contains initialized but as yet unused (in other words, available) hold records.

The hold chains contain ordered HRs (hold records) where each record represents an object hold of a specified type. A process (or task) can have holds on multiple objects and can have multiple holds (of the same type or differing types) on the same object.

Hold Chain Structure

Figure 9-2 shows the six data fields involved with hold and free. Two 6-byte addresses are maintained in the processor control address table. One is the address of a 4096-byte HHT (hold hash table) and the other is the address of the first hold record remaining in the preinitialized chain of available (free) records. Two halfword fields in the current TDE (task dispatching element) that are used are a unique TDE ID (task dispatching element identifier) and a count of the number of hold records currently in use by the task.

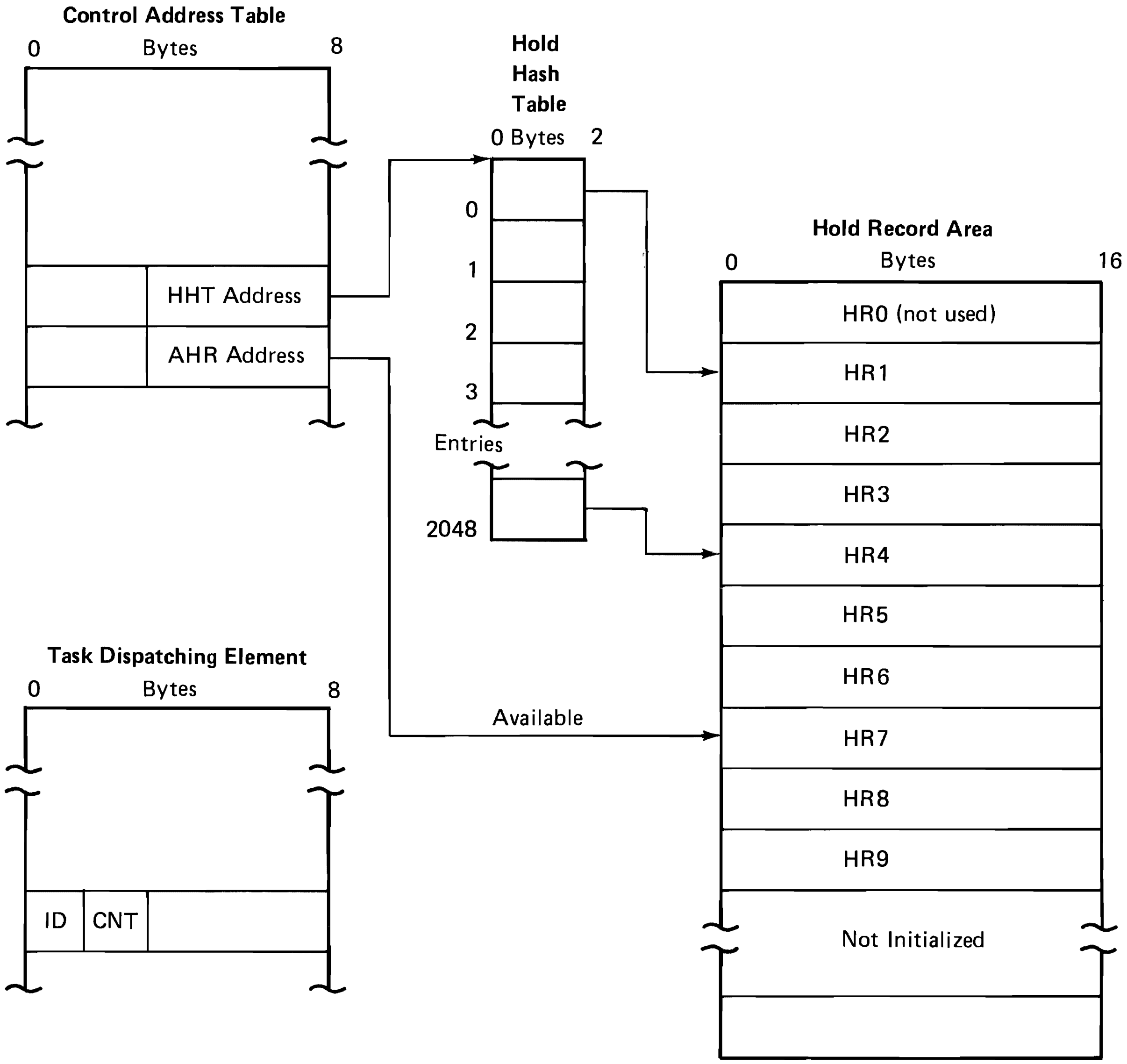
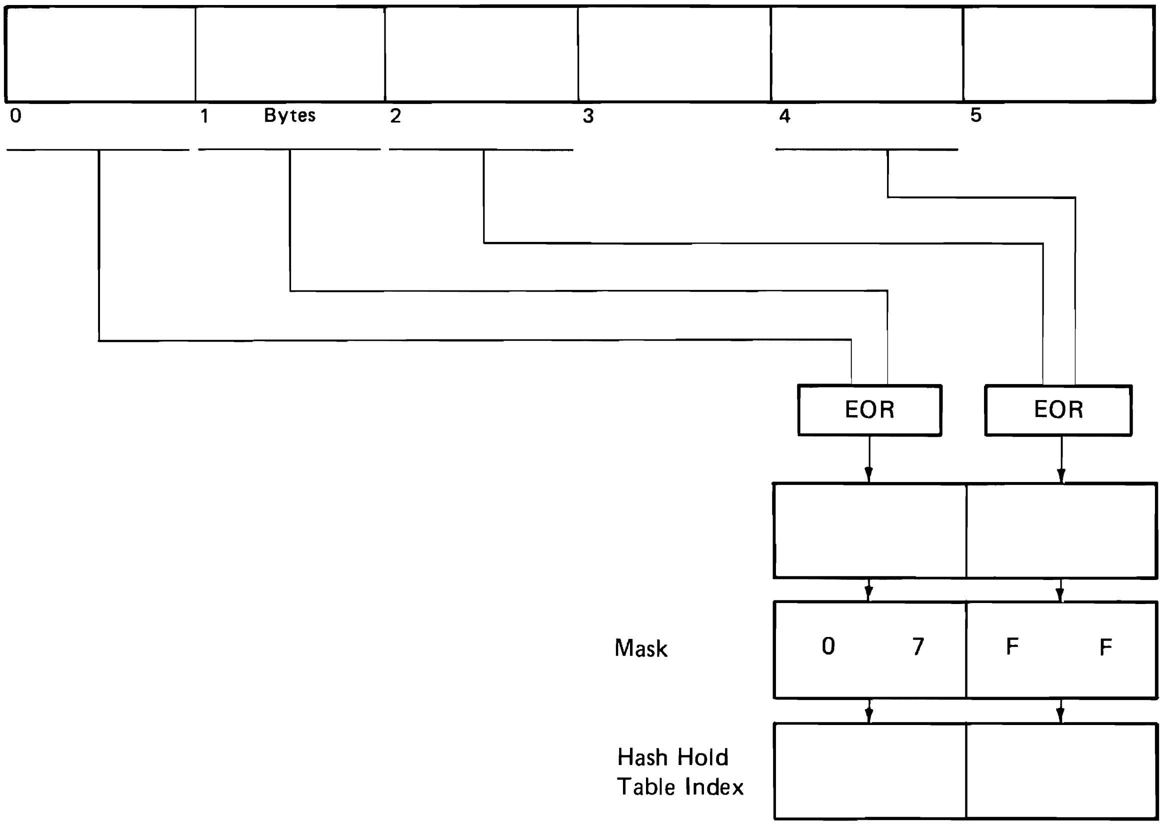


Figure 9-2. Hold/Free Data Fields

Programming Note: The HHT entry of hexadecimal 0000 indicates a null HR chain. Therefore, HRO is not used.

The other two storage areas used by hold and free are the HHT and the HR (hold record) area. The HHT is a 4096-byte storage page which contains 2048 halfword entries. When hold/free activity is performed in an object, the 6-byte address of that object is hashed, forming a 2-byte index as shown below.

Object Address (6-Byte Virtual Address)



This 2-byte index is used as the address of one of the 2048 HHT entries. The halfword contents of the selected entry (if not = 0) are then used as a halfword index which, when multipled by 16 and concatenated on the right of the high-order 28 bits of the address SID (segment identifier) obtained from the AHR (available hold record) chain address located in the control address table, forms the address of the first (most recent) hold record granted for this object and its hash synonyms (other object addresses which hash to the same 2-byte index value).

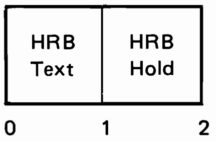
The second storage area, which is composed of 16 consecutive 64 K segments of address space, is a virtual addressing segment called the HR (hold record) area. It contains all the HRs used in all object chains plus the chain of preinitialized HRs available for additional holds. It may also contain some uninitialized area for expansion of the available hold record chain. The HR area segments are identified by the high-order 28 SID bits of the AHR chain address contained in the control address table.

Hold/Free activity involves the use of seven IMP instructions. These instructions are:

• SCB	Set Chain Busy
• RCB	Reset Chain Busy
• GHRF	Grant Hold Record First
• GHR	Grant Hold Record
• FHRF	Free Hold Record First
• FHR	Free Hold Record
• RAHR	Return Available Hold Record

The first six of these instructions have as one operand a base register containing the 6-byte effective address of the object involved. The Grant/Free and SCB instructions have another base register operand to receive the address of a hold record of interest (GHR and FHR instructions have this register preloaded with a HR of interest and update it to a new HR if necessary). The last four instructions have yet another operand which is a storage halfword data field called a hold request block. This hold request block contains (1) the hold types to be checked for and (2) the holds to be granted for hold, or which were granted if freeing as shown in Figure 9-3.

HRB (hold request block) Format



Byte	Description
HRBTEST	Holds to be tested for
HRBHOLD	Holds to be granted or freed

HR (hold record) Format

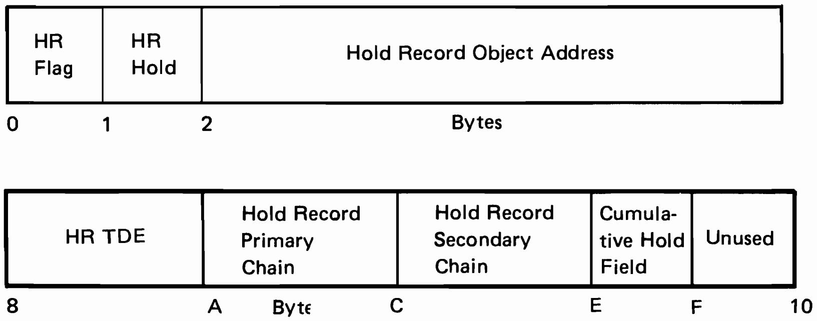


Figure 9-3 (Part 1 of 2). Formats of the Hold Request Block and the Hold Record

Bytes (Hex)	Bits	Description
0		**Hold Record Flag.**
	0	Head of secondary chain:
		0 = 0 Not the head of a secondary chain.
		0 = 1 Head of a secondary chain.
	1	Secondary chain monitored: The monitored hold is on a secondary chain. This bit is set by the VMC exception handler.
		1 = 0 A request was received to free the object.
		1 = 1 No request was received to free the object.
	2-4	Unused.
	5	Hold record busy:
		5 = 0 Not busy.
		5 = 1 Busy.
	6	Hold record end of primary chain:
		6 = 0 Not end of chain.
		6 = 1 End of chain.
	7	Hold record monitored:
		7 = 0 Not monitored. 7 = 1 Monitored.
1		**Hold Record Hold:** Contains the HRBHOLD when a hold is granted, indicating the type of hold.
2-7		**Hold Record Object Address:** Contains the 6-byte address of a hold object.
8-9		**Hold Record TDE:** Contains the TDE identifier when the hold is granted.
A-B		**Hold Record Chain:** Contains a halfword index of the next hold record in the primary chain for hash synonyms. This index is also used as a backward pointer on the secondary chain.
C-D		**Hold Record Secondary Chain:** Contains a halfword index of the next hold record in the chain for this object. A zero indicates the end of chain.
E		**Cumulative Hold Field:** Contains the OR of all the holds on the hold records in the secondary chain (exists only when there is a secondary chain).
F		**Unused.**

Figure 9-3 (Part 2 of 2). Formats of the Hold Request Block and the Hold Record

HARDWARE TAGS

Each word in storage has an associated hardware tag bit. Tag bits are used to differentiate between data and pointers. There is one pointer tag (logical AND of the 4 hardware tag bits) associated with each quadword (4 consecutive words) of storage. If a tag bit is set, a valid pointer is located in the quadword corresponding to that tag bit. If a tag bit is zero, no pointer is located in that quadword.

A quadword in storage that is quadword-aligned (on an address divisible by 16) is considered tagged when all 4 words in the quadword have their hardware tag bits set. The quadword is not tagged when any or all of the hardware bits in the quadword are reset.

There are five IMP instructions that can set the hardware tag bits. These are Call Internal, Store and Set Tags, Move and Set Tags, Insert Tags, and Move Characters and Tags. The Add Space Pointer Offset instructions (AHSPOI, AHSPO, and AFSPO) can be used to modify tagged pointers without setting the corresponding tag bits off. The Load and Verify Tags instruction checks the hardware tags but does not alter them. All other instructions and other facilities that store data in storage cause the associated hardware tag bits of the words stored to be reset. Thus, a tagged quadword cannot be manipulated by an instruction other than the instructions referenced above and still remain tagged.

Storage management must save the tags when it writes a page to auxiliary storage and restore them when it reads the page into main storage. Storage management uses the Extract Tags and Insert Tags instructions to do this.

VMC Service Aids

Direct support for servicing the VMC exists in two facilities: a task switch trace facility and a programmable address compare facility. Each of these is described in the following sections.

TASK SWITCH TRACE FACILITY

General

The task switch trace facility provides a trace record for each IMP task switch in by the processor. Trace records are placed into the trace event buffer in storage as they occur. When that buffer is full, a buffer-full condition is signaled and an alternate buffer is used. Task switch trace activity is controlled by a bit in the TDE (task dispatching element) of the task being switched in.

The trace event buffering operation is controlled by the trace control table (described later in this chapter), which is a control block addressed by an entry in the processor control address table (see Figure 2-2). The processor control address table contains:

- Buffer offsets and thresholds.
- An SRC (send/receive counter) used by the processor to signal the program of a buffer-full condition, and a control bit used to suppress that buffer-full condition signal.
- A damage indicator which the processor notifies the program of a buffer wraparound condition.

The task switch trace record contains:

- The TDE (task dispatching element) identifier.
- A time stamp.
- The binary overflow and Instruction Address Register (control storage address register if an HMC procedure) values, for the first procedure to be executed in the new task only.

TDE Control Bit

The TDE bit that controls TDE tracing is bit 2 of hex byte 0C. If the bit is a one, the trace function is performed when the task is switched in. If the bit is zero, the function is not performed.

Trace Control Table Address

Hex bytes 98-9F of the processor control address table contain the address of the trace control table. Hex bytes 98-99 are reserved, and hex bytes 9A-9F contain the 6-byte address. The address has use code b (see *Permanent Storage Assignments* in Chapter 2) and the table itself must be doubleword aligned, resident in main storage, and must not cross a page boundary; otherwise, a machine check occurs.

TRACE CONTROL TABLE

The trace control table is a 28-byte object which contains the entries that control the logging of trace records and notification of the program when the trace event buffer is full.

Format:

Type	Control	Reserved	Trace Count	Primary Buffer Offset	Primary Buffer Threshold
0	1	2 Bytes	4	6	8

Reserved	Alternate Buffer Threshold	Alternate Buffer Offset	Send/Receive Counter	
A Bytes	C	E	10	1C

Bytes (Hex)	Bits	Description
0		**Type:** Contains the trace control table type (hex E3).
1		**Control:** Control.
	0	0 Buffer wraparound has *not* occurred.
		1 Buffer wraparound has occurred.
	1	0 Disable SENDC on wraparound condition.
		1 Enable SENDC on wraparound condition.
	2-7	Reserved: May be any value.
2-3		**Reserved:** May be any value
4-5		**Trace Count:** The number of 4-byte blocks currently in the primary trace event buffer.
6-7		**Primary Buffer Offset:** Provides the address offset within the trace control table segment to the beginning of the primary trace event buffer.
8-9		**Primary Buffer Threshold:** The number of 4-byte blocks of data that can be put into the primary buffer.
A-B		**Reserved:** May be any value.
C-D		**Alternate Buffer Threshold:** The number of 4-byte blocks of data which can be put into the altenate buffer. When the primary buffer becomes full, alternate buffer threshold is copied to the primary buffer threshold by the processor.
E-F		**Alternate Buffer Offset:** Provides the address offset within the trace control table segment to the beginning of the buffer area which is to be used when the primary buffer exceeds its limits. When this occurs, the alternate buffer offset is copied to the primary buffer offset by the processor.
10-1B		**Send/Receive Counter:** The send/receive counter in the trace control table is used to control the dispatching of the task which handles buffer-full conditions.

Task Switch Trace Record

The task switch trace record is a 16-byte record.that contains the information to be logged each time a task switch occurs. This information includes a time stamp, the TDE (task dispatching element) identifier, and the current instruction stream address.

Format:

Type	Length	TDE Identifier		Time Stamp
0	1	2	Bytes	4

Address Register		Base Register 0	
8	Bytes	A	10

Bytes (Hex)	Bits	Description
0		**Type**: Contains the task switch trace record type (hex F0).
1		**Length**: Contains the trace record length, expressed in terms of number of 4-byte blocks, as well as the type of procedure contained in the TDE CRE (call/return element).
	0	0 IMP procedure.
	1	1 HMC procedure.
	1	Reserved: Must be zero.
	2-5	Length (= binary 0100).
	6-7	Reserved: Must be zeros.
2-3		**TDE Identifier**: Contains the ID field of the TDE.
4-7		**Time Stamp**: Contains bytes 2-5 of the current time of day clock value. If time of day clock is not running, it contains all zeros. By using bytes 2-5 of the clock, approximately 20 hours of unique time stamps are available.
8-9		**Address Register**: If byte 1, bit 0 is zero, this contains the IAR (instruction address register) value of the TDE CRE; otherwise it contains the CSAR (control store address register) value of TDE CRE.
A-F		**Base Register 0**: Contains the base register zero value of the TDE CRE.

Operation

Each time a task switch-in occurs, TDE byte C, bit 2 is tested by the processor. If on, a task switch trace record is generated and added to the trace event buffer in storage, as follows:

1. The trace control table is accessed via bytes hex 98-9F of the control address table and the trace count is multiplied by four to obtain an index to the current entry of the buffer.

2. The result of step 1 is added to the primary stack offset to obtain the offset of the first byte of the new buffer entry. A carry from bit 0 or a nonfullword-aligned result causes a machine check to occur.

3. The offset obtained in step 2 is concatenated with the trace control table SID (segment identifier) and the resulting virtual address is translated. If the address cannot be translated, a machine check occurs.

4. A value of 1 is subtracted from 4 times the length and this new value is added to the result of step 2 to obtain the offset of the rightmost byte of the new buffer entry. A carry from bit 0 of the result causes a machine check to occur.

5. Step 3 is repeated, using the offset value obtained in step 4.

6. The trace record is written to the address obtained in step 4.

7. Length is added to the trace count in storage.

8. The result of step 7 is tested for a value greater than or equal to the primary buffer limit. If not greater than or equal, the buffer operation is complete; otherwise the operation continues.

9. The trace count is loaded with hex 0000.

10. The primary stack offset and alternate buffer offset are compared. If equal, byte 1, bit 0 of the trace control table is set, and the operation continues.

11. The alternate trace offset and alternate trace limit are copied to the primary trace offset and primary trace limit, respectively.

12. Byte 1, bit 1 of the trace control table is tested. If set, a send count operation is performed using bytes hex 10-1B of the trace control table.

Programming Note: The threshold test ensures that a trace record does not begin beyond the threshold. However, it is possible that the end of a record can extend beyond the threshold. Hence, an overflow area should be provided at the end of each buffer. The length of the overflow area should equal the trace record length minus 4 bytes.

ADDRESS COMPARE MODE

The address compare mode allows the program to be signaled whenever one or more of the following events occur:

- An instruction is fetched from a designated virtual storage location.
- The contents of a designated virtual storage location are accessed by either the processor or the I/O channel.
- The contents of a designated virtual storage location are altered by either the processor or the I/O channel.
- The contents of a designated virtual storage location are altered to a predetermined value by either the processor or I/O channel.

An address compare mode is established through the use of the Set Address Compare Mode instruction. When an address compare match occurs, the program is signaled via an address compare exception; when the exception occurs, the address compare mode remains set and the compare address is unchanged. The address compare mode is terminated via the Reset Address Compare Mode instruction.

An address compare mode can be set up to occur for an instruction stream fetch, a processor data access, or an I/O channel data access, selectively. Also, it can be set up to occur if any of the preceding three access types occur. Two other setup options are available with the address compare mode. The first is the capability to specify that an exception is to be recognized only if a store type access to the designated storage location occurs during a processor or I/O channel data access, as opposed to the general capability where either a fetch or store type access causes an exception to be recognized. The second is the capability to compare the value of a prespecified character to the character stored in the designated storage location by the processor or I/O channel; an address compare exception occurs only if the two characters compare.

When an address compare match is detected for an instruction stream fetch, the exception occurs prior to execution of the designated instruction. If the instruction consists of multiple units of operation, an exception occurs prior to execution of each of the units of operation. When an address compare match occurs for processor data accesses, an exception occurs after completion of the unit of operation during which it occurred, where the unit of operation can be either an IMP or HMC unit of operation. If the processor is not executing a unit of operation when the address compare occurs, the exception is recognized after completion of the next unit of operation to be executed. For example, if the processor is in the wait state and an address compare occurs due to servicing of the IMP timers, the exception is not recognized until a task switch occurs. When an address compare match is detected for an I/O channel access, an exception is recognized after completion of the unit of operation currently being executed by the processor. If none is being executed, the exception is recognized as in the case above.

The address compare exception is maskable if it occurs on the instruction stream. If bit 9 of the TDE (task dispatching element) exception mask field is 0, it is set to 1 by the processor, but no exception occurs and the instruction is completed normally. The mask allows the IMP exception handler to leave an address compare set at a particular instruction after the address compare has initially occurred. Without the mask, it would be necessary for the program to remove the address compare in order to avoid an endless loop.

The following is the processing sequence for an instruction address compare, set by the Set Address Compare Mode Instruction:

1. Address compare mode is set for an instruction fetch at storage location L.

2. Bit 9 of the TDE (task dispatching element) exception mask field is set to 1, allowing normal operation of instruction-fetch address compare exceptions.

3. An address compare exception occurs when the instruction is fetched from storage location L. The instruction has not yet executed.

4. The IMP exception handler responds to the exception by setting bit 9 of the TDE exception mask to 0. The instruction at storage location L is retried.

5. No exception occurs and the instruction completes normally. The processor sets bit 9 of the TDE exception mask to 1, enabling the address compare exception to be presented the next time the instruction at storage location L is fetched.

The address compare exception can occur concurrently with other exception types, typically PEM (program event monitor) and certain other program exceptions that are detected after the instruction has accessed an operand in storage. A PEM exception and instruction stream address compare can be detected simultaneously and, if they are, they will be reported in the same CRE (call/return element). Also, many of the other program exception types can be detected simultaneously with a processor or I/O data exception, in which case both will be reported in the same CRE.

The processor address compare facility handles both V=R and V=V addresses. If the compare address is a V=R address, it is converted to a real address format and loaded into the address compare facility. If the compare address is a V=V address, an attempt is made to translate it to a real address. If the translation is successful, the real address is loaded into the address compare facility. If the translation is not successful, the virtual address is buffered in the processor. Then, whenever a new address is loaded into the lookaside buffer or resolved for I/O use, the buffered virtual address and address being resolved are compared. If the segment identifier and page identifier portion of the addresses compare, the buffered virtual address is converted to a real address and loaded into the address compare facility. Conversely, when the Invalidate Primary Directory Entry instruction (or Examine Primary Directory Entry instruction, under certain conditions) is executed, a test is made to determine if the page being invalidated in the primary directory contains the address in the address compare facility; if it does, address compare mode remains set and the buffered virtual address is retained but the address compare facility is purged.

Next Command				
Current Mode	**SACM Operation**	**RACM Operation**	**Set Command from Console**	**Reset Command from Console**
Reset	Set programmed AC mode	Reset AC mode	Set console AC mode	Reset AC mode
Set, from program	Cancel old, set new	Reset AC mode	Cancel old, set new	Reset AC mode
Set, from console	Not set condition code	Not reset condition code	Cancel old, set new	Reset AC mode

Programming Notes:

1. The processor has an address compare facility which is capable of handling a single AC (address compare) at a time. This facility is used by both the programmed and console-set address compare features and when contention occurs, the console-set mode receives priority, as shown in the following table. The instruction length value stored in the CRE (call/return element) when an address compare exception occurs is zero.
2. The processor and I/O address compare exceptions are recognized whenever the fullword containing the designated byte is accessed.
3. The character compare operation which occurs as the result of the store with compare option of the Set Address Compare Mode instruction is performed at the end of the unit of operation during which the storage access was detected. This means that if more than one store to the designated address occurs within a single unit of operation, the compare is made using the last character stored. Also, since the processor detects only fullword accesses, it is possible that the compare may occur when in fact only bytes adjacent to the tested character were modified.
4. Normally, the I/O device which causes an address compare match continues to transfer data. However, there is a System/38 control facility available to the customer engineer which, when set, causes the device to halt its data transfer after the match occurs. Hence, completion of the data transfer cannot be guaranteed under all conditions.
5. If an address compare match is detected when the task dispatcher is blocked, the exception is not recognized, the match is reset (bit 9 of the TDE [task dispatching element] exception mask is set), and processing continues; the match is ignored.
6. Performance is reduced when an address compare mode is set and the address compare facility is loaded.
7. For all programmable address compare exceptions, the instruction length is set to zero in the CRE (call return element).

FUNCTION CALL LINKAGE

Function call support is provided to enable the direct calling of one VMC function by another, and to provide status retention of the calling function without the use of the SVL (supervisor linkage) facility. The function call support provides a means for indexing into a FRAT (function routine address table) to obtain routing information for the called function. It also provides a mechanism by which the status of the calling function can be saved, through the use of the IMP stack support. The FNC2 (Function Call Double) instruction assumes that base register 3 points to the next available stack entry. Figure 9-4 represents an overview of the function call flow.

Function Routine Address Table

The FRAT consists of 256 10-byte entries, and is located in virtual storage. The 6-byte address of the FRAT is maintained in the control address table entry which starts at byte A0. The format of each entry is as follows:

Entry Address	B (Ø)	Not Used

0 2 Bytes 8 A

Bytes 0-1 Entry address of the first instruction to be executed in the function

Bytes 2-7 Instruction base register value for the function being called

Bytes 8-9 Not used

Note: If the function routine address table is not halfword aligned when accessed by the processor, a specification exception is recognized and the operation is suppressed.

Function Call Stack Usage

The function call facility uses a stack entry to save the status of the calling function as follows:

Stack Entry (Hex Byte)	Usage
0-1	Forward stack pointer
2-17	Not used
18-23	Base registers 1-2 save area
24-77	Not used
78-79	Instruction address register
7A-7F	Base register 0 save area

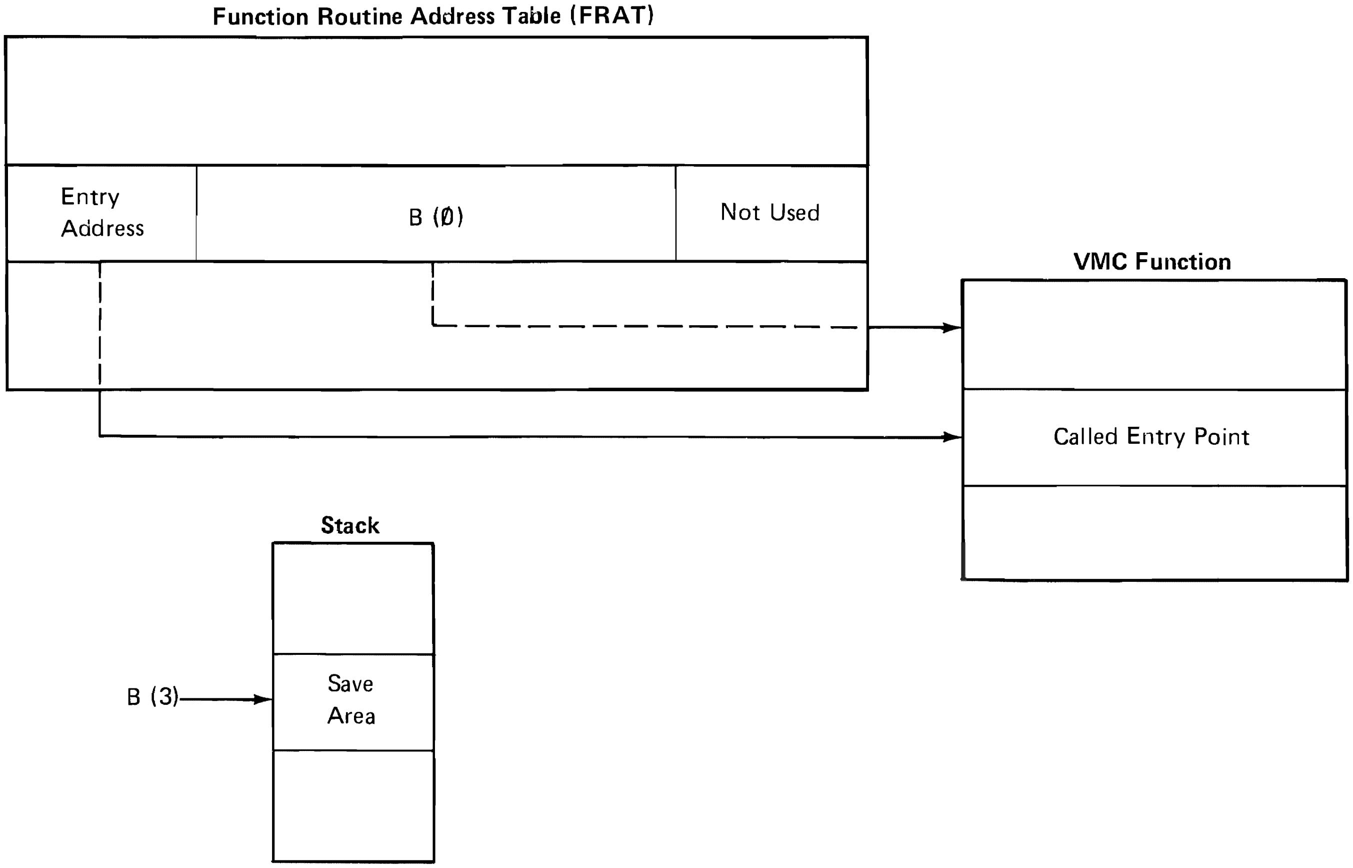


Figure 9-4. Function Call Flow

Space Pointer Support

A space pointer is a System/38 object which provides addressability to a specific byte in the data area associated with that object. The following instructions assist the VMC in the processing and validation of space pointers:

- Add Fullword Space Pointer Offset
- Add Halfword Space Pointer Offset
- Add Halfword Space Pointer Offset Immediate
- Compute Address Long
- Compute Address Long Halfword
- Compare Logical Address Register
- Load Space Offset Pointer
- Store Space Offset Pointer

The following discussion defines the space pointer and segment group header fields referenced or manipulated by the above listed instructions.

Space Pointer Fields

The format of a space pointer is as follows:

Type	Unused

0 1 Bytes

SID Extender	Segment Group Identifier	Segment Group Offset

8 A D 10

Byte	Bits	Description
0		**Type.**
	0-1	00 System pointer.
		01 Instruction pointer.
		10 Space pointer.
		11 Data pointer.
	2	0 The pointer is rèsolved (contains a valid address).
		1 The pointer is not resolved.
	3-7	Reserved: Must be zeros.
1-7		**Not used.**
8-9		**SID Extender:** These bytes are specified as a 2-byte logical extension to the segment group identifier and are used and assigned by VMC storage management.
A-C		**Segment Group Identifier:** Used to identify a 16 megabyte address space. The 3-byte segment group identifier is the high-order 3 bytes of an IMP 6-byte virtual address.
D-F		**Segment Group Offset:** Used to address a byte within a 16-megabyte segment group. The 3-byte segment group offset is the low-order 3 bytes of a 6-byte virtual address. The segment group offset is always greater than or equal to the space locator offset found in the segment group header identified by the segment group identifier field.

Segment Group Header

The first 32 bytes of the 16-megabyte segment group allocated by VMC form the segment group header. The segment group header fields that can be referenced implicitly via the IMP instruction set are formatted as follows:

Unused	SID Extender	Unused	Space Locator Offset

0 Bytes 4 6 1D 20

Byte **Description**

0-3 **Not used.**

4-5 **SID Extender**: Used as a 2-byte logical extension of the segment group identifier (bytes A-C of the space pointer), used and assigned by VMC storage management.

6-1C **Not used.**

1D-1F **Space Locator Offset**: These bytes specify a 3-byte offset into the segment group and identify the lowest available byte in the segment group.

Note: The segment group offset is greater than or equal to the space locator offset found in the segment group header identified by the segment group identifier field.

Chapter 10. Instruction Descriptions

The instructions are described in alphabetical order (by instruction name) with an example adjacent to each instruction. Appendix C is an alphabetical list of the instructions by mnemonic; Appendix B is a chart of operation code assignments showing the mnemonics and operation codes.

Refer to Chapter 2 for more detailed information about instruction formats and registers.

Some VMC instructions are treated as implicit SVLs (supervisor linkage). (These instructions are identified in Appendix B.) Whenever an attempt is made to execute one of these instructions, the processor causes an implicit SVL operation to be performed. The operation code of the instruction is used as the index into the SVL table. The SVL routine located through the SVL table performs the instruction execution. For a detailed description of the SVL operation, see Chapter 6.

Notes:

1. The result of an instruction is placed in the first operand unless stated otherwise within the description of the instruction.
2. The L, L_1, and L_2 fields in the instructions specify a value that is one less than the actual number of bytes for each operand.

Data not critical to the execution of an instruction is indicated in the instruction format diagram as a blank field; the same field is represented in the example format diagram with a placeholder value of 0. Nonessential data is indicated in the storage example with one lower case x per half-byte. Other data used in the examples is assumed for the purpose of explanation.

The examples will be better understood by looking at them while reading the instruction description and operation. Sequence numbers (for example, **1**) have been used in some of the more complicated instructions.

ADD CHARACTERS (AC)

Instruction Description

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

Format: SS

C0	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The addition is performed with both operands treated as signed binary quantities. If the operands are unequal in length, the shorter operand is considered to be extended to the left with bits equal to the sign bit.

Overflow: If the carry from the sign-bit position and the carry from the leftmost numeric bit position agree, no overflow occurs; if they disagree, an overflow occurs. If the first operand is too short to contain all significant bits of the result, an overflow occurs and significant bits are lost.

Sign Code: The sign bit of the sum is not changed after the overflow. The sign of the sum is unpredictable when significant bits are lost.

Condition Code: If significant bits are lost the condition code indicates the sign the sum would have if an overflow had not occurred.

0	Sum	=	0
1	Sum	<	0
2	Sum	>	0
3	--		

Carry: See *Overflow.*

Boundary Requirements: The operands can overlap in storage if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Effective address overflow

AC Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
C0	5	4	3	040	3	152

0 Bits 8 12 16 20 32 36 47

Assembler: AC $D_1(L_1, B_1)$, $D_2(L_2, B_2)$

Machine: C054 3040 3152

B_1(3) and B_2(3): 0001 4120 0000

Storage — Before

	0/8	2/A	4/C	6/E
0001 4120 0040	0000	A542	BC24	
0001 4120 0152		2901	1132	A6

Storage — After

	0/8	2/A	4/C	6/E
0001 4120 0040	0029	A653	EECA	
0001 4120 0152		2901	1132	A6

	Before	**After**
Condition Code:	x	2

This page is intentionally left blank.

ADD FULLWORD SPACE POINTER OFFSET (AFSPO)

Instruction Description

The space pointer specified by the second operand is verified as a tagged pointer; the third operand is used as a signed displacement which is added to the 3-byte offset portion of the second-operand space pointer. The 6-byte address that results is loaded onto the first-operand register and the second operand space pointer, leaving the pointer tagged.

Format: SS

BE	B_1	2	B_2	D_2	B_3	D_3

0 Bits 8 12 16 20 32 36 47

Operation: The following validity checks are performed on the second operand:

- The second operand must be tagged.
- Bits 0-2 must be binary 100 (a space pointer).
- Bytes 8 and 9 of the second operand must match the halfword in storage (segment group header SID extender field) located at the address determined by concatenating hex 00 0004 to the right of bytes 10, 11, and 12 of the second operand.

If any of these validity checks fails, a verify exception is recognized, and the operation is suppressed.

The address computation is carried out as follows: the segment group offset portion of the space pointer (bytes 13, 14, and 15) is logically padded on the left with a byte of zeros, creating a positive 4-byte signed binary integer. This value is then added to the displacement identified by the third operand (a 32-bit signed integer). The result of this calculation must satisfy the following validity checks:

- It must be a positive result.
- It must not be less than the value of the space locator offset. The space locator offset is a 3-byte logical binary field in storage located at the address determined by concatenating a hex 00 001D to the right of the 3-byte segment group identifier specified in bytes 10, 11, and 12 of the second operand.
- The 4-byte sum must be less than hex 00FF FFFF.

If any of these validity checks fails, an invalid segment group address exception is recognized, and the operation is suppressed. Otherwise, the rightmost 3 bytes of the calculated result are concatenated to the right of bytes 10, 11, and 12 of the second operand to form the resultant 6-byte address. This resultant address is placed into the first operand and into the address field of the space pointer (bytes 10-15 of the second operand). No storage reference is made to check for addressing exceptions, using the resultant address. The space pointer remains tagged.

Overflow: See *Operation.*

Sign Code: See *Operation.*

Condition Code: Not changed.

Boundary Requirements: The second operand is a quadword and must begin on a quadword boundary; otherwise, a specification is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification
- Invalid segment group address
- Verify

AFSPO Example

Op	B_1	E	B_2	D_2	B_3	D_3
BE	8	2	2	020	4	104

0 Bits 8 12 16 20 32 36 47

Assembler: AFSPO B_1,D_2(B_2),D_3(B_3)

Machine: BE82 2020 4104

	Before	After
B_1(8):	xxxx xxxx xxxx	00A5 2000 1320
B_2(2):	00C1 B000 4BC0	00C1 B000 4BC0
B_3(4):	00C1 B000 BC24	00C1 B000 BC24

Storage — Before

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	8000	0000	0000	0000
	0005	00A5	2000	0B20
00C1 B000 BD28	0000	0800		

Storage — After

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	8000	0000	0000	0000
	0005	00A5	2000	1320
00C1 B000 BD28	0000	0800		

ADD HALFWORD (AH)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RS

80	R_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: Both operands are treated as signed binary quantities.

Overflow: If the carry from the sign-bit position and the carry from the leftmost numeric bit position agree, no overflow occurs; if they disagree, an overflow occurs.

Sign Code: Not changed after the overflow.

Condition Code: If significant bits are lost, the condition code indicates the sign the sum would have if an overflow had not occurred. A sum and a negative result that overflows yields a positive sign.

0	Sum	=	0
1	Sum	<	0
2	Sum	>	0
3	--		

Carry: See *Overflow.*

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Effective address overflow
- Specification

AH Example

Op	R_1	E	B_2	D_2
80	0	0	2	120
0 Bits	8	12	16	20 31

Assembler: AH R_1, $D_2(B_2)$

Machine: 8000 2120

B_2(2): 0023 5430 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
0023 5430 0120	FFFE			

	Before	After
R_1(0):	0019	0017
Condition Code:	x	2

ADD HALFWORD IMMEDIATE (AHI)

Instruction Description

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

Format: SI

A0		0	B_1	D_1	I_2
0 Bits	8	12	16	20	32 47

Operation: Both operands are treated as signed binary quantities.

Overflow: If the carry from the sign-bit position and the carry from the leftmost numeric bit position agree, no overflow occurs; if they disagree, an overflow occurs.

Sign Code: Not changed after an overflow.

Condition Code: If an overflow occurs, the condition code indicates the sign that the sum would have if an overflow had not occurred. A sum and a negative result that overflows yields a positive sign.

0	Sum	=	0
1	Sum	<	0
2	Sum	>	0
3	--		

Carry: See *Overflow.*

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Effective address overflow
- Specification

AHI Example

Op A0		**E** 0	**B_1** 3	**D_1** 130	**I_2** 0234
0 Bits	8	12	16	20	32 47

Assembler: AHI $D_1(B_1)$, I_2

Machine: A000 3130 0234

$B_1(3)$: 0001 0036 0000

Storage — Before

	0/8	2/A	4/C	6/E
0001 0036 0130	0123			

Storage — After

	0/8	2/A	4/C	6/E
0001 0036 0130	0357			

	Before	After
Condition Code:	x	2

ADD HALFWORD REGISTER (AHR)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RR

20	R_1	R_2

0 Bits 8 12 15

Operation: Both operands are treated as signed binary quantities.

Overflow: If the carry from the sign-bit position and the carry from the leftmost numeric bit position agree, no overflow occurs; if they disagree, an overflow occurs.

Sign Code: Not changed after an overflow.

Condition Code: If an overflow occurs, the condition code indicates the sign that the sum would have if an overflow had not occurred. A sum and a negative result that overflows yields a positive sign.

0	Sum	=	0
1	Sum	<	0
2	Sum	>	0
3	--		

Carry: See *Overflow*.

Boundary Requirements: None.

Program Exception: Binary overflow.

AHR Example

Op	R_1	R_2
20	5	6

0 Bits 8 12 15

Assembler: AHR R_1, R_2

Machine: 2056

	Before	After
R_1(5):	0021	001E
R_2(6):	FFFD	FFFD
Condition Code:	x	2

ADD HALFWORD REGISTER IMMEDIATE (AHRI)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RI

50	R_1		I_2
0 Bits	8	12	16 31

Operation: Both operands are treated as signed binary quantities.

Overflow: If the carry from the sign-bit position and the carry from the leftmost bit position agree, no overflow occurs; if they disagree, an overflow occurs.

Sign Code: Not changed after an overflow.

Condition Code: If an overflow occurs, the condition code indicates the sign the sum would have if an overflow had not occurred. A sum and a negative result that overflows yields a positive sign.

0	Sum	=	0
1	Sum	<	0
2	Sum	>	0
3	--		

Carry: See *Overflow.*

Boundary Requirements: None.

Program Exception: Binary overflow.

AHRI Example

Op 50	R_1 4		I_2 0234
0 Bits	8	12	16 31

Assembler: AHRI R_1, I_2

Machine: 5040 0234

	Before	After
R_1 (4):	0012	0246
Condition Code:	x	2

ADD HALFWORD SPACE POINTER OFFSET (AHSPO)

Instruction Description

The space pointer specified by the second operand is verified as a tagged pointer; the third operand is used as a signed displacement which is added to the 3-byte offset portion of the second-operand space pointer. The 6-byte address that results is loaded into the first-operand register and the second-operand space pointer, leaving the pointer tagged.

Format: SS

BE	B_1	1	B_2	D_2	B_3	D_3

0 Bits 8 12 16 20 32 36 47

Operation: The following validity checks are performed on the second operand:

- The second operand must be tagged.
- Bits 0-2 must be binary 100 (a space pointer).
- Bytes 8 and 9 of the second operand must match the halfword in storage (segment group header SID extender field) located at the address determined by concatenating hex 00 0004 to the right of bytes 10, 11, and 12 of the second operand.

If any of these validity checks fails, a verify exception is recognized, and the operation is suppressed.

The address computation is carried out as follows: the segment group offset portion of the space pointer (bytes 13, 14, and 15) is logically padded on the left with a byte of zeros, creating a positive 4-byte signed binary integer. This value is then added to the displacement identified by the third operand (a 16-bit signed integer). The address arithmetic is performed by propagating the sign bit through the third and fourth (left) offset bytes and performing the 4-byte signed binary addition. The result of this calculation must satisfy the following validity checks:

- It must be a positive result.
- It must not be less than the value of the space locator offset. The space locator offset is a 3-byte logical binary field in storage located at the address determined by concatenating a hex 00 001D to the right of the 3-byte segment group identifier specified in bytes 10, 11, and 12 of the second operand.
- The 4-byte sum must be less than hex 00FF FFFF.

If any of these validity checks fails, an invalid segment group address exception is recognized, and the operation is suppressed. Otherwise, the rightmost 3 bytes of the calculated result are concatenated to the right of bytes 10, 11, and 12 of the second operand to form the resultant 6-byte address. This resultant address is placed into the first operand and into the address field of the space pointer (bytes 10-15 of the second operand). No storage reference is made to check for addressing exceptions, using the resulant address. The space pointer remains tagged.

Overflow: See *Operation.*

Sign Code: See *Operation.*

Condition Code: Not changed.

Boundary Requirements: The second operand is a quadword and must begin on a quadword boundary; otherwise, a specification is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group address
- Specification
- Verify

AHSPO Example

Op	B_1	E	B_2	D_2	B_3	D_3
BE	8	1	2	020	4	104

0 Bits 8 12 16 20 32 36 47

Assembler: AHSPO B_1,D_2(B_2),D_3(B_3)

Machine: BE81 2020 4104

	Before	After
B_1(8):	xxxx xxxx xxxx	00A5 2000 1320
B_2(2):	00C1 B000 4BC0	00C1 B000 4BC0
B_3(4):	00C1 B000 BC24	00C1 B000 BC24

Storage — Before

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	8000	0000	0000	0000
	0005	00A5	2000	0B20
00C1 B000 BD28	0800			

Storage — After

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	8000	0000	0000	0000
	0005	00A5	2000	1320
00C1 B000 BD28	0800			

ADD HALFWORD SPACE POINTER OFFSET IMMEDIATE (AHSPOI)

Instruction Description

The space pointer specified by the second operand is verified as a tagged pointer; the third operand is used as a signed displacement which is added to the 3-byte offset portion of the second-operand space pointer. The 6-byte address that results is loaded into the first-operand register and the second-operand space pointer, leaving the pointer tagged.

Format: SI

BE	B_1	0	B_2	D_2	I_3

0 Bits 8 12 16 20 32 47

Operation: The following validity checks are performed on the second operand:

- The second operand must be tagged.

- Bits 0-2 must be binary 100 (a space pointer).

- Bytes 8 and 9 of the second operand must match the halfword in storage (segment group header SID extender field) located at the address determined by concatenating hex 00 0004 to the right of bytes 10, 11, and 12 of the second operand.

If any of these validity checks fails, a verify exception is recognized and the operation is suppressed.

The address computation is carried out as follows: the segment group offset portion of the space pointer (bytes 13, 14, and 15) is logically padded on the left with a byte of zeros, creating a positive 4-byte signed binary integer. This value is then added to the displacement identified by the third operand (a 16-bit signed integer). The address arithmetic is performed by propagating the sign bit through the third and fourth (left) offset bytes and performing the 4-byte signed binary addition. The result of this calculation must satisfy the following validity checks:

- It must be a positive result.

- It must not by less than the value of the space locator offset. The space locator offset is a 3-byte logical binary field in storage located at the address determined by concatenating a hex 00 001D to the right of the 3-byte segment group identifier specified in bytes 10, 11, and 12 of the second operand.

- The 4-byte sum must be less than hex 00FF FFFF.

If any of these validity checks fail, an invalid segment group address exception is recognized, and the operation is suppressed. Otherwise, the rightmost 3 bytes of the calculated result are concatenated to the right of bytes 10, 12, and 12 of the second operand to form the resultant 6-byte address. This resultant address is placed into the first operand and into the address field of the space pointer (bytes 10-15 of the second operand). No storage reference is made to check for addressing exceptions, using the resultant address. The space pointer remains tagged.

Overflow: See *Operation.*

Sign Code: See *Operation.*

Condition Code: Not changed.

Boundary Requirements: The second operand is a quadword and must begin on a quadword boundary; otherwise, a specification is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group address
- Specification
- Verify

AHSPOI Example

Op	B_1	E	B_2	D_2	I_3
BE	8	0	2	020	0800

0 Bits 8 12 16 20 32 47

Assembler: AHSPOI B_1,D_2(B_2),I_3

Machine: BE80 2020 0800

	Before	After
B_1(8):	xxxx xxxx xxxx	00A5 2000 1320
B_2(2):	00C1 B000 4BC0	00C1 B000 4BC0

Storage — Before

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	8000	0000	0000	0000
	0005	00A5	2000	0B20

Storage — After

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	8000	0000	0000	0000
	0005	00A5	2000	1320

ADD LOGICAL BYTE (ALB)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RS

71	r_1	1	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The addition is performed with both operands treated as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

ALB Example

Op	r_1	E	B_2	D_2
71	8	1	3	00A

0 Bits 8 12 16 20 31

Assembler: ALB r_1, $D_2(B_2)$

Machine: 7181 300A

B_2(3): 0012 0001 1000

Storage — Before and After

	0/8	2/A	4/C	6/E
0012 0001 100A		95		

	Before	After
r_1(8):	24	B9
Condition Code:	x	1

ADD LOGICAL BYTE REGISTER (ALBR)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RR

10	r_1	r_2

0 Bits 8 12 15

Operation: The addition is performed with both operands treated as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements and Program Exceptions: None.

ALBR Example

Op 10	r_1 2	r_2 3

0 Bits 8 12 15

Assembler: ALBR r_1, r_2

Machine: 1023

	Before	**After**
r_1(2):	2A	C6
r_2(3):	9C	9C
Condition Code:	x	1

ADD LOGICAL BYTE REGISTER IMMEDIATE (ALBRI)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RI

43	r_1	0	I_2	

0 Bits 8 12 16 24 31

Operation: The addition is performed with both operands treated as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements and Program Exceptions: None.

ALBRI Example

Op	r_1	E	I_2	
43	A	0	12	

0 Bits 8 12 16 24 31

Assembler: ALBRI r_1, I_2

Machine: 43A0 1200

	Before	After
r_1(A):	CC	DE
Condition Code:	x	1

ADD LOGICAL CHARACTER (ALC)

Instruction Description

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

Format: SS

C3	L	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Operation: The addition treats both operands as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

ALC Example

Op C3	L 03	B_1 3	D_1 108	B_2 3	D_2 166

0 Bits 8 16 20 32 36 47

Assembler: ALC $D_1(L, B_1), D_2(B_2)$

Machine: C303 3108 3166

$B_1(3)$: 0010 2250 5000

Storage — Before

	0/8	2/A	4/C	6/E
0010 2250 5108	7683	A591		
0010 2250 5166				3729
	5895			

Storage — After

	0/8	2/A	4/C	6/E
0010 2250 5108	ADAC	FE26		
0010 2250 5166				3729
	5895			

	Before	After
Condition Code:	x	1

ADD LOGICAL HALFWORD (ALH)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RS

90	R_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The addition treats both operands as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

ALH Example

Op	R_1	E	B_2	D_2
90	4	0	6	160
0 Bits	8	12	16	20 31

Assembler: ALH R_1, D_2(B_2)

Machine: 9040 6160

B_2(6): 0101 1130 2000

Storage — Before and After

	0/8	2/A	4/C	6/E
0101 1130 2160	1850			

	Before	After
R_1(4):	1150	29A0
Condition Code:	x	1

This page is intentionally left blank.

ADD LOGICAL HALFWORD AND BRANCH ON LIMIT (ALHBL)

Instruction Description

The increment of the second operand is added to the halfword register specified by the first operand and the result is stored in the halfword register specified by the first operand. The sum is then compared to the comparand of the second operand. If the mask specified by the third operand has a corresponding mask bit of 1, the IAR (instruction address register) is replaced by the sum of the branch displacement of the second operand and the offset of the instruction stream base address contained in base register 0; otherwise, instruction sequencing proceeds with the updated IAR.

Format: RS

9F	R_1	M_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The second operand occupies 6 bytes of storage.

Bytes	Contain
1 and 2	Increment value
3 and 4	Comparand
5 and 6	Branch displacement

The increment is added to the first operand and the sum is compared logically with the comparand. Subsequently, the sum is placed in the first-operand location, regardless of whether the branch is taken.

The mask field is used as a 4-bit mask generated by the compare. The 4 bits of the mask correspond, left to right, with the following comparison result:

Bit	Result
0	Sum = Comparand
1	Sum < Comparand
2	Sum > Comparand
3	--

Whenever the comparison result has a corresponding mask bit of one, the updated instruction address is replaced by the sum of the branch displacement and the offset portion of the instruction stream base address contained in base register 0. If the comparison result does not have a corresponding mask bit of one, instruction sequencing proceeds with the updated instruction address.

Logical addition is performed by adding all 16 bits of the first operand and the increment.

The 16-bit comparison is also performed with the quantities treated as unsigned binary values.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: If a carry from the high-order bit position occurs during the addition, it is ignored and does not affect the comparison.

Boundary Requirements: The second operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

ALHBL Example

Op	R_1	M	B_2	D_2
9F	4	4	3	AB0
0 Bits	8	12	16	20 — 31

Assembler: ALHBL $R_1, M_3, D_2(B_2)$

Machine: 9F44 3AB0

Before and After

B(0): 0250 AC2C 2E00

B_2(3): 00EF 021E 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
00EF 021E 0AB0	000A	002E	003C	

	Before	Updated	After
R_1(4):	0022	–	002C
IAR:	3A50	3A54	2E3C

ADD LOGICAL HALFWORD IMMEDIATE (ALHI)

Instruction Description

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

Format: SI

BO		0	B_1	D_1	I_2

0 Bits 8 12 16 20 32 47

Operation: The addition is performed with both operands treated as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

ALHI Example

Op		E	B_1	D_1	I_2
BO		0	3	170	0005

0 Bits 8 12 16 20 32 47

Assembler: ALHI $D_1(B_1)$, I_2

Machine: B000 3170 0005

$B_1(3)$: 0150 1442 6000

Storage — Before

	0/8	2/A	4/C	6/E
0150 1442 6170	D136			

Storage — After

	0/8	2/A	4/C	6/E
0150 1442 6170	D13B			

	Before	After
Condition Code:	x	1

ADD LOGICAL HALFWORD REGISTER (ALHR)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RR

30	R_1	R_2

0 Bits 8 12 15

Operation: The addition is performed with both operands treated as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements and Program Exceptions: None.

ALHR Example

Op 30	R_1 5	R_2 6

0 Bits 8 12 15

Assembler: ALHR R_1, R_2

Machine: 3056

	Before	After
R_1(5):	ABCD	EEEE
R_2(6):	4321	4321
Condition Code:	x	1

ADD LOGICAL HALFWORD REGISTER IMMEDIATE (ALHRI)

Instruction Description

The second operand is added to the first operand and the sum is placed in the first-operand register.

Format: RI

60	R_1	0	I_2

0 Bits 8 12 16 31

Operation: The addition is performed with both operands treated as unsigned binary quantities.

Overflow and Sign Code: Not applicable.

Condition Code: A carry from the leftmost bit position is recorded in the condition code.

0	Sum	=	0, no carry
1	Sum	≠	0, no carry
2	Sum	=	0, carry
3	Sum	≠	0, carry

Carry: See *Condition Code.*

Boundary Requirements and Program Exceptions: None.

ALHRI Example

Op	R_1	E	I_2
60	2	0	2002

0 Bits 8 12 16 31

Assembler: ALHRI R_1, I_2

Machine: 6020 2002

	Before	After
R_1(2):	8001	A003
Condition Code:	x	1

ADD LONG FLOAT (ALF)

Instruction Description

The second operand is added to the first operand (two-operand format) or the third operand is added to the second operand (three-operand format), and the sum is placed in the first operand location. Interchanging of the two source operands in floating-point addition does not affect the value of the sum, but can affect which operand is overwritten.

Format: SS

CE	B_3	1	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: A two-operand or three-operand format may be specified. A two-operand format is used if base register 0 is specified for the third operand. A three-operand format is used if one of the base registers hex 1 through hex F is specified for the third operand.

The exponents of the two operands are compared. The significand of the smaller exponent is shifted right as its exponent is increased until the exponents are the same. The significands are then added algebraically to form an intermediate sum.

The significand of the intermediate sum is rounded, if necessary, according to the rounding mode specified in the task dispatching element.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the sum. The source operands are checked for this value in order of their specification. If two masked not-a-numbers are encountered, the masked not-a-number with the larger fraction value is used as the sum.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the sum is determined by the rules of algebra. If the sum of two operands that have opposite signs is exactly 0, the sign is made plus for all rounding modes except round toward negative infinity, where the sign is made minus.

Condition Code: The result is compared to 0. Values of 0 compare equal even if they differ in sign. Not-a-number values and infinite values compare unordered with everything else.

0 Sum = 0
1 Sum < = 0
2 Sum > = 0
3 Sum is unordered

Carry: If a carry occurs, the sum is shifted right one binary digit position with a high-order 1 bit inserted, and the exponent is increased by 1.

Boundary Requirements: All operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Operands may overlap only if they are coincidental; otherwise, the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

Programming Note: The following chart shows the condition of the sum for various operands.

Sum	First Source (Addend)	Second Source (Addend)
+0	+0	+0
-0	-0	-0
+0	-Real number ≠ 0	+Real number ≠ 0
+0	+Real number ≠ 0	-Real number ≠ 0
+Real number ≠ 0	+Real number ≠ 0	+0 or -0
+Real number ≠ 0	+0 or -0	+Real number ≠ 0
-Real number ≠ 0	-Real number ≠ 0	+0 or -0
-Real number ≠ 0	+0 or -0	-Real number ≠ 0
Masked not-a-number	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-number	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operation	Unmasked not-a-number	Any
Invalid operation	Any	Unmasked not-a-number
+Infinity	+Real number ≠ 0 or -real number ≠ 0	+Infinity
+Infinity	+Infinity	+Real number ≠ 0 or -real number ≠ 0
-Infinity	+Real number ≠ 0 or -real number ≠ 0	-Infinity
-Infinity	-Infinity	+Real number ≠ 0 or -real number ≠ 0
Invalid operation	+Infinity or -infinity	+Infinity or -infinity
+0	+0	-0 Note 1
+0	-0	+0 Note 1
-0	+0	-0 Note 2
-0	-0	+0 Note 2

Notes:

1. Value is not rounded toward negative infinity.
2. Value is rounded toward negative infinity.

ALF Example

Op	B_3	E	B_1	D_1	B_2	D_2
CE	3	1	4	050	4	060
0 Bits	8	12	16 20		32	36 47

Assembler: ALF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: CE31 4050 4060

B_3(3): 0010 0200 0070

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	4880	0010	3000	2400
0010 0200 0070	4807	600A	BC00	9B00

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4880	1761	3ABC	249B
0010 0200 0060	4880	0001	3000	2400
0010 0200 0070	4807	600A	BC00	9B00

	Before	**After**
Condition Code:	x	2

ADD PACKED (AP)

Instruction Description

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

Format: SS

FO	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Addition is algebraic, taking into account the signs and all digits of both operands. All digit codes are checked for validity. Improper codes cause a data exception to be recognized, and the operation is terminated. If necessary, zeros are supplied for the leftmost bytes of either operand.

Overflow: Two possible causes: The first is the loss of a carry from the leftmost digit position of the result field. The second is an oversized result, which occurs when the first-operand field is too short to contain all significant digits of the sum, and significant result digits are lost.

Sign Code: The sign of the sum is determined by the rules of algebra. When the operation is completed without an overflow, a zero sum has a positive sign, but when significant result digits are lost because of an overflow, a zero sum may be either positive or negative, as determined by what the sign of the correct sum would have been.

The processor uses the preferred signs for the sum as follows: positive sign is encoded as 1111 (hex F); a negative sign is encoded as 1101 (hex D). All sign codes are checked for validity. Improper codes cause a data exception and the operation is terminated.

Condition Code: If an overflow occurs, the condition code indicates the sign the sum would have if an overflow had not occurred.

0	Sum	=	0
1	Sum	<	0
2	Sum	>	0
3	--		

Carry: See *Overflow.*

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Effective address overflow

AP Example

Op FO	L_1 3	L_2 2	B_1 4	D_1 210	B_2 4	D_2 261

0 Bits 8 12 16 20 32 36 47

Assembler: AP $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: F032 4210 4261

$B_1(4)$ and $B_2(4)$: 2793 4766 2000

Storage — Before

	0/8	2/A	4/C	6/E
2793 4766 2210	5718	942D		
2793 4766 2261	24	270F		

Storage — After

	0/8	2/A	4/C	6/E
2793 4766 2210	5694	672D		
2793 4766 2261	24	270F		

	Before	After
Condition Code:	x	1

ADD SHORT FLOAT (ASF)

Instruction Description

The second operand is added to the first operand (two-operand format) or the third operand is added to the second operand (three-operand format), and the sum is placed in the first operand location. Interchanging of the two source operands in floating-point addition does not affect the value of the sum, but can affect which operand is overwritten.

Format: SS

AE	B_3	1	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: A two-operand or three-operand format may be specified. A two-operand format is used, if base register 0 is specified for the third operand. A three-operand format is used, if one of the base registers hex 1 through hex F is specified for the third operand.

The exponents of the two operands are compared. The significand of the smaller exponent is shifted right as its exponent is increased until the exponents are the same. The significands are then added algebraically to form an intermediate sum.

The significand of the intermediate sum is rounded, if necessary, according to the rounding mode specified in the task dispatching element.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the sum. The source operands are checked for this value in order of their specification. The masked not-a-number with the larger fraction value is used as the sum.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the sum is determined by the rules of algebra. If the sum of two operands that have opposite signs is 0, the sign is made plus for all rounding modes except round toward negative infinity, where the sign is made minus.

Condition Code: The result is compared to 0. Values of 0 compare equal even if they differ in sign. Not-a-number values and infinite values compare unordered.

0	Sum = 0
1	Sum < 0
2	Sum > 0
3	Sum is unordered

Carry: If a carry occurs, the sum is shifted right one binary digit position with a high-order 1 bit inserted, and the exponent increased by 1.

Boundary Requirements: All operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Operands may overlap if they are coincidental; otherwise, the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

Programming Note: The following chart shows the condition of the sum for various operands.

Sum	**First Source (Addend)**	**Second Source (Addend)**
+0	+0	+0
-0	-0	-0
+0	-Real number ≠ 0	+Real number ≠ 0
+0	+Real number ≠ 0	-Real number ≠ 0
+Real number ≠ 0	+Real number ≠ 0	+0 or -0
+Real number ≠ 0	+0 or -0	+Real number ≠ 0
-Real number ≠ 0	-Real number ≠ 0	+0 or -0
-Real number ≠ 0	+0 or -0	-Real number ≠ 0
Masked not-a-number	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-number	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operation	Unmasked not-a-number	Any
Invalid operation	Any	Unmasked not-a-number
+Infinity	+Real number ≠ 0 or -real number ≠ 0	+Infinity
+Infinity	+Infinity	+Real number ≠ 0 or -real number ≠ 0
-Infinity	+Real number ≠ 0 or -real number ≠ 0	-Infinity
-Infinity	-Infinity	+Real number ≠ 0 or -real number ≠ 0
Invalid operation	+Infinity or -infinity	+Infinity or -infinity
+0	+0	-0 Note 1
+0	-0	+0 Note 1
-0	+0	-0 Note 2
-0	-0	+0 Note 2

Notes:

1. Value is not rounded toward negative infinity.
2. Value is rounded toward negative infinity.

ASF Example

Op		B_3	E	D_2	B_1	D_1
AE	3	1	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: ASF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: AE31 4050 4060

$B_3(3)$: 0010 0200 0070

$B_1(4)$ and $B_2(4)$: 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	4E81	2345		
0010 0200 0070	4E81	2345		

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4F01	2345		
0010 0200 0060	4E81	2345		
0010 0200 0070	4E81	2345		

	Before	**After**
Condition Code:	x	2

AND BYTE (NB)

Instruction Description

The first and second operands are ANDed and the result is placed in the first-operand register.

Format: RS

79	r_1	1	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address Translation
- Addressing
- Effective Address Overflow

NB Example

Op	r_1	E	B_1	D_1
79	C	1	8	542

0 Bits 8 12 16 20 31

Assembler: NB r_1, $D_2(B_2)$

Machine: 79C1 8542

B_1(8): 1224 1932 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
1224 1932 0542		40		

	Before	After
r_1 (C):	45	40
Condition Code:	x	1

AND BYTE IMMEDIATE (NBI)

Instruction Description

The first and second operand are ANDed and the result is placed in the first-operand location.

Format: SI

98	I_2	B_1	D_1
0 Bits	8	16	20 31

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

NBI Example

Op 98	I_2 FF	B_1 7	D_1 118
0 Bits	8	16	20 31

Assembler: NBI $D_1(B_1), I_2$

Machine: 98FF 7118

$B_1(7)$: 1180 1140 3000

Storage — Before and After

	0/8	2/A	4/C	6/E
1180 1140 3118	A1			

	Before	After
Condition Code:	x	1

AND BYTE REGISTER (NBR)

Instruction Description

The first and second operands are ANDed and the result is placed in the first-operand register.

Format: RR

18	r_1	r_2

0 Bits 8 12 15

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

NBR Example

Op	r_1	r_2
18	5	6

0 Bits 8 12 15

Assembler: NBR r_1, r_2

Machine: 1856

	Before	After
r_1(5):	FF	21
r_2(6):	21	21
Condition Code:	x	1

AND BYTE REGISTER IMMEDIATE (NBRI)

Instruction Description

The first and second operand are ANDed and the result is placed in the first-operand register.

Format: RI

48	r_1	0	I_2	

0 Bits 8 12 16 24 31

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program. Exceptions: None.

NBRI Example

Op	r_1	E	I_2	
48	3	0	32	

0 Bits 8 12 16 24 31

Assembler: NBRI r_1, I_2

Machine: 4830 3200

	Before	After
r_1 (3):	4C	00
Condition Code:	x	0

AND CHARACTERS (NC)

Instruction Description

The first and second operand are ANDed and the result is placed in the first-operand location.

Format: SS

C8	L_1	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32 36	47

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands (operand fields are processed left to right) contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result =	0
1	Result ≠	0
2	--	
3	--	

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

NC Example

Op C8	L_1 03	B_1 6	D_1 540	B_2 5	D_2 240
0 Bits	8	16	20	32 36	47

Assembler: NC $D_1(L_1, B_1)$, $D_2(B_2)$

Machine: C803 6540 5240

B_1(6): 5010 6400 A000

B_2(5): 5010 6400 B000

Storage — Before

	0/8	2/A	4/C	6/E
5010 6400 A540	A1A1	B123		
5010 6400 B240	A1A1	B111		

Storage — After

	0/8	2/A	4/C	6/E
5010 6400 A540	A1A1	B101		
5010 6400 B240	A1A1	B111		

	Before	After
Condition Code:	x	1

AND HALFWORD (NH)

Instruction Description

The first and second operand are ANDed and the result is placed in the first-operand register.

Format: RS

80	R_1	4	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

NH Example

Op 80	R_1 3	E 4	B_2 5	D_2 160

0 Bits 8 12 16 20 31

Assembler: NH R_1, $D_2(B_2)$

Machine: 8034 5160

B_2(5): 5718 9423 2000

Storage — Before and After

	0/8	2/A	4/C	6/E
5718 9423 2160	0503			

	Before	After
R_1(3):	008A	0002
Condition Code:	x	1

AND HALFWORD REGISTER (NHR)

Instruction Description

The first and second operand are ANDed and the result is placed in the first-operand register.

Format: RR

28	R_1	R_2

0 Bits 8 12 15

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

NHR Example

Op	R_1	R_2
28	3	5

0 Bits 8 12 15

Assembler: NHR R_1, R_2

Machine: 2835

	Before	After
R_1(3):	008A	0002
R_2(5):	0503	0503
Condition Code:	x	1

AND HALFWORD REGISTER IMMEDIATE (NHRI)

Instruction Description

The first and second operand are ANDed and the result is placed in the first-operand register.

Format: RI

58	R_1	0	I_2

0 Bits 8 12 16 31

Operation: Operands are treated as logical quantities and the connective AND is applied bit by bit. A bit position in the result is set to one if the corresponding bit positions in both operands contain a one; otherwise, the result bit is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

NHRI Example

Op	R_1	E	I_2
58	4	0	FFFF

0 Bits 8 12 16 31

Assembler: NHRI R_1, I_2

Machine: 5840 FFFF

	Before	After
R_1(4):	A1A2	A1A2
Condition Code:	x	1

BRANCH AND LINK (BAL)

Instruction Description

The updated instruction address is loaded as link information in the halfword register designated by R_1. Subsequently, the instruction address is replaced by the branch address.

Format: RI

4F	R_1	0	D_2

0 Bits 8 12 16 31

Operation: The branch address is computed before the instruction address is loaded. The updated instruction address is replaced by the sum of the 16-bit displacement D_2 from the instruction and the offset portion of the instruction in base register 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

BAL Example

Op	R_1	E	D_2
4F	3	0	01C6

0 Bits 8 12 16 31

Assembler: BAL R_1, D_2

Machine: 4F30 01C6

	Before	Updated	After
B(0):	7314 2482 1130	–	7314 2482 1130
R_1(3):	xxxx	–	135A
IAR:	1356	135A	12F6

BRANCH AND LINK LONG (BALL)

Instruction Description

The updated instruction address is loaded as link information in the halfword register designated by R_1; and the instruction stream base address, contained in base register 0, is loaded in the base register designated by B_3. Subsequently, the instruction stream base address and instruction address are replaced by the second operand.

Format: RS

8F	R_1	B_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Bits 0-F of the second operand contain the new instruction address; bits 16-3F contain the new instruction stream base address that is loaded into base register 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand occupies 8 bytes in storage and must start on a fullword boundary; otherwise a specification exception occurs and the operation is suppressed. Both the instruction stream base address and the instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

BALL Example

Op	R_1	B_3	B_2	D_2
8F	E	F	0	7D0

0 Bits 8 12 16 20 31

Assembler: BALL R_1, B_3, $D_2(B_2)$

Machine: 8FEF 07D0

	Before	Updated	After
$B_2(0)$:	0100 D00C 0000	–	81BC 4560 0000
$B_3(F)$:	xxxx xxxx xxxx	–	0100 D00C 0000
$R_1(E)$:	xxxx	–	3308
IAR:	3304	3308	4330

Storage — Before and After

	0/8	2/A	4/C	6/E
0100 D00C 07D0	4330	81BC	4560	0000

BRANCH INTERNAL (BI)

Instruction Description

A branch is taken to the address contained in the first-operand register if that address is internal to the current segment group.

Format: RR

1E	B_1	0

0 Bits 8 12 15

Operation: The left 3 bytes of the first operand are compared with the left 3 bytes of B(0) (base register 0). If the values are not equal, an invalid segment group address exception occurs and the operation is suppressed.

If no exception is signaled, then the following is done:

1. Bytes 0-3 (the left 4 bytes) of B(0) are set equal to bytes 0-3 of the first operand.
2. Bytes 4-5 of B(0) are set equal to zero.
3. The IAR is set from bytes 4 and 5 of the first operand.
4. Execution resumes at the new B(0) and IAR location.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand must point to a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Invalid segment group address
- Specification

BI Example

Op	B_1	E
1E	3	0

0 Bits 8 12 15

Assembler: BI B_1

Machine: 1E30

	Before			After		
B(0):	1133	6422	0000	1133	6422	0000
B_1(3):	1133	6422	6420	1133	6422	6420
IAR:	0330			6420		

BRANCH ON CONDITION (BC)

Instruction Description

The updated instruction address is replaced by the branch address if the condition code is as specified by M_3; otherwise, normal instruction sequencing proceeds with the updated instruction address.

Format: RI

6E		M_3	D_2	
0 Bits	8	12	16	31

Operation: M_3 is used as a 4-bit mask. The 4 bits of the mask correspond, left to right, with the four condition codes (0, 1, 2, and 3). The branch is successful when the condition code has a corresponding mask bit of 1. A mask of all zeros results in normal instruction sequencing.

The updated instruction address is replaced by the sum of the 16-bit displacement (D_2) and the offset portion of the instruction stream base address contained in the base register 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: The IMP Instruction Assembler uses the following extended mnemonics:

Extended Mnemonics	Meaning	Standard Mnemonic	Mask
BH	Branch High	BC	2
BL	Branch Low	BC	4
BE	Branch Equal	BC	8
BNH	Branch Not High	BC	D
BNL	Branch Not Low	BC	B
BNE	Branch Not Equal	BC	7
BP	Branch Positive	BC	2
BM	Branch Minus	BC	4
BZ	Branch Zero	BC	8
BNP	Branch Not Plus	BC	D
BNM	Branch Not Minus	BC	B
BNZ	Branch Not Zero	BC	7
BO	Branch If Ones	BC	1
BM	Branch If Mixed	BC	4
BZ	Branch If Zeros	BC	8
BNO	Branch If Not Ones	BC	E

BC Example

Op 6E		M_3 4	D_2 0430	
0 Bits	8	12	16	31

Assembler: BC M_3, D_2

Machine: 6E04 0430

Condition Code: 1

	Before			After		
B(0):	5425	3111	5100	5425	3111	5100
IAR:	5860			5530		

BRANCH ON CONDITION INDIRECT (BCN)

Instruction Description

The updated instruction address is replaced by the branch address if the condition code is as specified by M_3; otherwise, normal instruction sequencing proceeds with the updated instruction address.

Format: RS

9E		M_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: M_3 is used as a 4-bit mask. The 4 bits of the mask correspond, left to right, with the four condition codes (0, 1, 2, and 3). The branch is successful when the condition code has a corresponding mask bit of 1. A mask of all zeros results in normal instruction sequencing.

The halfword at the second-operand location contains the branch displacement. The branch address is formed by adding the branch displacement to the offset portion of the instruction stream base address contained in base register 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The halfword storage operand and the updated instruction address must each start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

BCN Example

Op		M_3	B_2	D_2
9E		2	C	310

0 Bits 8 12 16 20 31

Assembler: BCN M_3, D_2 (B_2)

Machine: 9E02 C310

	Before	Updated	After
B(0):	0023 1430 5680	–	0023 1430 5680
B_2 (C):	0114 1180 4000	–	0114 1180 4000
IAR:	6234	6238	6ED0

Condition Code: 2

Storage – Before and After

	0/8	2/A	4/C	6/E
0114 1180 4310	1850			

BRANCH ON CONDITION INDIRECT INDEXED (BCNX)

Instruction Description

The updated instruction address is replaced by the branch address if the condition code is as specified by the mask; otherwise, normal instruction sequencing proceeds with the updated IAR.

Format: RS

7F	R_1	M_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: M_3 is used as a 4-bit mask. The 4 bits of the mask correspond, left to right, with the four condition codes (0, 1, 2, and 3). The branch is successful when the condition code has a corresponding mask bit of 1. A mask of all zeros results in no branch.

The contents of the halfword register specified by R_1 is added to the effective address of the second operand to form the address of a halfword in storage that contains the branch displacement. The branch address is formed by adding the branch displacement to the offset portion of the instruction stream base address contained in base register 0.

Overflow: An overflow is recognized as an effective address overflow exception, and the operation is suppressed.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The halfword storage operand and the updated instruction address must each start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

BCNX Example

Op 7F	R_1 B	M_3 8	B_2 4	D_2 630

0 Bits 8 12 16 20 31

Assembler: BCNX R_1, M_3, D_2(B_2)

Machine: 7FB8 4630

	Before	Updated	After
B(0):	0023 1430 5680	–	0023 1430 5680
B_2(4):	0375 2102 6000	–	0375 2102 6000
R_1(B):	4C20	–	4C20
IAR:	75C0	75C4	7060

Condition Code: 0

Storage — Before and After

	0/8	2/A	4/C	6/E
0375 2102 B250	19E0			

BRANCH ON COUNT (BCT)

Instruction Description

The binary quantity contained in the halfword register specified by R_1 is reduced by 1. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address is replaced by the branch address.

Format: RI

8E	R_1	0	D_2

0 Bits 8 12 16 31

Operation: The updated instruction address is replaced by the sum of the 16-bit displacement D_2 from the instruction and the offset portion of the instruction stream base address contained in base register 0, if R_1 does not equal zero.

The branch address is computed before the counting operation. Counting does not change the condition code. The subtraction proceeds as in binary arithmetic and all 16 bits of the halfword register participate in the operation.

Overflow: The overflow occurring on transition from maximum positive number is ignored.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

BCT Example

Op	R_1	E	D_2
8E	C	0	02A0

0 Bits 8 12 16 31

Assembler: BCT R_1, D_2

Machine: 8EC0 02A0

	Before	Updated	After
R_1(C):	0009	–	0008
IAR:	IEF0	1EF4	12C0
B(0):	000A 2130 1020	–	000A 2130 1020

BRANCH REGISTER (BR)

Instruction Description

The instruction address is replaced by the contents of the halfword register designated by R_1.

Format: RR

2E	R_1	0

0 Bits 8 12 15

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Specification

BR Example

Op	R_1	E
2E	C	0

0 Bits 8 12 15

Assembler: BR R_1

Machine: 2EC0

	Before	After
R_1 (C):	5320	5320
IAR:	3252	5320

BRANCH REGISTER LONG (BRL)

Instruction Description

The instruction address is replaced by the contents of the halfword register designated by R_1; the instruction stream base address, contained in base register 0, is replaced by the contents of the base register designated by B_2.

Format: RR

2F	R_1	B_2

0 Bits 8 12 15

Operation: See *Instruction Description*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Specification

BRL Example

Op	**R_1**	**B_2**
2F	9	5

0 Bits 8 12 15

Assembler: BRL R_1, B_2

Machine: 2F95

	Before	**After**
R_1(9):	14E0	14E0
B_2(5):	32A3 57C9 0000	32A3 57C9 0000
IAR:	2344	14E0
B(0):	21F2 334A 0000	32A3 57C9 0000

BRANCH UNCONDITIONAL (BU)

Instruction Description

The updated instruction address is replaced by the branch address.

Format: RI

6F		0	D_2
0 Bits	8	12	16 31

Operation: The branch address is the sum of the 16-bit displacement D_2 from the instruction and the offset portion of the instruction stream base address contained in base register 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: The IMP instruction assembler uses the extended mnemonic B meaning branch unconditional.

BU Example

Op		**E**	**D_2**
6F		0	11B0
0 Bits	8	12	16 31

Assembler: BU D_2

Machine: 6F00 11B0

	Before	**After**
B(0):	1B30 2CC0 0100	1B30 2CC0 0100
IAR:	0B20	12B0

CALL INTERNAL (CALLI)

Instruction Description

The second operand identifies a branch target. After execution of this instruction the updated instruction address is replaced by the sum of the second operand and the offset portion of base register 0.

Format: SI

EF	I_3	0	B_1	D_1	D_2	
0 Bits	8	12	16	20	32	47

Operation: The first operand points to a 16-byte area in storage, where the instruction creates a tagged pointer containing the return address by putting the leftmost two bits of I_3 into bits 0 and 1 of byte 0 of the tagged pointer. The return address points to the next instruction, which resides in the storage area immediately following the CALLI instruction. After the two bits from I_3 are put into byte 0 of the tagged pointer, the instruction zeros bits 2-7 of byte 0 of the tagged pointer. Bytes 8 and 9 are fetched from the storage location whose address is formed by concatenating hex 00 0004 to the right of the leftmost 3 bytes of base register 0. Pointer bytes 10-15 are loaded with the return address formed from the 2-byte updated IAR contents concatenated to the right of the leftmost 4 bytes of base register 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The tag pointer must be quadword aligned; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CALLI Example

Op EF	I_3 4	E 0	B_1 5	D_1 100	D_2 11B0	
0 Bits	8	12	16	20	32	47

Assembler: CALLI $D_1(B_1)$, D_2, I_3

Machine: EF40 5100 11B0

	Before	After
B(0):	1B30 2CC0 0000	1B30 2CC0 0000
B(5):	1A40 0000 0000	1A40 0000 0000
IAR:	0B20	11B0

Address	Before	After
1A40 0000 0100	xxxx xxxx	4000 0000
	xxxx xxxx	0000 0000
	xxxx xxxx	1234 1B30
	xxxx xxxx	2CC0 0B26
1B30 2CC0 0004	1234	1234

This page is intentionally left blank.

COMPARE AND SWAP HALFWORD (CSH)

Instruction Description

The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location.

Format: RS

7D	R_1	R_3	B_2	D_2	
0 Bits	8	12	16	20	31

Operation: The first and third operands are 16 bits in length, with each operand occupying a halfword register. The second operand is a halfword in main storage.

The result of the 16-bit comparison, either equal or unequal, is used to set the condition code. When the result of the comparison is unequal, no attempt to store occurs.

When an equal comparison occurs, no access by another instruction is permitted at the second-operand location between the moment that the second operand is fetched for comparison and the moment that the third operand is stored at the second-operation location.

Overflow and Sign Code: Not applicable.

Condition code:

0	First Operand	=	Second operand; second operand replaced by third operand.
1	First Operand	≠	Second operand; first operand replaced by second operand.
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements: The second operand must be on a halfword boundary; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: The Compare and Swap Halfword instruction does not interlock against storage accesses by the channel. Therefore, the instruction should not be used to update a halfword that is partly or entirely in an I/O input area, since the input data may be lost.

CSH Example

Op	R_1	R_3	B_2	D_2
7D	5	6	3	330

0 Bits 8 12 16 20 31

Assembler: CSH $R_1, R_3, D_2(B_2)$

Machine: 7D56 3330

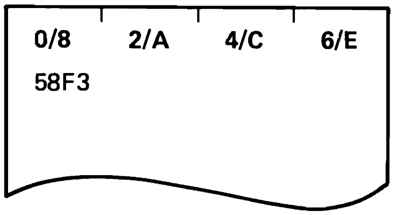
B_2(3): 1072 92D0 E000

R_1(5): 58F3

R_3(6): 845F

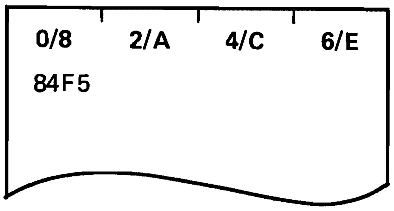
Storage — Before

1072 92D0 E330



Storage — After

1072 92D0 E330



	Before	**After**
Condition Code:	x	0

COMPARE BYTE IMMEDIATE AND BRANCH EQUAL (CBIBE)

Instruction Description

The first operand is compared with the second operand. If the operands are equal, the updated instruction address is replaced by the branch address; otherwise, normal instruction sequencing proceeds with the updated instruction address.

Format: SI

D0	I_2	B_1	D_1	D_3

0 Bits 8 16 20 32 47

Operation: The immediate operand, I_{I2}, is compared with the byte in storage addressed by B_I, D_1. If equal, the updated instruction address is replaced by the sum of the 16-bit displacement (D_3) and the offset portion of the instruction stream base address contained in base register zero.

Overflow and Sign Codes: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CBIBE Example

Op	I_2	B_1	D_1	D_3
D0	E8	e	009	4928

0 Bits 8 16 20 32 47

Assembler: CBIBE D_1 (B_1), I_2, D_3

Machine: D0E8 3009 4928

B(0): 3978 21F4 0100

B(3): 49E2 C301 0200

Storage — Before and After

	0/8	2/A	4/C	6/E
49E2 C301 0209	23	E8		

	Before	After
IAR:	093C	4A28

COMPARE BYTE IMMEDIATE AND BRANCH NOT EQUAL (CBIBN)

Instruction Description

The first operand is compared with the second operand. If the operands are equal, the updated instruction address is replaced by the branch address; otherwise, normal instruction sequencing proceeds with the updated instruction address.

Format: SI

D1	I_2	B_1	D_1	D_3

0 Bits 8 16 20 32 47

Operation: The immediate operand, I_2, is compared with the byte in storage addressed by B_1, D_1. If not equal, the updated instruction address is replaced by the sum of the 16-bit displacement (D_3) and the offset portion of the instruction stream base address contained in base register zero.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must start on a halfword boundary; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CBIBN Example

Op	I_2	B_1	D_1	D_3
D1	E8	4	005	4E31

0 Bits 8 16 20 32 47

Assembler: CBIBN $D_1(B_1)$, I_2, D_3

Machine: D1E8 4005 4E31

B(0): 56B3 4792 5AC4

B(4): 359D 0200 4EC8

Storage — Before and After

	0/8	2/A	4/C	6/E
359D 0200 4ECD			01	02

	Before	After
IAR:	6F02	6F08

COMPARE CHARACTERS (CC)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: SS

C2	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Comparison is algebraic, treating both operands in signed binary quantities. Operands in registers or storage are not changed. If the operands are unequal in length, the shorter operand is considered to be extended to the left with bits equal to the sign bit.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

CC Example

Op C2	L_1 3	L_2 3	B_1 4	D_1 320	B_2 5	D_2 130

0 Bits 8 12 16 20 32 36 47

Assembler: CC $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: C233 4320 5130

B_1(4): 1ABC 2DEF 0000

B_2(5): 312B 45C6 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
1ABC 2DEF 0320	8125	B2CC		
312B 45C6 0130	7AC0	465F		

	Before	After
Condition Code:	x	1

COMPARE HALFWORD (CH)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RS

80	R_1	2	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Comparison is algebraic, treating both operands as signed binary quantities. Operands in registers or storage are not changed.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The storage operands must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CH Example

Op	R_1	E	B_2	D_2
80	7	2	2	BC0

0 Bits 8 12 16 20 31

Assembler: CH R_1, $D_2(B_2)$

Machine: 8072 2BC0

B_2(2): 00A0 3120 0000

R_1(7): 1AF3

Storage — Before and After

	0/8	2/A	4/C	6/E
00A0 3120 0BC0	1AB2			

	Before	**After**
Condition Code:	x	2

COMPARE HALFWORD IMMEDIATE (CHI)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: SI

A2		0	B_1	D_1	I_2

0 Bits 8 12 16 20 32 47

Operation: Comparison is algebraic, treating both operands as signed binary quantities. Operands in registers or storage are not changed.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CHI Example

Op		E	B_1	D_1	I_2
A2		0	7	A10	B13C

0 Bits 8 12 16 20 32 47

Assembler: CHI $D_1(B_1)$, I_2

Machine: A200 7A10 B13C

$B_1(7)$: 000A 24B0 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
000A 24B0 0A10	B13C			

	Before	After
Condition Code:	x	0

COMPARE HALFWORD REGISTER (CHR)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RR

22	R_1	R_2

0 Bits 8 12 15

Operation: Comparison is algebraic, treating both operands as signed binary quantities. Operands in registers are not changed.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CHR Example

Op 22	R_1 3	R_2 4

0 Bits 8 12 15

Assembler: CHR R_1, R_2

Machine: 2234

R_1(3): 5590

R_2(4): 8320

	Before	After
Condition Code:	x	2

COMPARE HALFWORD REGISTER IMMEDIATE (CHRI)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RI

52	R_1	0	I_2

0 Bits 8 12 16 31

Operation: Comparison is algebraic, treating both operands as signed binary quantities. Operands in registers are not changed.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CHRI Example

Op	R_1	E	I_2
52	9	0	2243

0 Bits 8 12 16 31

Assembler: CHRI R_1, I_2

Machine: 5290 2243

R_1(9): 2233

	Before	After
Condition Code:	x	1

COMPARE LOGICAL ADDRESS REGISTER (CLAR)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RR

23	B_1	B_2
0 Bits	8	12 15

Operation: The two 6-byte operands are treated as unsigned binary integers and are compared, setting the condition code in the following manner.

First, the high-order 3 bytes (segment group) of the two operands are compared; if they are not equal, the condition code is set to 3 and the instruction is complete. If the operands are equal, the low-order 3 bytes (segment group offset) of the operands are compared as unsigned binary integers.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand.

1 High-order 3 bytes of the operands are equal; low-order 3 bytes of the first operand are low.

2 High-order 3 bytes of the operands are equal; low-order 3 bytes of the first operand are high.

3 High-order 3 bytes of the operands are not equal.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CLAR Example

Op 23	B_1 8	B_2 6
0 Bits	8	12 15

Assembler: CLAR B_1,B_2

Machine: 2386

B_1(8): 00C1 B000 4BE0

B_2(6): 00C1 B000 4BD0

	Before	After
Condition Code:	x	2

COMPARE LOGICAL BYTE (CLB)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RS

71	r_1	2	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

CLB Example

Op 71	r_1 8	E 2	B_2 6	D_2 120

0 Bits 8 12 16 20 31

Assembler: CLB r_1, $D_2(B_2)$

Machine: 7182 6120

$B_2(6)$: 4022 4045 0000

$r_1(8)$: 27

Storage — Before and After

	0/8	2/A	4/C	6/E
4022 4045 0120	27			

	Before	**After**
Condition Code:	x	0

COMPARE LOGICAL BYTE IMMEDIATE (CLBI)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition codes.

Format: SI

9C	I_2	B_1	D_1

0 Bits 8 16 20 31

Operation: The comparison is performed with the operands as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

CLBI Example

Op	I_2	B_1	D_1
9C	03	4	032

0 Bits 8 16 20 31

Assembler: CLBI $D_1(B_1)$, I_2

Machine: 9C03 4032

B_1(4): 4128 7147 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
4128 71A7 0032		32		

	Before	After
Condition Code:	x	2

COMPARE LOGICAL BYTE REGISTER (CLBR)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RR

12	r_1	r_2

0 Bits 8 12 15

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CLBR Example

Op	r_1	r_2
12	3	4

0 Bits 8 12 15

Assembler: CLBR r_1, r_2

Machine: 1234

r_1(3): 42

r_2(4): 42

	Before	After
Condition Code:	x	0

COMPARE LOGICAL BYTE REGISTER IMMEDIATE (CLBRI)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RI

42	r_1	0	I_2	

0 Bits 8 12 16 24 31

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CLBRI Example

Op 42	r_1 5	E 0	I_2 FF	

0 Bits 8 12 16 24 31

Assembler: CLBRI r_1, I_2

Machine: 4250 FF00

r_1(5): F4

	Before	After
Condition Code:	x	1

COMPARE LOGICAL CHARACTERS (CLC)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: SS

C5	L_1	B_1	D_1	B_2	D_2	
0 Bits	8	16	20	32	36	47

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

The operation proceeds left to right and ends as soon as an inequality is found or an end of the field is reached.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The first and second operands can overlap in storage. If either operand crosses a segment boundary, an effective address overflow exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

CLC Example

Op C5	L_1 00	B_1 2	D_1 001	B_2 7	D_2 000	
0 Bits	8	16	20	32	36	47

Assembler: CLC $D_1(L_1, B_1), D_2(B_2)$

Machine: C500 2001 7000

B_1(2): 4417 8418 0000

B_2(7): 4417 5232 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
4417 5232 0000	18			
4417 8418 0001	41			

	Before	After
Condition Code:	x	2

COMPARE LOGICAL CHARACTER REGISTER (CLCR)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code. The length is variable and is found as the contents of the third operand byte register.

Format: SS

E9	r_3	0	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

The operation proceeds left to right and ends as soon as an inequality is found or an end of the field is reached.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The first and second operands can overlap in storage. If either operand crosses a segment boundary, an effective address overflow exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

CLCR Example

Op E9	r_3 5	E 0	B_1 2	D_1 001	B_2 7	D_2 000	
0 Bits	8	12	16	20	32	36	47

Assembler: CLCR $D_1(B_1), D_2(B_2), R_3$

$R_3(5)$: 00

Machine: E950 2001 7000

$B_1(2)$: 4417 8418 0000

$B_2(7)$: 4417 5232 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
4417 5232 0000	18			
4417 8418 0001	41			

	Before	After
Condition Code:.	x	2

COMPARE LOGICAL CHARACTERS LONG (CLCL)

Instruction Description

The first operand is compared with the second operand and the result is indicated in the condition code.

Format: SS

EA	I_3	B_1	D_1	B_2	D_2	
0 Bits	8	16	20	32	36	47

Operation: The shorter operand is considered extended to the right with the padding character contained in the I_3 field of the instruction.

The leftmost bytes of the first and second operands as well as the lengths are located indirectly through addresses contained in storage. These addresses are 8-byte fields. Bytes 0-1 of these 8-byte fields specify 1 less than the number of bytes in the operand location; bytes 2-7 contain the address of the leftmost byte of the operand.

Length	Operand Address	
	SID	Offset
0 Bytes	2	7

The comparison is performed with the operands treated as unsigned binary quantities. The operation proceeds left to right and ends as soon as an inequality is detected or the end of the longest operand is reached. If the operands are not of the same length, the shorter operand is assumed to be extended to the right with the padding character.

If the 8-byte field associated with either field contains all zeros, the operand is assumed to be of zero length and the padding character is used for the entire field. If both 8-byte fields contain all zeros, condition code 0 is set.

The execution of the instruction is interruptible (the operation can be suspended). When an interruption occurs after a unit of operation other than the last one, the IAR is not advanced to the next instruction address, the length fields are decremented by the number of bytes compared, and the address fields are incremented by the same number, so that the instruction, when reexecuted, resumes at the point of interruption. If the operation is interrupted after the shorter operand has been exhausted, the length and address fields for that operand are all zeros.

If the operation ends because of a mismatch, the length and address fields at completion identify the byte of mismatch. The length counts are decremented by the number of bytes that matched, and the address fields are incremented by the same amount. If the mismatch occurred with the padding character, the length and address fields of the shorter operand contain all zeros. If the two operands including the padding character are equal, then the length and address fields for both operands contain all zeros.

Overflow and Sign Code: Not applicable.

Condition Code: The condition code is not set by this instruction until it has completed. Therefore, if the instruction was interrupted, no mismatch has occurred up to this point.

0	First operand	=	Second operand, or both fields are of zero length
1	First operand	<	Second operand
2	First operand	>	Second operand
3	--		

Carry: Not applicable.

Boundary Requirements: The leftmost byte of each operand address identifies an 8-byte field in storage that must begin on a word boundary and must not cross a page boundary; otherwise a specification exception occurs and the operation is suppressed. The operand fields can overlap in storage but neither may cross a segment boundary; otherwise an effective address overflow exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CLCL Example

Op	I_3	B_1	D_1	B_2	D_2
EA	FF	4	000	4	7B0

0 Bits 8 16 20 32 36 47

Assembler: CLCL $D_1(B_1)$, $D_2(B_2)$, I_3

Machine: EAFF 4000 47B0

$B_1(4)$ and $B_2(4)$: 6250 2938 0000

Storage — Before

	0/8	2/A	4/C	6/E
6250 2938 0000	0003	6250	2A00	0000
6250 2938 07B0	0007	6250	2B00	0000

Storage — After

	0/8	2/A	4/C	6/E
6250 2938 0000	0000	0000	0000	0000
6250 2938 07B0	0003	6250	2B00	0004

Storage — Before and After

	0/8	2/A	4/C	6/E
6250 2A00 0000	1234	5678		
6250 2B00 0000	1234	5678	9ABC	DEF0

	Before	**After**
Condition Code:	x	2

COMPARE LOGICAL HALFWORD (CLH)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RS

92	R_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CLH Example

Op	R_1	E	B_2	D_2
92	3	0	4	280

0 Bits 8 12 16 20 31

Assembler: CLH R_1, $D_2(B_2)$

Machine: 9230 4280

B_2(4): 9046 71A2 0000

R_1(3): 07D0

Storage – Before and After

	0/8	2/A	4/C	6/E
9046 71A2 0280	07D0			

	Before	After
Condition Code:	x	0

COMPARE LOGICAL HALFWORD IMMEDIATE (CLHI)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: SI

B2		0	B_1	D_1	I_2
0 Bits	8	12	16	20	32 47

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CLHI Example

Op		E	B_1	D_1	I_2
B2		0	3	A90	F1F3
0 Bits	8	12	16	20	32 47

Assembler: CLHI $D_1(B_1)$, I_2

Machine: B200 3A90 F1F3

$B_1(3)$: 9046 2140 A000

Storage — Before and After

	0/8	2/A	4/C	6/E
9046 2140 AA90	F1A3			

	Before	After
Condition Code:	x	1

COMPARE LOGICAL HALFWORD REGISTER (CLHR)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RR

32	R_1	R_2

0 Bits 8 12 15

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CLHR Example

Op	R_1	R_2
32	3	4

0 Bits 8 12 15

Assembler: CLHR R_1, R_2

Machine: 3234

R_1(3): 2C3E

R_2(4): 2C3E

	Before	After
Condition Code:	x	0

COMPARE LOGICAL HALFWORD REGISTER IMMEDIATE (CLHRI)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: RI

62	R_1	0	I_2

0 Bits 8 12 16 31

Operation: The comparison is performed with the operands treated as unsigned binary quantities. Operands in registers or storage are not changed by the operation.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

CLHRI Example

Op	R_1	E	I_2
62	5	0	111C

0 Bits 8 12 16 31

Assembler: CLHRI R_1, I_2

Machine: 6250 111C

R_1(5): 111F

	Before	After
Condition Code:	x	2

COMPARE LONG FLOAT (CLF)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: SS

CE		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Comparison is algebraic, considering the sign, the significand, and the exponent of each operand. Neither operand is changed as a result of operation. The comparison is made following the rules of floating-point subtraction as follows. The subtrahend is subtracted from the minuend; if the difference is 0, they compare equal. If the subtrahend is larger than the minuend, then the first operand is low. If the subtrahend is smaller than the minuend, then the first operand is high.

Floating-point values of 0 compare equal with each other even when they have different signs. Floating-point values of infinity compare equal with each other even when they have different signs, and a floating-point value of infinity compares unordered with any other floating point value. A not-a-number floating-point value compares unordered with all other values including another not-a-number value.

If a denormalized floating-point number is compared, the comparison is made as if the denormalized number had first been normalized.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 Operands are unordered

Carry: Not applicable.

Boundary Requirements: Both operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Operands may overlap only if they are coincidental; otherwise, the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point invalid operand
- Specification

Programming Note: The following is a summary of the results for various combinations of operands.

Comparison Result	First Source (Minuend)	Second Source (Subtrahend) finity
=	+0	+0
=	+0	-0
=	-0	+0
=	-0	-0
<	-Real number ≠ 0	+Real number ≠ 0
>	+Real number ≠ 0	-Real number ≠ 0
>	+Real number ≠ 0	+0
>	+Real number ≠ 0	-0
<	+0	+Real number ≠ 0
<	-0	+Real number ≠ 0
<	-Real number ≠ 0	+0
<	-Real number ≠ 0	-0
>	+0	-Real number ≠ 0
>	-0	-Real number ≠ 0
Unordered	Masked not-a-number	Any
Unordered	Any	Masked not-a-number
See note	Unmasked not-a-number	Any
See note	Any	Unmasked not-a-number
=	+Infinity	+Infinity
=	+Infinity	-Infinity
=	-Infinity	+Infinity
=	-Infinity	-Infinity
Unordered	Not infinity	+Infinity
Unordered	Not infinity	-Infinity
Unordered	+Infinity	Not infinity
Unordered	-Infinity	Not infinity

Legend:

Not Infinity = Anything but infinity or an unmasked not-a-number.

Any = Any floating-point field value.

Note: An unmasked not-a-number value results in a floating-point invalid operation exception unless the exception is masked. An unmasked not-a-number value results in an unordered comparison result if the floating-point invalid operation excpetion is masked.

CLF Example

Op		E	B_1	D_1	B_2	D_2
CE		0	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: CLF $D_1(B_1)$, $D_2(B_2)$

Machine: CE00 4050 4060

$B_1(4)$ and $B_2(4)$: 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	4000	0000	0000	0000
0010 0200 0060	4000	0000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4000	0000	0000	0000
0010 0200 0060	4000	0000	0000	0000

	Before	After
Condition Code:	x	0

COMPARE PACKED (CP)

Instruction Description

The first operand is compared with the second operand and the result is indicated in the condition code.

Format: SS

F2	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The comparison is algebraic including the signs and all digits of both operands. All digit codes are checked for validity. Invalid digit codes cause a data exception and the operation is terminated. If the fields are unequal in length, the shorter field is considered extended to the left with zeros.

Overflow: Not applicable.

Sign Code: All sign codes are checked for validity, and any valid plus or minus sign is considered equal to any other plus or minus sign, respectively. Invalid sign codes cause a data exception and the operation is terminated.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 --

Carry: Not applicable.

Boundary Requirements: The first and second-operand fields can overlap when their rightmost bytes coincide. Because digit and sign codes are checked for validity, improperly overlapping fields cause data exceptions, and the operation is terminated.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow

CP Example

Op F2	L_1 4	L_2 3	B_1 3	D_1 410	B_2 4	D_2 570

0 Bits 8 12 16 20 32 36 47

Assembler: CP $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: F243 3410 4570

B_1(3): 45C8 6928 5000

B_2(4): 45C8 6053 4000

Storage — Before and After

	0/8	2/A	4/C	6/E
45C8 6053 4570	6121	521F		
45C8 6928 5410	7061	2152	1F	

	Before	After
Condition Code:	x	2

COMPARE SHORT FLOAT (CSF)

Instruction Description

The first operand is compared with the second operand, and the result determines the setting of the condition code.

Format: SS

AE		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Comparison is algebraic, considering the sign, the significand, and the exponent of each operand. Neither operand is changed as a result of operation. The comparison is made following the rules of floating-point subtraction as follows. The subtrahend is subtracted from the minuend; if the difference is 0, they compare equal. If the subtrahend is larger than the minuend, then the first operand is low. If the subtrahend is smaller than the minuend, then the first operand is high.

Floating-point values of 0 compare equal with each other even when they have different signs. Floating-point values of infinity compare equal with each other even when they have different signs, and a floating-point value. A not-a-number floating-point value compares unordered with all other values including another not-a-number.

If a denormalized floating-point number is compared, the comparison is made as if the denormalized number had first been normalized.

Overflow and Sign Code: Not applicable.

Condition Code:

0 First operand = Second operand
1 First operand < Second operand
2 First operand > Second operand
3 Operands are unordered

Carry: Not applicable.

Boundary Requirements: Both operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed. Operand may overlap only if they are coincidental; otherwise, the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point invalid operand
- Specification

Programming Note: The following is a summary of the results for various combinations of operands.

Comparison Result	First Source (Minuend)	Second Source (Subtrahend)
=	+0	+0
=	+0	-0
=	-0	+0
=	-0	-0
<	-Real number ≠ 0	+Real number ≠ 0
>	+Real number ≠ 0	-Real number ≠ 0
>	+Real number ≠ 0	+0
>	+Real number ≠ 0	-0
<	+0	+Real number ≠ 0
<	-0	+Real number ≠ 0
<	-Real number ≠ 0	+0
<	-Real number ≠ 0	-0
>	+0	-Real number ≠ 0
>	-0	-Real number ≠ 0
Unordered	Masked not-a-number	Any
Unordered	Any	Masked not-a-number
See note	Unmasked not-a-number	Any
See note	Any	Unmasked not-a-number
=	+Infinity	+Infinity
=	+Infinity	-Infinity
=	-Infinity	+Infinity
=	-Infinity	-Infinity
Unordered	Not infinity	+Infinity
Unordered	Not infinity	-Infinity
Unordered	+Infinity	Not infinity
Unordered	-Infinity	Not infinity

Legend:

Not Infinity = Anything but infinity or an unmasked not-a-number.

Any = Any floating-point field value.

Note: An unmasked not-a-number value results in a floating-point invalid operation exception unless the exception is masked. An unmasked not-a-number value results in an unordered comparison result if the floating-point invalid operation excpetion is masked.

CSF Example

Op		E	B_1	D_1	B_2	D_2
AE		0	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: CSF $D_1(B_1)$, $D_2(B_2)$

Machine: AE00 4050 4060

$B_1(4)$ and $B_2(4)$: 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	4000	0000		
0010 0200 0060	C000	0000		

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4000	0000		
0010 0200 0060	C000	0000		

	Before	**After**
Condition Code:	x	2

This page is intentionally left blank.

COMPUTE ADDRESS LONG (CAL)

Instruction Description

The value located in storage by the second-operand address is used as a signed displacement to be added to the address value in the base register identified by the third operand; the resultant address is placed in the base register identified by the first operand.

Format: RS

4C	B_1	B_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The displacement value is a 32-bit signed integer, occupying 4 bytes of storage at the second-operand location.

The address computation is performed as follows. The rightmost 3 bytes of the address value 1 identified by the third operand are logically padded on the left with 1 byte of zeros. The displacement identified by the second operand 2 is then added to this value following the rules of signed arithmetic. The result of this calculation 3 must satisfy the following validity checks:

- Must not be greater than a value of 16 megabytes less 1 (FF FFFF or decimal 16 777 215).

- Must be a positive result.

- Must not be less than the value of the 3-byte logical binary field in storage located at the address 4 determined by concatenating hex 00 001D on the right of the leftmost 3 bytes of the third-operand address value.

If any of the above checks fail, an invalid segment group address exception occurs and the operation is suppressed. Otherwise, the rightmost 3 bytes of the calculated result are concatenated on the right with the leftmost 3 bytes of the third-operand address value forming the resultant address 5. No storage reference is made using the resultant address placed in the first operand, so that the address is not inspected for addressing exceptions.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must start on a fullword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group address
- Specification

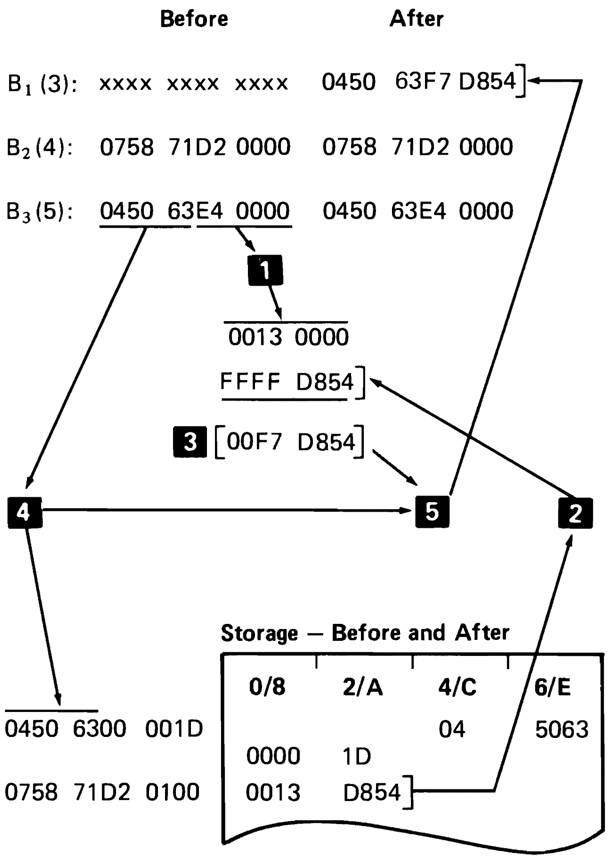
CAL Example

Op	B_1	B_3	B_2	D_2
4C	3	5	4	100

0 Bits 8 12 16 20 31

Assembler: CAL B_1, B_3, D_2 (B_2)

Machine: 4C35 4100



Before
After
B1 (3): xxxx xxxx xxxx 0450 63F7 D854
B2 (4): 0758 71D2 0000 0758 71D2 0000
B3 (5): 0450 63E4 0000 0450 63E4 0000
1
0013 0000
FFFF D854
3
00F7 D854
4
5
2
Storage — Before and After
0/8
2/A
4/C
6/E
04
5063
0000
1D
0013
D854
0450 6300 001D
0758 71D2 0100

COMPUTE ADDRESS LONG HALFWORD (CALH)

Instruction Description

The value located in storage by the second-operand address is used as a signed displacement to be added to the address value in the base register identified by the third operand; the resultant address is placed in the base register identified by the first operand.

Format: RS

4D	B_1	B_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The displacement value is a 16-bit signed integer on a halfword boundary. If the integer is not halfword aligned, a specification exception is recognized and the operation is suppressed. The sign bit is propagated through the third and fourth (left) offset bytes, and a 4-byte signed binary add is performed.

The address computation is performed as follows. The rightmost 3 bytes (segment group offset) of the address value **1** identified by the third operand are logically padded on the left with 1 byte of zeros, creating a positive 4-byte binary integer. The displacement identified by the second operand **2** is then added to this value following the rules of signed arithmetic. The result of this calculation **3** must satisfy the following validity checks:

- Must not be greater than a value of 16 megabytes less 1 (hex FF FFFF or decimal 16 777 215).

- Must be a positive result.

- Must not be less than the value of the space locator offset, 3-byte logical binary field in storage located at the address **4** determined by concatenating hex 00 001D on the right of the leftmost 3 bytes of the third-operand address value.

If any of the above checks fail, an invalid segment group address exception occurs and the operation is suppressed. Otherwise, the rightmost 3 bytes of the calculated result are concatenated on the right with the leftmost 3 bytes of the third-operand address value (segment group identifier) forming the resultant address **5**. No storage reference is made using the resultant address placed in the first operand, so that the address is not inspected for addressing exceptions.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must start on a halfword boundary; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group address
- Specification

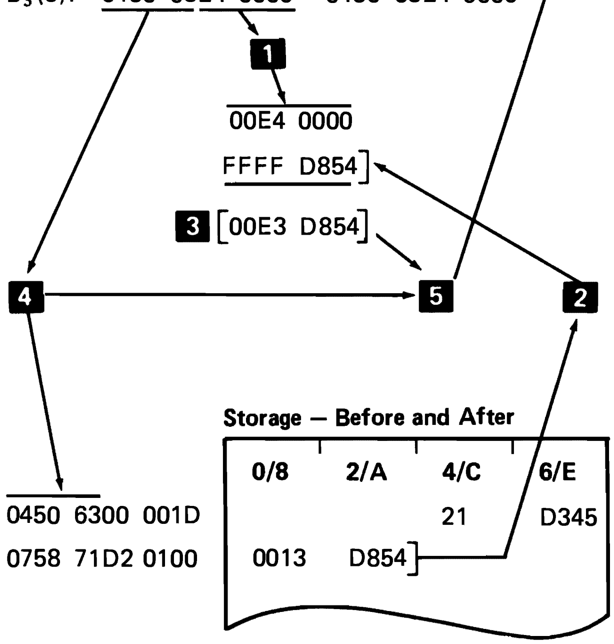
CALH Example

Op 4D	B_1 8	B_3 5	B_2 4	D_2 102
0 Bits	8	12	16	20 31

Assembler: CALH $B_1, B_3, D_2(B_2)$

Machine: 4D85 4102

	Before	After
B_1(8):	xxxx xxxx xxxx	0450 63E3 D854
B_2(4):	0758 71D2 0000	0758 71D2 0000
B_3(5):	0450 63E4 0000	0450 63E4 0000



Storage — Before and After

0/8	2/A	4/C	6/E
		21	D345
0013	D854		

COMPUTE ADDRESS LONG HALFWORD IMMEDIATE (CALHI)

Instruction Description

The second operand (I_2) is used as a signed displacement to be added to the address value in the base register identified by the third operand; the resultant address is placed in the base register identified by the first operand.

Format: RI

5C	B_1	B_3	I_2
0 Bits	8	12	16 31

Operation: The displacement value is a 16-bit signed integer. The sign bit is propagated through the third and fourth (left) offset bytes, and a 4-byte signed binary add is performed.

The address computation is performed as follows. The rightmost 3 bytes (segment group offset) of the address value (1) identified by the third operand are logically padded on the left with 1 byte of zeros, creating a positive 4-byte binary integer. The displacement identified by the second operand (2) is then added to this value following the rules of signed arithmetic. The result of this calculation (3) must satisfy the following validity checks:

- Must not be greater than a value of 16 megabytes less 1 (hex FF FFFF or decimal 16 777 215).

- Must be a positive result.

- Must not be less than the value of the space locator offset, 3-byte logical binary field in storage located at the address (4) determined by concatenating hex 00 001D on the right of the leftmost 3 bytes of the third-operand address value.

If any of the above checks fail, an invalid segment group address exception occurs and the operation is suppressed. Otherwise, the rightmost 3 bytes of the calculated result are concatenated on the right with the leftmost 3 bytes of the third-operand address value (segment group identifier) forming the resultant address (5). No storage reference is made using the resultant address placed in the first operand, so that the address is not inspected for addressing exceptions.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must start on a halfword boundary; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Invalid segment group address

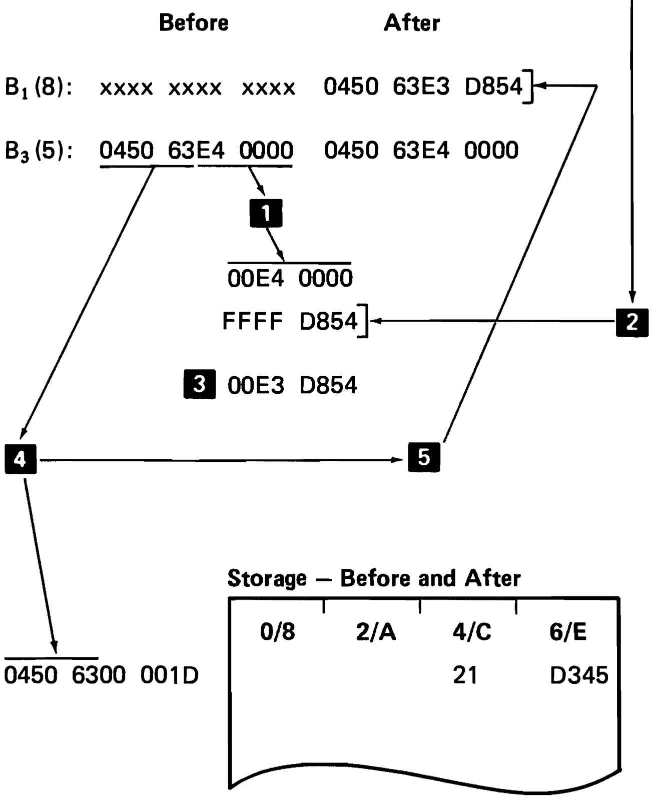
CALHI Example

Op	B_1	B_3	I_2
5C	8	5	D854

0 Bits 8 12 16 31

Assembler: CALHI B_1, B_3, I_2

Machine: 5C85 D854



	Before	After
B_1(8):	xxxx xxxx xxxx	0450 63E3 D854
B_3(5):	0450 63E4 0000	0450 63E4 0000

Storage — Before and After

0/8	2/A	4/C	6/E
		21	D345

COMPUTE LONG FLOAT MATH FUNCTION USING ONE INPUT VALUE (CLFMF1)

Instruction Description

The operation is performed by computing the mathematical function according to the controls (operand 3). The source (operand 2) is used as the argument, and the result is placed into the receiver (operand 1). The computation is always done in floating-point.

Format: SS

CE		E	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first and second operands occupy 8 bytes each, and have the long floating-point field format.

The third operand, halfword register hex F, contains control information that indicates the mathematical function to be performed. The meaning of the 2-byte control information is:

Hex Value	Meaning
0001	Sine
0003	Cosine
0005	Tangent
0006	Arc tangent
0010	Exponential function
0011	Natural logarithm (base e)
0020	Square root

All other values are reserved

- *Sine* (hex 0001). The sine of the numeric value of the source operand, whose value is considered to be in radians, is computed and placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

  The result is in the range $-1 \leq \text{SIN}(x) \leq 1$.

- *Cosine* (hex 0003). The cosine of the numeric value of the source operand, whose value is considered to be in radians, is computed and placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

  The result is in the range $-1 \leq \text{COS}(x) \leq 1$.

- *Tangent* (hex 0005). The tangent of the source operand, whose value is considered to be in radians, is computed, and the result is placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

- *Arc Tangent* (hex 0006). The arc tangent of the source operand is computed, and the result (in radians) is placed in the receiver operand.

  If the source operand is a value of positive infinity in affine mode, the result is +pi/2.

  If the source operand is a value of negative infinity in affine mode, the result is -pi/2.

  The result is in the range $-\text{pi}/2 \leq \text{ATAN}(x) \leq \text{pi}/2$.

- *Exponential Function* (hex 0010). The value e is raised to the power specified in the source operand, and the result is placed in the receiver operand.

  If the source operand is a value of positive infinity in affine mode, the result is positive infinity. If the source operand is a value of negative infinity in affine mode, the result is positive 0.

- *Natural Logarithm (base e)* (hex 0011). The natural logarithm of the source operand is computed, and the result is placed in the receiver operand.

  If the source operand is a value of 0, the result is negative infinity.

- *Square Root* (hex 0020). The square root of the numeric value of the source operand is computed and placed in the receiver operand.

  If the source operand has a value of negative 0, the result is negative 0. Any attempt to form the square root of any other negative value causes a floating-point invalid operation exception to be signaled.

  The square root of positive infinity is positive infinity.

  The result is accurate to the least significant bit.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

CLFMF1 Example

Op		E	B_1	01	B_2	D_2
CE		E	2	040	2	048

0 Bits 8 12 16 20 32 36 47

Assembler: CLFMF1 $D_1(B_1)$, $D_2(B_2)$

Machine: CE0E 2040 2048

$B_1(2)$ and $B_2(2)$: 800D 0C00 0000

R(F): 0020

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0040	xxxx	xxxx	xxxx	xxxx
800D 0C00 0048	4000	0000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0040	3FF6	A09E	667F	3BCC
800D 0C00 0048	4000	0000	0000	0000

COMPUTE LONG FLOAT MATH FUNCTION USING TWO INPUT VALUES (CLFMF2)

Instruction Description

The operation is performed by computing the mathematical function specified in the controls (operand 4). The two source values (one is operand 2 and the other is addressed by operand 3) are used as arguments and the result is placed into the receiver (operand 1). The computation is always done in floating-point.

Format: *SS*

CE	B_3	F	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first and second operands, and the data addressed by operand 3, each occupy 8 bytes and have the long floating-point field format.

Operand 3, bits 8 through 11, specifies a base register that contains the address of the second souce operand.

The fourth operand, halfword register hex F, contains control information that indicates the mathematical function to be performed. The meaning of the 2-byte control information is:

Hex Value	**Meaning**
0001	Power (X to the Y)
All other values are reserved	

- *Power (X to the Y)* (hex 0001). The computation X power Y, where X is the first source operand (operand 2) and Y is the second (operand 3), is performed, and the result is placed in the receiver operand (operand 1).

  For each combination of the two source values that would deliver a complex value as the result, a floating-point invalid operand exception is signaled (for example, if source 1 (operand 2) is a real number less than 0 and source 2 (operand 3) is 1/2).

Some special cases in affine mode are:

Source 1	**Source 2**	**Result**
Infinity	Infinity	Infinity
Infinity	±Infinity	Invalid operation

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

CLFMF2 Example

Op	B_3	E	B_1	D_1	B_2	D_2
CE	E	F	2	040	2	050

0 Bits 8 12 16 20 32 36 47

Assembler: CLFMF2 $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: CEEF 2040 2050

B_1(2) and B_2(2): 800D 0C00 0000

B_3(E): 800D 0C00 0300

R(F): 0001

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0040	xxxx	xxxx	xxxx	xxxx
800D 0C00 0050	405E	C000	0000	0000
800D 0C00 0300	4000	0000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0040	40CD	8C80	0000	0000
800D 0C00 0050	405E	C000	0000	0000
800D 0C00 0300	4000	0000	0000	0000

This page is intentionally left blank.

COMPUTE SHORT FLOAT MATH FUNCTION USING ONE INPUT VALUE (CSFMF1)

Instruction Description

The operation is performed by computing the mathematical function according to the controls (operand 3). The source (operand 2) is used as the argument, and the result is placed into the receiver (operand 1). The computation is always done in floating-point.

Format: SS

AE		E	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first and second operands occupy 4 bytes each and have the short floating-point field format.

The third operand, halfword register hex F, contains control information that indicates the mathematical function to be performed. The meaning of the 2-byte control information is:

Hex Value	Meaning
0001	Sine
0003	Cosine
0005	Tangent
0006	Arc tangent
0010	Exponential function
0011	Natural logarithm (base e)
0020	Square root

All other values are reserved

- *Sine* (hex 0001). The sine of the numeric value of the source operand, whose value is considered to be in radians, is computed and placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

  The result is in the range $-1 \leq \text{SIN(X)} \leq 1$.

- *Cosine* (hex 0003). The cosine of the numeric value of the source operand, whose value is considered to be in radians, is computed and placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled. The result is in the range $-1 \leq \text{COS(c)} \leq 1$.

- *Tangent* (hex 0005). The tangent of the source operand, whose value is considered to be in radians, is computed, and the result is placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

- *Arc Tangent* (hex 0006). The arc tangent of the source operand is computed, and the result (in radians) is placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

  The result is in the range $-\text{pi}/2 \leq \text{ATAN(x)} \leq \text{pi}/2$.

- *Exponential Function* (hex 0010). The value e is raised to the power specified in the source operand, and the result is placed in the receiver operand.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

- *Natural Logarithm (base e)* (hex 0011). The natural logarithm of the source operand is computed, and the result is placed in the receiver operand.

  If the source operand is a value of 0 or less than 0, a specification exception is signaled.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled.

- *Square Root* (hex 0020). The square root of the numeric value of the source operand is computed and placed in the receiver operand.

  If the source operand has a value of negative 0, the result is negative 0. Any attempt to form the square root of any other negative value causes a specification exception to be signaled.

  If the source operand is a value of infinity, a floating-point invalid operand exception is signaled. The result is accurate to the least significant bit.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- *Address translation*
- *Addressing*
- *Effective address overflow*
- *Floating-point invalid operand*
- *Floating-point overflow*
- *Floating-point underflow*
- *Specification*

CSFMF1 Example

Op		E	B_1	D_1	B_2	D_2
AE		E	2	058	2	05C

0 Bits 8 12 16 20 32 36 47

Assembler: CSFMF1 $D_1(B_1)$, $D_2(B_2)$

Machine: AE0E 2058 205C

B_1(2) and B_2(2): 800D 0C00 0000

R(F): 0020

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0058	xxxx	xxxx		
800D 0C00 005C			4000	0000

Storage — After

	0/8	2/A	4/C	6/E
8000 0C00 0058	3FB5	04F3		
8000 0C00 005C			4000	0000

COMPUTE SHORT FLOAT MATH FUNCTION USING TWO INPUT VALUES (CSFMF2)

Instruction Description

The operation is performed by computing the mathematical function specified in the controls (operand 4). The two source values (one is operand 2 and the other is addressed by operand 3) are used as arguments, and the result is placed into the receiver (operand 1). The computation is always done in floating-point.

Format: SS

AE	B_3	F	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The first and second operands, and the data addressed by operand 3, each occupy 4 bytes and have the short floating-point field format.

Operand 3, bits 8 through 11, specifies a base register that contains the address of the second source operand.

The fourth operand, halfword register hex F, contains control information that indicates the mathematical function to be performed. The meaning of the 2-byte control information is:

Hex Value	**Meaning**
0001	Power (X to the Y)

All other values are reserved

- (Power (X to the Y) (hex 0001). The computation X power Y, where X is the first source operand (operand 2) and Y is the second (operand 3), is performed, and the result is placed in the receiver operand (operand 1).

  For each combination of the two source values that would deliver a complex value as the result, a specification exception is signaled (for example, if source 1 (operand 2) is a real number less than 0 and source 2 (operand 3) is 1/2).

  If both source operands have a value of 0, a specification exception is signaled.

  If either of the source operands is a value of infinity, a floating-point invalid operand exception is signaled.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

CSFMF2 Example

Op	B_3	E	B_1	D_1	B_2	D_2
AE	E	F	2	058	2	060

0 Bits 8 12 16 20 32 36 47

Assembler: CSFMF2 $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: AEEF 2058 2060

$B_1(2)$ and $B_2(2)$: 800D 0C00 0000

$B_3(E)$: 800D 0C00 0300

R(F): 0001

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0058	xxxx	xxxx		
800D 0C00 0060	405E	C000		
800D 0C00 0300	4000	0000		

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0058	4141	D190		
800D 0C00 0060	405E	C000		
800D 0C00 0300	4000	0000		

COMPUTE SUBSCRIPT ADDRESS (CSA)

Instruction Description

The value of the second operand in storage is reduced by one and multiplied by I_3. The product of this multiplication is added to the first operand, and the sum is placed in the first-operand register.

Format: SI

AD	B_1	0	B_2	D_2	I_3
0 Bits	8	12	16	20	32 47

Operation: The second operand is unchanged by the operation. If the second or third operand is zero, a specification exception is raised and the operation is suppressed.

The first operand is treated as a virtual address. The second and third operands are treated as 16-bit unsigned binary integers. The second operand, which occupies 2 bytes in storage, is reduced by a value of 1 and multiplied by the contents of the I_3 field from the instruction. This product, which is considered to be a 24-bit unsigned binary integer, is then added to the contents of the base register designated by B_1, and the sum replaces the contents of the register.

Overflow: If a carry occurs from bit 24 to bit 23 as a result of either the multiply or the add operation, an effective address overflow exception occurs and the operation is suppressed.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CSA Example

Op	B_1	E	B_2	D_2	I_3
AD	7	0	3	410	0050
0 Bits	8	12	16	20	32 47

Assembler: CSA B_1, D_2 (B_2), I_3

Machine: AD70 3410 0050

B_2(3): 6A39 8357 B000

	Before	After
B_1(7):	5328 C102 03B0	5328 C102 D5B0

Storage — Before and After

	0/8	2/A	4/C	6/E
6A39 8357 B410	02A1			

(02A1–1) (0050) + 5328 C102 03B0 = 5328 C102 D5B0

This page is intentionally left blank.

COMPUTE SUBSCRIPT ADDRESS CONSTRAINED (CSAC)

Instruction Description

The value of the third operand in storage is validated, reduced by one, and multiplied by the halfword found in bytes 4-5 of the second storage operand. The product of this multiplication is added to the fourth operand and the sum is placed in the first-operand register.

Format: SS

BF	B_1	B_4	B_2	D_2	B_3	D_3
0 Bits	8	12	16	20	32	36 47

Operation: The second, third, and fourth operands are unchanged by the operation. If the second-operand bytes 0-3, the second-operand bytes 4-5, or the third operand contains zero values, a specification exception is recognized and the operation is suppressed.

The fourth operand is treated as a 6-byte virtual address. The third operand is a 32-bit logical value. If it and the 6-byte second-operand field are not fullword aligned, a specification exception is recognized and the operation is suppressed. The third-operand value is validated as being nonzero, but less than hex 0100 0000 (that is, the high-order byte must be zero) and less than or equal to the limit value found in bytes 0-3 (a 32-bit logical value) of the second operand. If found to be outside this range, a specification exception is recognized and the operation is suppressed. If valid, the third operand is reduced by a value of one and multiplied by the logical value found in bytes 4-5 of the second operand. This product is a 32-bit logical value with an absolute value of less than hex 0100 0000; otherwise, an invalid segment group address exception is recognized and the operation is suppressed. The product is added to the base register designated by the fourth operand, and the result is placed into the first-operand base register.

Overflow and Sign Code: Not applicable.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: If a carry occurs from the 3-byte offset as a result of the add operation, an invalid segment group address exception occurs and the operation is suppressed.

Boundary Requirements: The second and third operands must be fullword aligned. If not, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group!tf –Specification

CSAC Example

Op BF	B_1 6	B_4 5	B_2 4	D_2 100	B_3 3	D_3 200

0 Bits 8 12 16 20 32 36 47

Assembler: CSAC B_1, $D_2(B_2)$, $D_3(B_3)$, B_4

Machine: BF65 4100 3200

	Before	After
B(3):	0001 2345 0000	0001 2345 0000
B(4):	0001 2345 0000	0001 2345 0000
B(5):	9999 9955 5555	9999 9955 5555
B(6):	6666 6666 6666	9999 9955 7777

Main storage is unchanged by the operation.

Storage – Before and After

	0/8	2/A	4/C	6/E
0001 2345 0100	0034	5678	0002	
0001 2345 0200	0000	1112	xxxx	

COMPUTE SUBSCRIPT ADDRESS CONSTRAINED HALFWORD (CSACH)

Instruction Description

The value of the third operand in storage is validated, reduced by one, and multiplied by the halfword found in bytes 4-5 of the second storage-operand. The product of this multiplication is added to the fourth operand and the sum is placed in the first-operand register.

Format: SS

AF	B_1	B_4	B_2	D_2	B_3	D_3
0 Bits	8	12	16	20	32	36 47

Operation: The second, third, and fourth operands are unchanged by the operation. If the second-operand bytes 0-3, the second-operand bytes 4-5, or third operand contains zero values, a specification exception is recognized and the operation is suppressed.

The fourth operand is treated as a 6-byte virtual address. The third operand is a 16-bit logical value. If the third-operand field is not halfword aligned, or the second operand fullword aligned, a specification exception is recognized and the operation is suppressed. The third-operand value is validated as being nonzero, but less than hex 8000 (that is, the high-order bit must be zero) and less than or equal to the limit value found in bytes 0-3 (a 32-bit logical value) of the second operand. If found to be outside this range, a specification exception is recognized and the operation is suppressed. If valid, the third operand is reduced by a value of one and multiplied by the logical value found in bytes 4-5 of the second operand. This product is a 32-bit logical value with an absolute value of less than hex 0100 0000; otherwise, an invalid segment group address exception is recognized and the operation is suppressed. The product is added to the base register designated by the fourth operand and the result is placed into the first-operand base register.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: If a carry occurs from the 3-byte offset of the fourth operand as a result of the add operation, an invalid segment group exception is recognized and the operation is suppressed.

Boundary Requirements: The third operand must be halfword aligned and the second operand 6-byte field fullword aligned. If not, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group
- Specification

CSACH Example

Op	B_1	B_4	B_2	D_2	B_3	D_3
AF	6	5	4	100	3	200

0 Bits 8 12 16 20 32 36 47

Assembler: CSACH B_1, $D_2(B_2)$, $D_3(B_3)$, B_4

Machine: AF65 4100 3200

	Before	After
B(3):	0001 2345 0000	0001 2345 0000
B(4):	0001 2345 0000	0001 2345 0000
B(5):	9999 9955 5555	9999 9955 5555
B(6):	6666 6666 6666	9999 9955 7777

Storage – Before and After

	0/8	2/A	4/C	6/E
0001 2345 0100	0034	5678	0002	
0001 2345 0200	1112	xxxx	xxxx	

CONVERT BINARY TO LONG FLOAT (CVBLF)

Instruction Description

The value of the second operand is converted from binary to floating point, and the result is placed in the first operand location.

Format: SS

CE	L_2	9	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first operand (receiver) occupies 8 bytes of storage in order to accomodate the long floating-point format.

The second operand (source) occupies either 4 or 8 bytes. The length (4 or 8 bytes) is specified by the length operand (bits 8 through 11) in the instruction. The length operand has the following format:

Bits	Meaning
8	Reserved
9-11	Length of source 011 = 4 bytes 111 = 8 bytes All other values are invalid

The second operand contents is treated as a right-aligned, signed binary integer value (whole number rather than a fraction) with an assumed binary point to the right of its rightmost digit.

The result of the operation is a normalized floating-point number, rounded, if necessary, according to the rounding mode specified in the task dispatching element.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Operand 1 data must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed. The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Specification

CVBLF Example

Op CE	L_2 7	E 9	B_1 4	D_1 050	B_2 4	D_2 060

0 Bits 8 12 16 20 32 36 47

Assembler: CVBLF $D_1(B_1)$, $D_2(L_2B_2)$

Machine: CE79 4050 4060

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	0000	0000	0000	00FF

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	406F	E000	0000	0000
0010 0200 0060	0000	0000	0000	00FF

Condition Code: Not changed.

CONVERT BINARY TO PACKED (CVBP)

Instruction Description

The radix of the second operand is changed from binary to decimal, and the result is placed in the first-operand location.

Format: SS

F8		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The number is treated as a right-aligned, binary value both before and after conversion.

The second operand is a 32-bit, signed, binary integer occupying a word in storage. The first operand occupies 8 bytes in storage and is formed using the packed decimal format with the rightmost 4 bits representing the sign.

Overflow: Not applicable.

Sign Code: The preferred signs are used for the result as follows: a positive sign is encoded as 1111 (hex F); a negative sign is encoded as 1101 (hex D).

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Both operands must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed. The operands can overlap in storage.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

CVBP Example

Op F8		E 0	B_1 3	D_1 210	B_2 3	D_2 320

0 Bits 8 12 16 20 32 36 47

Assembler: CVBP $D_1(B_1)$, $D_2(B_2)$

Machine: F800 3210 3320

B_1(3) and B_2(3): 0310 0004 A000

Storage — Before

	0/8	2/A	4/C	6/E
0310 0004 A210	xxxx	xxxx	xxxx	xxxx
0310 0004 A320	0021	3FA4		

Storage — After

	0/8	2/A	4/C	6/E
0310 0004 A210	0000	0000	2178	980F
0310 0004 A320	0021	3FA4		

CONVERT BINARY TO SHORT FLOAT (CVBSF)

Instruction Description

The value of the second operand is converted from binary to floating point, and the result is placed in the first operand location.

Format: SS

CE	L_2	9	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first operand (receiver) occupies 4 bytes of storage in order to accomodate the short floating-point format.

The second operand (source) occupies either 4 or 8 bytes. The length (4 or 8 bytes) is specified by the length operand (bits 8 through 11) in the instruction. The length operand has the following format:

Bits	Meaning
8	Reserved
9-11	Length of source 011 = 4 bytes 111 = 8 bytes All other values are invalid

The second operand contents is treated as a right-aligned, signed binary integer value (whole number rather than a fraction) with an assumed binary point to the right of its rightmost digit.

The result of the operation is a normalized floating-point number, rounded, if necessary, according to the rounding mode specified in the task dispatching element.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Operand 1 data must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed. The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Specification

CVBSF Example

Op AE	L_2 3	E 9	B_1 4	D_1 050	B_2 4	D_2 060

0 Bits 8 12 16 20 32 36 47

Assembler: CVBSF $D_1(B_1)$, $D_2(L_2B_2)$

Machine: AE39 4050 4060

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx		
0010 0200 0060	00FF	0000		

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4B7F	0000		
0010 0200 0060	00FF	0000		

Condition Code: Not changed.

CONVERT CHARACTER TO SNA (CVTCS)

Instruction Description

The operation converts the data at the second operand location from character to SNA (systems network architecture) compressed format. The conversion is controlled by information whose address is in the base register specified in the third operand. The result is placed in the first operand.

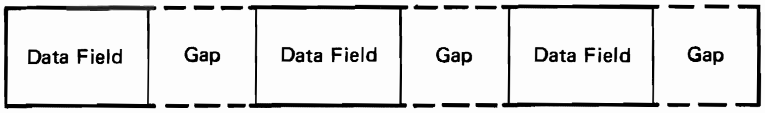
The operands are as follows:

Operand	Description
1	The base and displacement for the starting address of the result string that is to contain the converted data.
2	The base and displacement for the starting address of the source string that contains the data to be converted.
3	The base register that contains the address of the control information for the conversion operation to be performed.
4	Halfword register 14 specifies the length of the first operand (result string). A length of zero causes a specification exception.
5	Halfword register 15 specifies the length of the second operand (source string). A length of zero causes a specification exception.

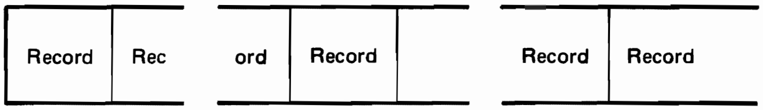
The source operand (2) contains one or more fixed-length data fields that may be separated by fixed-length gaps of characters to be ignored during the conversion. The source operand is described by the controls operand (3), which also specifies the number of bytes of data from the source to be processed to produce a converted record in the result string. The source record length does not need to be the same as the source data field length.

The following diagram explains this structure for the source operand:

Actual Source Operand Bytes



Data Processed as Source Records



For example, notice that the record length is less than the data field length and some records may have gaps in the middle.

The controls operand is a 15-byte string that specifies additional information to be used to control the conversion. The controls operand has the following format:

Result Offset	Source Offset	Mod-ifier	Source Record Length	Source Data Field Length	Gap Offset	Gap Length	Record Separator Character	Prime Compression Character	Unconverted Source Record Bytes
0	2	4	5	6	8	10	12	13	14

Bytes	Description
0-1	Offset into the result operand
2-3	Offset into the source operand
4	Modifier
5	Source record length (>0 if record processing is specified)
6-7	Data field length
8-9	Offset to the next gap in the source operand
10-11	Gap length
12	Record separator character
13	Prime compression character
14	Unconverted source record bytes

Initially, the source offset and the result offset fields specify which byte of the source field is to be processed next, and where the next byte of the result shoud be entered in the result field. The source and result offset fields are set to values that indicate how much of the conversion is complete when the instruction is interrupted or complete. An initial offset beyond the end of the related source or result operand causes a specification exception.

The *modifier* has the following valid values:

Bit(s) **Meaning**

0 Compression

- 0 = Do not perform compression
- 1 = Perform full compression

1-2 Processing Mode

- 00 = String processing. Do not use record separators; do not do blank truncation; do not perform data transparency conversion.
- 01 = Reserved
- 10 = Record processing. Use record separators and do blank truncation; do not perform data transparency conversion.
- 11 = Record processing. Use record separators, do blank truncation, and perform data transparency conversion.

3 Do not perform record Spanning

- 0 = Do not perform record spanning.
- 1 = Perform record spanning (allowed only if bit 1 = 1).

4-7 Reserved. Must be zero.

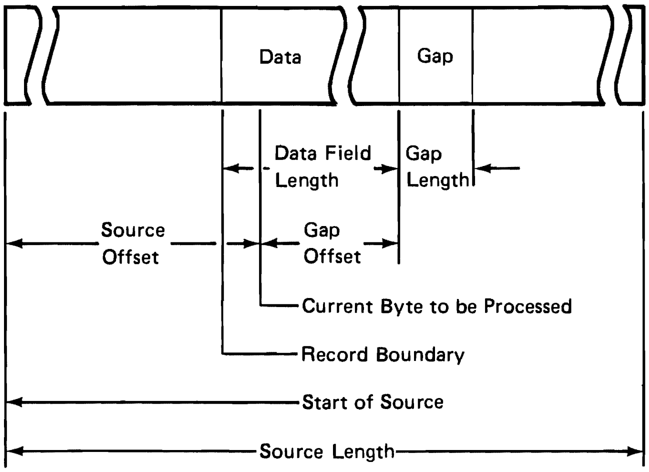
Note: An invalid modifier value causes a specification exception.

The *source record length* specifies the number of bytes to be processed to produce a converted record in the result operand. If record processing a source record length of zero results in a specification exception. Data fields in the source may be separated by gaps of characters. These gaps are ignored during conversion.

The *source data field length* specifies the number of bytes in the source data fields. Specifying a data field length of zero indicates the source length is one data field; in this case, the gap length and gap offset are ignored.

The following diagrams illustrate the makeup of the source and result operands.

Source Operand



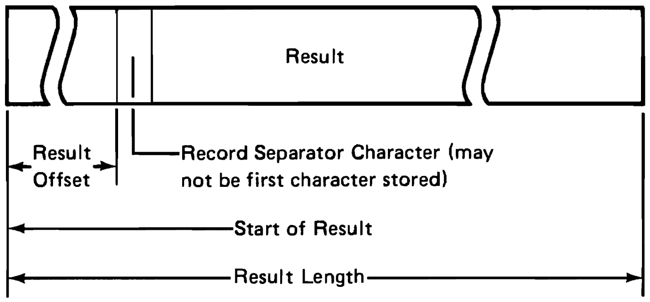
The *gap offset* specifies the offset (relative to the source offset) to the beginning of the next gap in the source. Gap offset is updated when the instruction is terminated. It is not used as input if the source data field length is specified as zero. It may be modified during execution of the instruction.

The *gap length* specifies the number of bytes (hex) of data between data fields in the source operand. This length is ignored if the data field length is specified with a value of zero. The gap length starts with a value of 0.

The *record separator character* field specifies the value that is to precede the converted form of each record in the result operand. This value also serves as a delimiter for the prior record when trailing blanks are truncated; the last record will not have this delimiter. The record separator character field can have any hexadecimal value. However, the Convert SNA to Characters instruction recognizes only values less than hex 40 as record separators.

This field is ignored if string processing is specified in the modifier.

Result Operand


Result
Result Offset
Record Separator Character (may not be first character stored)
Start of Result
Result Length

The *prime compression character* specifies the value to be used as the prime compression character. It can be any value. It is ignored if compression is not specified in the algorithm modifier.

The *unconverted source record bytes* contains a count of the residual, unconverted bytes in the current source record. This parameter is not used as input if record spanning is not specified in the algorithm modifier. The count may be set to zero during execution of the instruction.

Format: SS

BE	B_3	6	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The operation begins by accessing the bytes of the source operand at the location specified by the source offset. This location is assumed to be the start of a record. Gaps between data fields are ignored, causing the access of data bytes to occur as if the data fields were contiguous with one another.

Accessed bytes are considered to be a source record for the conversion. They are converted according to the following modes and optional functions and then stored in the result.

String Processing Mode

String processing occurs when bit 1 of the algorithm modifier is equal to zero. The bytes accessed in the source are converted, compressed, and then stored in the result.

Compression

The compression function is always performed in string processing mode. Compression reduces the length of duplicate character strings in the source data.

Compressed data is built by concatenating one or more compression strings together to describe the converted record. The bytes of the converted source data are checked in order to locate the:

- Prime compression character strings (two or more consecutive prime compression characters)
- Duplicate character strings (three or more duplicate nonprime characters)
- Nonduplicate character strings occurring in the source.

The character strings encountered (prime, duplicate, and non-duplicate) are reflected in the compressed data by building one or more compression strings to describe them. Compression strings are comprised of a string control byte (SCB), followed by prescribed bytes of data related to the character string being described.

The SCB has the following format and bit definitions:

Bits **Meaning**

0-1 Control

00 = n nonduplicate characters are between this SCB and the next one. n is the value in the count field; possible values are 1-63 (decimal).

01 = Reserved

10 = This SCB represents n deleted prime compression characters. n is the value in the count field; possible values are 2-63 (decimal). The next byte is the next SCB.

11 = This SCB represents n deleted duplicate characters. n is the value of the count field; possible values are 3-63 (decimal). The next byte contains a specimen of the deleted characters. The byte following the specimen character contains the next SCB.

2-7 Count

The value n (in binary) in this field represents the count of the number of characters that have been deleted for a prime compression character string, a duplicate character string, or the number of characters to the next SCB for a nonduplicate character string. A count value of 0 cannot be produced.

In string processing mode:

- Compression is performed on the entire source operand on a string basis. The fields in the controls operand related to record processing are ignored.

- If the compressed data cannot be completely contained in the receiver, the instruction ends with a receiver overrun condition code.
  - As much of the compressed data as will fit is placed into the receiver, and the controls operand is updated to describe how much of the source data was successfully converted into the receiver.
  - The last compression entry placed into the receiver may be adjusted, if necessary, to a length which fits in the receiver. This length adjustment applies only to compression entries for non-duplicate strings.
  - Compression entries for duplicate strings are only placed in the receiver if they fit with no adjustment. By doing this, no more than 1 byte of unused space will remain in the receiver; its value is unpredictable.

- If the compressed data can be completely contained in the receiver, the instruction ends with a source exhausted condition code. The compressed data is placed into the receiver, and the controls operand is updated to indicate that all of the source data was successfully converted into the receiver.

Record Processing Mode

Record processing occurs when bit 1 of the algorithm modifier is equal to 1.

The source offset locates either the start of a full or the start of a partial record. If record spanning is not specified, source offset locates a full record. If record spanning is specified, the source offset is assumed to locate a point at which processing of a partially converted record is to be resumed (this could actually be the start of a full record). The unconverted source record bytes value (which could be 0) gives the length of the remaining portion of the source record to be converted. The conversion process is started by completing the conversion of the current source record (if such is the case), before processing the next full source record.

When the conversion process for a record is complete (including trailing blank truncation, data transparency conversion (if specified), and compression (if specified)) and a receiver overrun has not occurred, the process is started for the next record.

A check for end of source is made at the start of conversion for each record. If the source does not contain a full record, the source exhausted condition is recognized and the instruction is terminated. Conversion of a partial source record is not performed.

Trailing Blank Truncation

The trailing blank truncation function is always performed in record processing mode. This function can be performed with, or without, the optional transparency conversion and compression functions.

A truncated record is built by logically appending the record data to the record separator (a value specified in the controls operand) and removing all blank characters after the last nonblank character. If a record has no trailing blanks, then no actual truncation takes place. A null record (a record consisting entirely of blanks), will be converted as just the record separator character with no other data following it. The truncated records, then, consist of the record separator character followed by the full record data, the truncated record data, or no data.

If the truncated record cannot be completely contained in the receiver, the instruction ends with a receiver overrun condition code. If record spanning is specified, as much of the truncated record as will fit is stored into the receiver, and the controls operand is updated to describe how much of the source record was successfully converted. If record spanning is not specified, the controls operand is updated to describe only the last fully converted record; the values of the remaining bytes in the receiver are unpredictable.

Data Transparency Conversion

The data transparency conversion function is performed in record processing mode only. It is optional, not mandatory; compression may also be done, but is not required.

This function makes the data in a record transparent to the Convert SNA to Character instruction in the area of its scanning for record separator values.

A transparent record is built by placing 2 bytes of transparency control information after the record separator, but before the actual data. The first byte has a fixed value of hex 35 and is referred to as the TRN (transparency) control character. The second byte is a 1-byte hexadecimal count (with allowable values of 1-255 decimal) of the number of transparent data bytes that follow and is referred to as the TRN count. This count contains the length of the data (before compression) and does not include these TRN control information bytes, the record separator, or trailing blanks that have been truncated.

For a null record, no TRN control information is placed after the record separator as there is no record data to be made transparent.

If the transparent record cannot be completely contained in the receiver, the instruction ends with a receiver overrun condition code.

- If record spanning is specified, as much of the transparent record as will fit is placed in the receiver and the controls operand is updated to describe how much of the source record was converted. The TRN count is adjusted to describe the length of the successfully converted data; thus, the transparent data for the record is not spanned out of the receiver. The remaining bytes of the transparent record, if any, will be processed as a partial source record on the next invocation of the instruction and will be preceded by the appropriate TRN control information.

  For the special case where only 1 to 3 bytes are available at the end of the receiver (not enough room for the record separator, the transparency control, and a byte of data), just the record separator is placed in the receiver for the record being converted. This can cause up to 2 bytes of unused space at the end of the receiver; the values of these unused bytes are unpredictable.

- If record spanning is not specified, the controls operand is updated to describe only the last fully converted record in the receiver. The values of the remaining bytes in the receiver are unpredictable.

Compression

The compression function is performed on the converted form of the current source record, including the record separator character; this can be a truncated record or a transparent truncated record. TRN control information bytes are always treated as part of a non-duplicate compression entry to provide for length adjustment of the TRN count, if necessary.

If the compressed record cannot be completely contained in the receiver, the instruction ends with a receiver overrun condition code.

When record spanning is specified:

- As much of the compressed record as will fit is placed into the receiver and the controls operand is updated to describe how much of the source record was successfully converted into the receiver.

- The last compression entry placed into the receiver may be adjusted, if necessary, to a length that fits in the receiver. This applies only to nonduplicate strings.

- Compression entries for duplicate strings are placed in the receiver only if they fit with no adjustment.

- For the special case where data transparency conversion is specified, the transparent data being described is not spanned out of the receiver; the TRN count is adjusted to describe only the amount of data successfully placed into the receiver.

- For the special case where only 2-5 bytes are available at the end of the receiver, there may not be enough room for the compression entry for the nonduplicate string containing the record separator, the TRN control, and up to a 2-byte compression entry for some of the transparent data. In this case, the non-duplicate compression entry is adjusted to describe only the record separator. By doing this, no more than 3 bytes will remain in the receiver; the values of these unused bytes are unpredictable. Unconverted source record bytes, if any, will be processed as a partial source record on the next invocation of the instruction and will be preceded by the appropriate TRN control information when performing transparency conversion.

When record spanning is not specified, the controls operand is updated to describe only the last full converted record in the receiver; the values of the remaining unused bytes in the receiver are unpredictable.

Instruction Termination

The CVTCS instruction terminates when:

- The end of the source operand is reached (see note). This results in a source exhausted condition code.

- The end of the receiver is reached (see note). This results in a receiver overrun condition code.

Note: For the special case of a tie between the source exhausted and receiver overrun conditions, the source exhausted condition is recognized first because when source exhausted is the resultant condition, the receiver may also be full. In this case, the offset into the receiver operand may contain a value equal to the length specified for the receiver, which would cause an exception to be detected on the next invocation of the instruction. The processing performed for the source exhausted condition should provide for this case if the instruction is to be invoked multiple times with the same controls operand value. When the receiver overrun condition is the resultant condition, the source will always contain data remaining to be converted.

At the completion of the instruction execution, the source and receiver offset parameters are updated to point to the next bytes to be operated on in the source and receiver, respectively. The source offset may point to the start of a gap, but will never point within a gap.

If record spanning is specified, the unconverted source record bytes parameter is updated to specify the number of remaining unconverted source record bytes.

If the source data field length is not 0, the gap offset parameter is updated to point to the next gap, relative to the source offset parameter just updated.

Any form of overlap between the operands of this instruction yields unpredictable results.

Programming Notes

If the source operand does not end on a record boundary (meaning the last record is spanned out of the source), this instruction performs conversion only up to the start of that partial record. The user of this instruction must move this partial record to combine it with the rest of the record in the source operand to provide for subsequent correct processing. If full records are provided, the instruction performs its conversion out to the end of the source operand and no special processing is required.

At the completion of this instruction, any bytes in the receiver beyond the location pointed to by the receiver offset are unpredictable.

Although any value of record separator is allowed, use of hex 40 can possibly cause some unanticipated results. With no transparency, and a completely blank record, use of a hex 40 record separator will result in no output being stored for that record. This is because the record separator is included with the blanks and discarded as part of blank truncation.

This instruction is interruptible. If interrupted, information required to continue is stored in the controls operand and the instruction address register will point to the instruction so that processing will continue after the interrupt.

Overflow and Sign Code: Not applicable.

Condition Codes

0	Source exhausted
1	Receiver overrun
2	--
3	--

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions

- Address translation
- Addressing
- Effective address overflow
- Specification

CVTCS Example

Op	B_3	E	B_1	D_1	B_2	D_2
BE	7	6	8	AA3	9	541

0 Bits 8 12 16 20 32 36 47

Assembler: CVTCS $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: BE76 8AA3 9541

$B_1(8)$:	0021	A123	0000	(Base register for result)
$B_2(9)$:	0022	0015	0000	(Base register for source)
$B_3(7)$:	0100	0303	F105	(Address of control operand)
R(14):			0005	(Length of result)
R(15):			0009	(Length of source)

Storage — Before

	0/8	2/A	4/C	6/E
0021 A123 0AA3		xx	xxxx	xxxx
0022 0015 0541	AC	5454	ACF1	5454
	F1F1			
0100 0303 F105			00	0000
	00D0	0300	0200	0100
	0211	AC02		

Storage — After

	0/8	2/A	4/C	6/E
0021 A123 0AA3		82	0111	C3F1
0022 0015 0541	AC	5454	ACF1	5454
	F1F1			
0100 0303 F105			00	0500
	09D0	0300	0200	0000
	0211	AC00		

	Before	After
Condition Code:	x	0

CONVERT CHARACTERS TO MULTI-LEAVING REMOTE JOB ENTRY (CVTCM)

Instruction Description

The operation converts the data at the second operand location from character to MRJE (MULTI-LEAVING Remote Job Entry) format. The conversion is controlled by information whose address is in the base register specified in the third operand. The result is placed in the first operand.

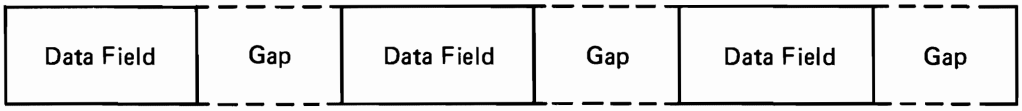
The operands are as follows:

Operand	Description
1	The base and displacement for the starting address of the result string that is to contain the converted data.
2	The base and displacement for the starting address of the source string that contains the data to be converted.
3	The base register that contains the address of the control information for the conversion operation to be performed.
4	Halfword register 14 specifies the length of the first operand (result string). A length of zero causes a specification exception.
5	Halfword register 15 specifies the length of the second operand (source string). A length of zero causes a specification exception.

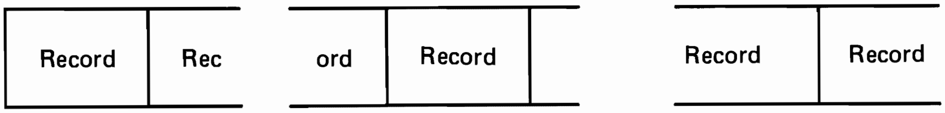
The source operand (2) contains one or more fixed-length data fields that may be separated by fixed-length gaps of characters to be ignored during the conversion. The source operand is described by the controls operand (3), which also specifies the number of bytes of data from the source to be processed to produce a converted record in the result string. The source record length does not need to be the same as the source data field length.

The following diagram explains this structure for the source operand:

Actual Source Operand Bytes



Data Processed as Source Records



The controls operand is a 13-byte string that specifies additional information to be used to control the conversion. The controls operand has the following format:

Result Offset	Source Offset	Mod-ifier	Source Record Length	Data Field Length	Gap Offset	Gap Length	Record Control Block
0	2	4	5	6	8	10	12

Bytes

Bytes	Description
0-1	Offset into the result operand
2-3	Offset into the source operand
4	Modifier
5	Source record length (>0)
6-7	Data field length (>0)
8-9	Offset to the next gap in the source operand
10-11	Gap length
12	Record control block value

Upon input to the instruction, the *result offset* and the *source offset* fields specify which bytes of the source field are processed and entered into the result field. The source and result offset fields are set to values that indicate how much of the conversion is complete when the instruction is interrupted or complete. An offset beyond the end of the related source or result operand causes a specification exception.

The *modifier* has the following valid values:

Value (Hex)	Description
00	Perform full compression.
01	Perform only truncation of trailing blanks.

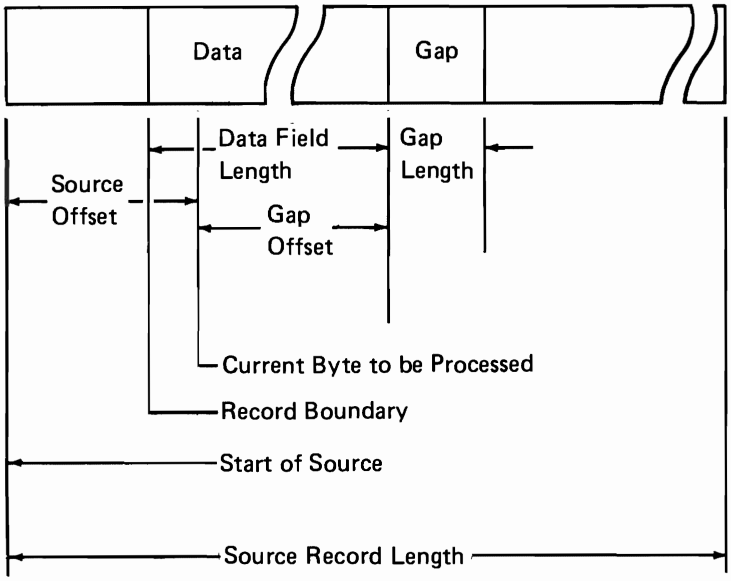
Note: An invalid modifier value causes a specification exception.

The *source record length* specifies the number of bytes to be processed to produce a converted record in the result operand. A source record length of zero results in a specification exception. Data fields in the source may be separated by gaps of characters. These gaps are to be ignored during conversion.

The *data field length* specifies the number of bytes in the source data fields. Specifying a data field length of zero indicates the source length is one data field; in this case, the gap length and gap offset are ignored.

The following diagrams illustrate the makeup of the source and result operands.

Source Operand

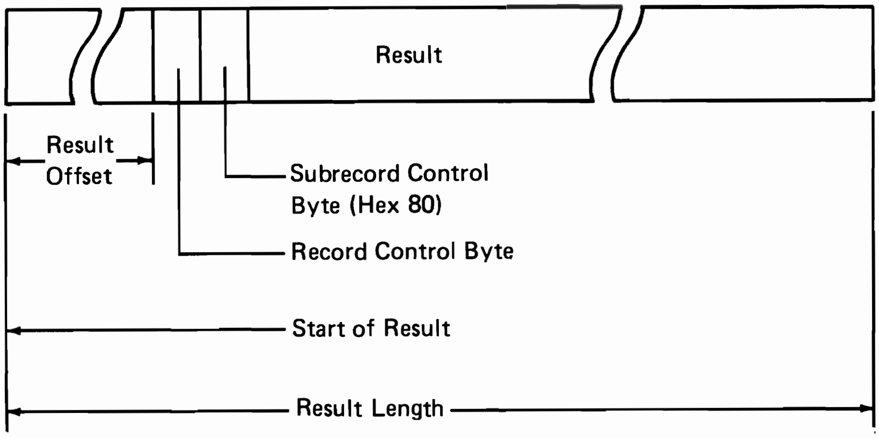


The *gap offset* specifies the offset to the next gap in the source. This offset is both input to and output from the instruction. The gap offset decreases as the source increases until the gap is reached. The gap offset then becomes the offset to the next gap.

The *gap length* specifies the number of bytes (hex) of data between data fields in the source operand. This length is ignored if the data field length is specified with a value of zero. The gap length starts with a value of 1.

The *record control block* field specifies the value that is to precede the converted form of each record in the result operand. The record control block field can have any hexadecimal value.

Result Operand



Format: SS

BE	B_3	4	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The operation begins by accessing the bytes of the source operand at the location specified by the source offset. This location is assumed to be the start of a record. Gaps between data fields are ignored, causing the access of data bytes to occur as if the data fields were contiguous with one another.

Accessed bytes are considered to be a source record for the conversion. They are converted into the result according to the following procedure.

The record control block value is put into the first byte of the result record. A subrecord control block value of hex 80 is put into the second byte of the result record.

If a modifier specifies full compression, then the bytes of the source record, as they are accessed in the source, are checked for:

- Blank character strings (2 or more consecutive blanks)
- Identical character strings (3 or more consecutive identical characters)
- Unidentical character strings

A blank character string occurring at the end of the record is treated as follows. If the record is not completely blank, then an end-of-record string control bytes (hex 00) is stored in the result. If the entire record is blank, then a string control byte indicating 1 blank (a nonrepeating character) followed by an end-of-record string control byte is in the result.

If the modifier specifies blank truncation, then the bytes of the source record are checked for a blank character string at the end of the source record. If one exists, it is treated as a string of trailing blanks. All characters prior to a string of trailing blanks in the record are treated as one string of unidentical characters.

The strings encountered–blank, identical, or unidentical–are related in the result of building one or more string control bytes to describe them. The format of the string control bytes is as follows:

End of Record	Com- press	Delete	Number of Characters (Binary)

0 1 2 3 Bits 8

Byte	Bit	Value (Binary)	
0-1	0	0	End of record; the end-of-record string control byte is hex 00.
		1	All other string control bytes.
	1	0	The string is compressed.
		1	The string is not compressed.
	2		If bit 1 = 0.
		0	Blanks have been deleted (hexadecimal 40s).
		1	Nonblank characters have been deleted. The next character in the data stream is the specimen character. If bit 1 = 1, this bit is part of the length field for length of uncompressed data.
	3-7	00010 11111	If bit 1 = 0, this is the number of characters that have been deleted. The value can be from 2 through 31.
	2-7	000001- 111111	If bit 1 = 1, this is the number of characters before the next string control byte (no compression). The uncompressed (unidentical) bytes follow the string control bytes in the data stream. The value can be from 1 through 63.

When the end-of-source record is encountered, an end-of-record string control byte (hex 00) is built into the result operand. Trailing blanks in a record, including a record of all blanks, are represented in the result by an end-of-record character. Additionally, the values in the controls operand for the result offset, and source ofset, and gap offset are updated. These values describe the offsets for the next record to be converted, allowing for the interruption of the instruction on a record boundary.

If the end-of-source record is not encountered, the operation continues as described at the beginning of the *Operation* section.

If the end of source is encountered while processing a field, whether or not in conjunction with a record boundary, the instruction ends with a condition code of zero (source exhausted). See *Programming Note.*

If the converted record cannot be completely contained in the result, the instruction ends with a condition code of 1 (result overrun). See *Programming Note.*

Programming Note: The source offset locates the byte following the last source record for which conversion was completed. The gap offset indicates the offset to the next gap. The gap offset has no meaning and is not set when the data field length is zero. The result offset locates the byte following the last fully converted record in the result. The contents of the remaining bytes in the result after the last converted record are unpredictable.

Any form of overlap between the operands yields unpredictable results in the result operand.

Overflow and Sign Code: Not applicable.

Condition Code:

0 Source used up
1 Result overrun
2 - -
3 - -

Carry and Boundary Requirements: Not applicable.

Program Exceptions:

- Address translation
- Addressing
- Effective address translation
- Specification

Programming Note: If the data field length is zero, the gap length and gap offset are ignored.

CVTCM Example

Op	B_3	E	B_1	D_1	B_2	D_2
BE	5	4	4	BC5	3	582
0 Bits	8	12	16	20	32	36 47

Assembler: CVTCM $D_1(B_1)$,$D_2(B_2)$,B_3

Machine: BE54 4BC5 3582

B_1 (4):	0001 5678 0000	(Base register for result)
B_2 (3):	0001 1234 0000	(Base register for source)
B_3 (5):	0001 036A 0620	(Address of control operand)
R (14):	0020	(Length of result)
R (15):	0020	(Length of source)

Storage—Before

```
                 0/8   2/A   4/C   6/E
0001 036A 0620   0000  0000  0010  0000
                 XXXX  XXXX  F0XX  XXXX

0001 1234 0580   XXXX  1111  1111  1111
                 E3C5  E2E3  E3C5  E2E3
                 E3E3  4040  2222  2222
                 2222  2222  4040  4040
                 4040  XXXX  XXXX  XXXX

0001 5678 0BC5                 XX  XXXX
                 XXXX  XXXX  XXXX  XXXX
                 XXXX  XXXX  XXXX  XXXX
                 XXXX  XXXX  XXXX  XXXX
                 XXXX  XXXX  XX
```

Storage—After

```
                 0/8   2/A   4/C   6/E
0001 036A 0620   0015  0020  0010  FFFF
                 XXXX  XXXX  F0XX  XXXX

0001 1234 0580   XXXX  1111  1111  1111
                 E3C5  E2E3  E3C5  E2E3
                 E3E3  4040  2222  2222
                 2222  2222  4040  4040
                 4040  XXXX  XXXX  XXXX

0001 5678 0BC5                 XX  XXXX
                 XXXX  F080  A611  C7E3
                 C5E2  E3E3  C5E2  A3E3
                 00F0  8082  A822  00XX
                 XXXX  XXXX  XX
```

	Before	After
Condition Code:	X	0

CONVERT DECIMAL FORM TO LONG FLOAT (CVDFLF)

Instruction Description

The decimal form of a floating-point value specified by a decimal exponent (operand 2) and a decimal significand (operand 3) is converted to binary floating-point format, and the result is placed in the binary floating-point field specified by the first operand.

Format: SS

CE		D	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first operand specifies a binary floating-point field that occupies 8 bytes, and has the long floating-point field format.

The second operand specifies the decimal exponent that occupies from 1 through 16 bytes as specified by the operand 4 value. This operand has the packed fixed-point decimal format.

The third operand, base register hex E, specifies the address of the decimal significand. This operand occupies up to 16 bytes of storage as specified by the operand 4 value and has the packed fixed-point decimal format.

The fourth operand, halfword register hex F, contains the digit lengths of the second and third operands. The total number of digits for the exponent (operand 2) is contained as a value between 1 and 31 in the leftmost byte of the halfword register. The total number of digits for the significand (operand 3) is contained as a value between 1 and 31 in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges, or a specification exception is signaled. The length of operands 2 and 3 (in bytes) is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The specified number of digits are considered right adjusted in their respective fields. An even value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the corresponding operand value.

The exponent and significand contain a decimal form of a floating-point number. The value of this number is:

Value = M * (10**E)
where:
M = the value of the significand operand
E = the value of the exponent operand
** denotes exponentiation
* denotes multiplication

The exponent is assumed to contain a decimal integer value. This signed integer value specifies a power of 10 that gives the floating-point value its magnitude. It has an assumed decimal point immediately to the right of its rightmost digit position.

The significand is assumed to contain a decimal value with a leading integer digit in its leftmost digit position and fractional digits in the digit positions to the right of the integer digit. The signed decimal value specifies the decimal digits that give the floating-point value its precision. The significand has an assumed decimal point immediately to the right of its leftmost digit position.

The decimal form floating-point value specified by the exponent and significand operands is converted to binary floating-point format as if to infinite precision. However, the precision provided for in floating-point fields is not as great as the precision that can be provided for by decimal fields. Long floating-point provides for unique representation of a maximum of 15 significant decimal digits of precision. The significant digits of the significand start with the leftmost nonzero decimal digit and continue to the right out to the end of the significand value. Significant digits beyond 15 for a long float receiver may not be preserved in the result and only serve to provide for uniqueness of the conversion as well as for proper rounding.

The result of this conversion is then normalized and rounded (according to the current float rounding mode) to the significand length of the operand 1 field.

The converted, normalized, and rounded result is then assigned to operand 1 in the long floating-point format. The result is subject to the normal floating-point overflow and underflow exception detection performed on assignment.

When floating-point overflow or underflow is detected and unmasked, the instruction operation is suppressed. This action occurs because all overflowed and underflowed values cannot be represented in the result field format even when employing the modified biased exponent representation.

Conversion of a zero value significand operand results in a zero value of the same sign being assigned to operand 1.

Operands 2 and 3 are checked for valid decimal sign and digit codes. The data exception is signaled if any invalid values are encountered, and the operation is suppressed. If an even number of digits is specified for either the exponent or the significand operands, the leftmost digit position of the packed operand field is not checked and is not used as part of the decimal value.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands may overlap.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow
- Floating-point inexact result
- Floating-point overflow
- Floating-point underflow
- Specification

CVDFLF Example

Op		E	B_1	D_2	B_1	D_2
CE		D	2	040	2	064

0 Bits 8 12 16 20 32 36 47

Assembler: CVDFLF $D_1(B_1)$, $D_2(B_2)$

Machine: CE0D 2040 2064

B_1(2) and B_2(2): 800D 0C00 0000

B_3(E): 800D 0C00 0300

R(F): 0712

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0040	xxxx	xxxx	xxxx	xxxx
800D 0C00 0064			0000	002F
800D 0C00 0300	0123	0000	0000	0000
	000F			

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0040	405E	C000	0000	0000
800D 0C00 0064			0000	002F
800D 0C00 0300	0123	0000	0000	0000
	000F			

CONVERT DECIMAL FORM TO SHORT FLOAT (CVDFSF)

Instruction Description

The decimal form of a floating-point value specified by a decimal exponent (operand 2) and a decimal significand (operand 3) is converted to binary floating-point format, and the result is placed in the binary floating-point field specified by the first operand.

Format: SS

AE		D	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first operand specifies a binary floating-point field that occupies 4 bytes, and has the short floating-point field format.

The second operand specifies the decimal exponent that occupies from 1 through 16 bytes as specified by the operand 4 value. This operand has the packed fixed-point decimal format.

The third operand, base register hex E, specifies the address of the decimal significand. This operand occupies up to 16 bytes of storage as specified by the operand 4 value and has the packed fixed-point decimal format.

The fourth operand, halfword register hex F, contains the digit lengths of the second and third operands. The total number of digits for the exponent (operand 2) is contained as a value between 1 and 31 in the leftmost byte of the halfword register. The total number of digits for the significand (operand 3) is contained as a value between 1 and 31 in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges or a specification exception is signaled. The length of operands 2 and 3 (in bytes), is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The specified number of digits are considered right adjusted in their respective fields. An even value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the corresponding operand value.

The exponent and significand contain a decimal form of a floating-point number. The value of this number is:

Value = M * (10**E)
where:
M = the value of the significand
E = the value of the exponent operand
** denotes exponentation
* denotes multiplication

The exponent is assumed to contain a decimal integer value. This signed integer value specifies a power of 10 that gives the floating-point value its magnitude. It has an assumed decimal point immediately to the right of its rightmost digit position.

The significand is assumed to contain a decimal value with a leading integer digit in its leftmost digit position and fractional digits in the digit positions to the right of the integer digit. The signed decimal value specifies the decimal digits that give the floating-point value its precision. The significand has an assumed decimal point immedately to the right of its leftmost digit position.

The decimal form floating-point value specified by the exponent and significand operands is converted to binary floating-point format as if to infinite precision. However, the precision provided for in floating-point fields is not as great as the precision that can be provided for by decimal fields. Short floating-point provides for unique representation of a maximum of 7 significant decimal digits of precision. The significant digits of the significand start with the leftmost nonzero decimal digit and continue to the right out to the end of the significand value. Significant digits beyond 7 for a short floating-point receiver may not be preserved in the result and only serve to provide for uniqueness of the conversion as well as for proper rounding.

The result of this conversion is then normalized and rounded (according to the current float rounding mode) to the significand length of the operand 1 field.

The converted, normalized, and rounded result is then assigned to operand 1 in the short floating-point format. The result is subject to the normal floating-point overflow and underflow exception detection performed on assignment.

When floating-point overflow or underflow is detected and unmasked, the instruction operation is suppressed. This action occurs because all overflowed and underflowed values cannot be represented in the result field format even when employing the modified biased exponent representation.

Conversion of a zero value significand operand results in a zero value of the same sign being assigned to operand 1.

Operands 2 and 3 are checked for valid decimal sign and digit codes. The data exception is signaled if any invalid values are encountered, and the operation is suppressed. If an even number of digits is specified for either the exponent or the significand operands, the leftmost digit position of the packed operand field is not checked and is not used as part of the decimal value.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands may overlap.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow
- Floating-point inexact result
- Floating-point overflow
- Floating-point underflow
- Specification

CVDFSF Example

Op		E	B_1	D_1	B_2	D_2
AE		D	2	058	2	064

0 Bits 8 12 16 20 32 36 47

Assembler: CVDFSF $D_1(B_1)$, $D_2(B_2)$

Machine: AE0D 2058 2064

B_1(2) and B_2(2): 800D 0C00 0000

B(E): 800D 0C00 0300

R(F): 0709

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0058	xxxx	xxxx		
800D 0C00 0064			0000	000F
800D 0C00 0300	3480	4687	5F	

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0058	405E	C000		
800D 0C00 0064			0000	000F
800D 0C00 0300	3480	4687	5F	

CONVERT LONG FLOAT TO BINARY (CVLFB)

Instruction Description

The value stored at the second operand location is converted from floating-point to binary and placed in the first operand.

Format: SS

CE	M	8	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Operand 1 has a signed binary format and is either 2, 4, or 8 bytes in length. The length of the operand is determined by an options mask.

Operand 2 is 8 bytes long, and has a long floating-point format. The data for this operand must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Operand 3 is a 4-bit options mask (bits 8 through 11) that controls the conversion operation. The format of the options mask is:

Bits	Meaning
8	Mode of rounding to be performed. 0 = Round using current floating-point rounding mode in effect. 1 = Round using decimal round algorithm.
9-11	Length of binary result (operand 1). 001 = 2 bytes. 011 = 4 bytes. 111 = 8 bytes. All other values are invalid.

The floating-point value of the second operand is converted to a fixed-point binary integer format. If necessary, the floating-point value is rounded to an integer value.

The rounding mode is specified by the options mask (bit 8 of operand 3). If floating-point rounding is specified, rounding is performed according to the current floating-point rounding mode in effect. If decimal rounding mode is specified, the current floating-point rounding mode is overridden, and the decimal round algorithm is performed. In this case, a value of 1/2 (a 1 bit) is added to the leftmost bit position of the fractional portion of the floating-point value, and that bit and those bits to the right are truncated from the resulting value.

The value assigned to operand 1 is formed as a right-aligned, binary integer value with an assumed binary point immediately to the right of its rightmost digit.

If the rounded integer portion of the floating-point value is 0, the first operand value is set to 0, and the sign is set positive, regardless of the sign of the second operand.

An invalid floating-point conversion exception is signaled for any number outside the range of integer values that can be contained in operand 1 (this includes NaNs and infinities).

The result obtained from overlapping operands is unpredictable.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must be on a fullword boundary; otherwise, a specification exception occurs.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Invalid floating-point conversion
- Specification

Programming Note: The following is a summary of the results for various combinations of operands.

Receiver	Source
0	±0
-B	-R
+B	+R
IFPC	±INF
IFPC	MNAN
IFPC	UNAN

Legend:

R	=	Real nonzero floating-point number
B	=	A nonzero binary number
MNAN	=	A masked NAN
UNAN	=	An unmasked NAN
INF	=	Infinity
IFPC	=	An invalid floating-point conversion exception.

The assignment of a real number (R) as the value of the binary field (B) is only successful if R is a value that can be contained within the value range of the binary field; otherwise, an invalid floating-point conversion may result.

CVLFB Example

Op	M_3	E	B_1	D_1	B_2	D_2
CE	7	8	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: CVLFB $D_1(B_1)$, $D_2(L_2B_2)$

Machine: CE78 4050 4060

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	416F	E000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	00FF	0000	0000	0000
0010 0200 0060	416F	E000	0000	0000

Condition Code: Not changed.

CONVERT LONG FLOAT TO DECIMAL FORM (CVLFDF)

Instruction Description

The binary floating-point value specified by operand 5 is converted to a decimal form of a floating-point value (a decimal exponent and a decimal significand) and placed into operand 1 (exponent) and operand 2 (significand) locations.

Format: SS

CE	M_3	C	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first operand specifies the decimal exponent and occupies from 3 through 16 bytes as specified by the operand 4 value. This operand is formed using the packed fixed-point decimal format.

The second operand specifies the decimal significand, and occupies a maximum of 16 bytes as specified by the operand 4 value. This operand is formed using the packed fixed-point decimal format.

The third operand, bits 8 through 11, specifies an options mask to control the conversion operation.

Bits	Meaning
8	Mode of rounding to be performed. 0 = Round using current float rounding mode in effect. 1 = Round using decimal round algorithm.
9-11	Reserved.

The fourth operand, halfword register hex F, contains the digit lengths of the first and second operands. The total number of digits for operand 1 is specified as a value between 5 and 31 in the leftmost byte of the halfword register. The total number of digits for operand 2 is specified as a value between 1 and 31 in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges or a specification exception is signaled. The length of operands 1 and 2 (in bytes) is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The number of digits specified are considered right adjusted in their respective fields. An even-value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the corresponding operand value.

The fifth operand, base register hex E, specifies the address of the binary floating-point number. The number occupies 8 bytes, and has the long floating-point field format.

The exponent (operand 1) and significand (operand 2) contain a decimal form of a floating-point number. The value of this number is:

Value = M * (10**E)
where:
M = the value of the decimal significand operand
E = the value of the exponent operand
** denotes exponentiation
* denotes multiplication

The exponent is formed as a decimal integer value. The exponent, which gives the floating-point value its magnitude, contains a signed integer value that specifies a power of 10. The exponent has an assumed decimal point immediately to the right of its rightmost digit position.

The significand is formed as a decimal value with a single integer digit in its leftmost digit position and fractional digits in the digit positions to the right of the integer digit. The significand contains a signed decimal value that specifies decimal digits, to give the floating-point value its precision. The significand has an assumed decimal point immediately to the right of its leftmost digit position.

The binary floating-point source is converted to a decimal form floating-point value as if to infinite precision. However, the precision provided for by floating-point fields is not as great as the precision provided for by decimal fields. Long floating-point provides for unique representation of a maximum of 15 significant decimal digits of precision. The significant digits of the significand start with the leftmost nonzero decimal digit and continue to the right out to the end of the significand value. The converted significand value is formed as a normalized value, the significant digits are left adjusted in the converted value, and the converted exponent is set accordingly. Significand digits beyond the leftmost 15 provide for uniqueness of the conversion and should be considered only as precise as the floating-point calculations that produced the source value.

The converted significand value is adjusted to the precision of the significand operand, if necessary, by using the rounding algorithm specified in the options mask operand. If the rounding algorithm causes a carry out of the leading integer digit position, the converted rounded significand value is shifted right one digit position and the converted exponent incremented by 1 to realign the significand back to having one leading integer digit. If floating-point rounding is selected, rounding is performed according to the current floating-point rounding mode in effect. If decimal rounding is selected, the current floating-point rounding mode is overridden and the decimal round algorithm is performed. In this case, a value of 5 is added to the converted significand in the leftmost digit position not provided for in operand 2, and that digit, and those digits to the right of it, are truncated from the resulting significand value.

The result of this conversion is then assigned to the exponent and significand operands. For an exponent or significand operand with an even number of digits, the leftmost digit position of the packed field in the operand is set to binary 0.

If the binary floating-point number being converted contains a value of 0, the exponent operand is set to positive 0, and the significand operand is set to 0 with the sign of the binary floating-point number. A positive 0 is set with the preferred positive sign of hex F. A negative 0 is set with the preferred negative sign of hex D.

A decimal overflow exception cannot occur on the assignment of the exponent or significand values.

When the binary floating-point number being converted contains a denormalized floating-point value, the first and second operand values are set with the correctly converted and rounded values; no exception is signaled.

When an infinity or NaN value is encountered in the second operand, the invalid floating-point conversion exception is signaled and the instruction operation is suppressed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlap between operands 1 and 2 is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid floating-point conversion
- Specification

Programming Note: The following is a summary of the results for various combinations of operands where:

Receivers	**Source**
-0*10**+0	-0
+0*10**+0	+0
-M*10**+E	-R<-1
+M*10**+E	+R>1
-M*10**-E	-R>-1
+M*10**-E	+R<1
IFPC	±INF
IFPC	MNaN
IFPC	UNaN

Legend:

R	=	a real nonzero, non-denormal floating-point number
E	=	the exponent, a nonzero decimal number
M	=	the significand, a nonzero decimal
MNaN	=	a masked NaN
UNaN	=	an unmasked NaN
INF	=	infinity
IFPC	=	invalid floating-point conversion exception
**		denotes exponentiation
*		denotes multiplication

CVLFDF Example

OP	M_3	E	B_1	D_1	B_2	D_2
CE	0	C	2	064	2	070

0 Bits 8 12 16 20 32 36 47

Assembler: CVLFDF $D_1(B_1)$, $D_2(B_2)$, M_3

Machine: CE0C 2064 2070

B_1(2) and B_2(2): 800D 0C00 0000

B(E): 800D 0C00 0300

R(F): 0712

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0064			xxxx	xxxx
800D 0C00 0070	xxxx	xxxx	xxxx	xxxx
	xxxx			
800D 0C00 0300	405E	C000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0064			0000	002F
800D 0C00 0070	0123	0000	0000	0000
	000F			
800D 0C00 0300	405E	C000	0000	0000

CONVERT LONG FLOAT TO PACKED DECIMAL (CVLFPD)

Instruction Description

The value of the second operand is converted from floating-point to packed decimal, and the result is placed in the first operand location.

Format: SS

CE	M_3	A	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The first operand occupies up to 16 bytes of storage, as specified by the operand 4 value, and is formed according to the packed fixed-point decimal format.

The second operand occupies 8 bytes and has the long floating-point field format.

The third operand, bits 8 through 11, specifies an options mask to control the conversion operation.

Bits	Meaning
8	Mode of rounding to be performed. 0 = Round using current float rounding mode in effect. 1 = Round using decimal round algorithm.
9-11	Reserved.

The fourth operand, halfword register hex F contains the total and fractional digit count information for the number of decimal digits contained in the first operand. The total number of digits for operand 1 is contained, as a value between 1 and 31, in the leftmost byte of the halfword register. The number of fractional digits for operand 1 is contained as a value between 0 and 31, in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges or a specification exception is signaled. The number of integer digits in operand 1 is determined by subtracting the fractional digit count from the total digit count. The length of operand 1, in bytes, is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The number of digits specified are considered right adjusted in the operand 1 field. An even-value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the operand value.

The floating-point value is converted to a fixed-point packed decimal number as if to infinite precision. However, the precision provided for in floating-point fields i ıot as great as that which can be provided for by decimal fields. Long floating-point provides for unique representation of a maximum of 15 significant decimal digits of precision. The leftmost nonzero digit of the converted packed decimal number is considered the start of the significant digits of the number. Significant digits produced in the first operand beyond the first 15 for long floating-point serve to provide for uniqueness of conversion and should be considered only as precise as the calculations that produced the floating-point number.

The result of this conversion is then rounded, if necessary, to match the fractional precision of the operand 1 field.. The rounding algorithm performed is controlled by the third operand mask value. If floating-point rounding is selected, rounding is performed according to the current floating-point rounding mode in effect. If decimal rounding is selected, the current floating-point rounding mode is overridden and the decimal round algorithm is performed. In this case, a value of 5 is added to the converted number in the leftmost digit position not provided for in operand 1, and that digit, and those to the right of it, are truncated from the resulting sum.

The converted and rounded result is then assigned to operand 1 in the fixed-point packed decimal format for the number of digits specified by the total digit count for operand 1. If an even number of digits was specified, the leftmost digit position of the packed operand 1 field is set to binary 0.

If the converted and rounded result is 0, the first operand value is set to 0 and the sign is set positive, regardless of the sign of the second operand.

When a denormalized floating-point value is converted from the source operand, the first operand is set with the correctly rounded value, and no exception is signaled.

When any nonzero integer digits are truncated on the left in assigning the converted and rounded result to operand 1, or when an infinity value or a NaN value is encountered in the second operand, the invalid floating-point conversion exception is signaled and the instruction operation is suppressed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlapping operands is unpredicatable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid floating-point conversion
- Specification

Programming Note: The following is a summary of the results for various combinations of operands.

Receiver	**Source**
+0	±R0
-D	-R
+D	+R
IFPC	±INF
IFPC	MNaN
IFPC	UNaN

Legend:

R	=	a real nonzero value converted and rounded form of the source floating-point number
R0	=	a real zero value converted and rounded form of the source floating-point number
D	=	a nonzero decimal number
MNaN	=	a masked NaN
UNaN	=	an unmasked NaN
INF	=	infinity
IFPC	=	invalid floating-point conversion exception

The assignment of a real number, R, as the value of the decimal field, D, is only successful if R is a value that can be contained within the value range of the decimal field; otherwise, an invalid floating-point conversion may result.

CVLFPD Example

Op	M_3	E	B_1	D_1	B_2	D_2
CE	0	A	2	068	2	050

0 Bits 8 12 16 20 32 36 47

Assembler: CVLFPD $D_1(B_1)$, $D_2(B_2)$, M_3

Machine: CE0A 2068 2050

B_1(2) and B_2(2): 800D 0C00 0000

R(F): 0703

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0068	xxxx	xxxx		
800D 0C00 0050	405E	C000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0068	0123	000F		
800D 0C00 0050	405E	C000	0000	0000

CONVERT LONG TO SHORT FLOAT (CVLSF)

Instruction Description

The value of the second operand is converted from the long floating-point format to the short floating-point format, and the result is placed in the first operand location.

Format: SS

CE		7	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first operand occupies 4 bytes in storage and is formed using the short floating-point field format.

The second operand occupies 8 bytes in storage and has the long floating-point field format.

When the second operand contains a normalized nonzero floating-point value, the significand value from the second operand is rounded (according to the current rounding mode) to the short floating-point format significand length. The biased exponent value of the second operand is adjusted to the correct biased exponent value for the short floating-point format. This converted floating-point value is then assigned to the first operand according to the short floating-point field format. This operation is subject to the detection of the floating-point overflow and underflow conditions.

When the second operand contains a value of 0, the first operand is assigned a zero value of the same sign.

When the second operand contains an infinity floating-point value or a masked NaN value, the exponent and significand values are truncated on the right to the length of the short format prior to their assignment into the first operand. If the truncation of a masked NaN results in a fraction value of 0, the system default masked NaN value is assigned to the first operand.

When the second operand contains an unmasked NaN value, the floating-point invalid operand condition is detected. For the case where an unmasked NaN value is encountered and the floating-point invalid operand exception is masked, the first operand is assigned a masked NaN value with the fraction value from the original unmasked NaN truncated on the right to the short format fraction length. If the truncation of the unmasked NaN results in a fraction value of 0, the system default masked NaN value is assigned to the first operand.

If the second operand contains a denormalized floating-point number, the floating-point underflow condition is detected.

If the floating-point underflow condition is detected and masked, the result is assigned a value as defined by the floating-point underflow exception in Chapter 6. If this condition is detected and unmasked, the floating-point underflow condition is signaled. However, the operation is suppressed, and no result is stored. This action is taken because the underflowed value cannot be represented in the short format result field, even when employing the modified biased exponent representation.

In addition to the previous exception conditions of floating-point overflow and floating-point underflow, the floating-point inexact result and floating-point invalid operand conditions can be detected as a result of the execution of this instruction. Refer to Chapter 6 for a detailed description of these conditions and the instruction status when one of these conditions is detected.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not changed.

Boundary Requirements: The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point overflow
- Floating-point underflow
- Floating-point inexact result
- Floating-point invalid operand
- Specification

CVLSF Example

Op		E	B_1	D_1	B_2	D_2
CE		7	2	058	2	050

0 Bits 8 12 16 20 32 36 47

Assembler: CVLSF $D_1(B_1)$, $D_2(B_2)$

Machine: CE07 2058 2050

$B_1(2)$ and $B_2(2)$: 800D 0C00 0000

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0058	xxxx	xxxx		
800D 0C00 0050	405E	C000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0058	42F6	0000		
800D 0C00 0050	405E	C000	0000	0000

CONVERT MULTI-LEAVING REMOTE JOB ENTRY TO CHARACTER (CVTMC)

Instruction Description

The operation converts the data at the second operand location from MRJE (MULTI-LEAVING Remote Job Entry) to character format. The conversion is controlled by information whose address is in the base register specified in the third operand. The result is placed in the first operand.

The operands are as follows:

Operand **Description**

1 The base and displacement for the starting address of the result string that is to contain the converted data.

2 The base and displacement for the starting address of the source string that contains the data to be converted.

3 The base register that contains the address of the control information for the conversion operation to be performed.

4 Halfword register 14 specifies the length of the first operand (result string). A length of zero causes a specification exception.

5 Halfword register 15 specifies the length of the second operand (source string). A length of zero causes a specification exception.

The controls operand is a 6-byte string that specifies additional information to control the conversion operation. The controls operand has the following format:

Result Offset	Source Offset	Mod-ifier	Result Record Length
0 Bytes	2	4	5

Bytes	Description
0-1	Result offset
2-3	Source offset
4	Modifier
5	Result record length

Upon input to the instruction, *result offset* and the *source offset* fields specify the offsets at which bytes of the source field are processed and entered into the result field. The source and result offset fields are set to values which indicate how much of the conversion is complete when the instruction is interrupted or complete. An offset beyond the end of the related source or result operand causes a specification exception.

The *modifier* has the following valid values:

Value (Hex)	Description
00	Do not move subrecord control blocks from the source into the result.
01	Move subrecord control blocks from the source into the result.

Note: An invalid modifier causes a specification exception.

The *result record length* specifies the record length to be used to convert source records into the result. A length of zero causes a specification exception.

Format: SS

BE	B_3	5	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The operation begins by accessing the bytes of the source operand at the location specified by the source offset. This location is assumed to be the start of a record. The source operand bytes are converted into the result according to the following procedure.

The first byte of the source record is considered a byte of a record control block to be ignored during conversion.

The second byte of the source record is considered to be a subrecord control block. If a modifier of hex 00 is specified, the subrecord control block is ignored. If a modifier of hex 01 is specified, the subrecord control block is copied into the result.

The following bytes of the source record make up the string control bytes. One or more string control bytes describe the strings to be built in the result record.

The format of the string control bytes is as follows:

Byte	Bit	Value (Binary)	Description
0-1	0	0	End of record; the end-of-record string control byte is hex 00.
		1	All other string control bytes.
	1	0	The string is compressed.
		1	The string is not compressed.
	2		When bit 1 = 0.
		0	Blanks (hex 40s) have been deleted.
		1	Nonblank characters have been deleted. The next character in the data stream is the specimen character. When bit 1 = 1, this bit is part of the length field for length of uncompressed data.
	3-7	00001-11111	If bit 1 = 0, this is the number of characters that have been deleted.
	2-7	000001-111111	If bit 1 = 1, this is the number of characters before the next string control block (no compression). The uncompressed (unidentical) bytes follow the string control bytes in the data stream.

Note: A length of zero in a string control byte results in a conversion exception.

Strings of blanks or nonblank identical characters described in the source record are repeated in the result operand the number of times indicated by the string control block count. Strings of nonidentical characters described in the source record are moved into the result operand for the length indicated by the string control byte count.

The operation applies the above procedure to each record in the source until it encounters the end of the source. Updated values for the result offset and source offset are put into the appropriate fields in the controls operand. These values describe the start offsets for the next record to be converted, allowing for interruption of the instruction on a record boundary.

When an end-of-record string control byte (hex 00) is encountered in the source, the result is padded with blanks out to the end of the current record.

When the end-of-source record is encountered, whether or not in conjunction with a record boundary or end-of-string record control byte in the source, the instruction ends with a condition code of zero (source used up). See *Programming Note.*

If the converted form of a record cannot be completely contained in the result, the instruction ends with a condition code of 1 (result overrun). See *Programming Note.*

If the converted record is larger than the result record length, the instruction terminates by signaling a length conformance exception.

Programming Note: The result offset locates the byte following the last fully converted record in the result. The source offset locates the byte following the last source record for which conversion is complete. The contents of the remaining bytes in the result after the last converted record are unpredictable.

Any form of overlap between the operands on this instruction yields unpredictable results in the result operand.

Overflow and Sign Codes: Not applicable.

Condition Code:

0 Source used up
1 Result overrun
2 --
3 --

Carry and Boundary Requirements: Not applicable.

Program Exceptions:

- Address translation
- Addressing
- Conversion
- Effective address overflow
- Length conformance
- Specification

CVTMC Example

Op	B	E	B_1	D_1	B_2	D
BE	5	5	3	6A8	4	644

0 Bits 8 12 16 20 32 36 47

Assembler: CVTMC $D_1(B_1),D_2(B_2),B_3$

Machine: BE55 36A8 4644

B_1 (3):	0001 236A 0000	(Base register for result)
B_2 (4):	0001 136A 0000	(Base register for source)
B_3 (5):	0001 036A 0620	(Address of control operand)
R (14):	0020	(Length of result)
R (15):	0014	(Length of source)

Storage—Before

```
                 0/8   2/A   4/C   6/E
0001 036A 0620   0000  0000  0010  XXXX

0001 136A 0644                F080  A811
                 C8E3  C5E2  E3E3  C5E2
                 E300  FF80  84A8  2200

0001 236A 06A8   XXXX  XXXX  XXXX  XXXX
                 XXXX  XXXX  XXXX  XXXX
                 XXXX  XXXX  XXXX  XXXX
                 XXXX  XXXX  XXXX  XXXX
```

Storage—After

```
                 0/8   2/A   4/C   6/E
0001 036A 0620   0020  0014  0010  XXXX

0001 136A 0644                F080  A811
                 C8E3  C5E2  E3E3  C5E2
                 E300  FF80  8488  2200

0001 236A 06A8   1111  1111  1111  1111
                 E3C5  E2E3  E3C5  E2E3
                 4040  4040  2222  2222
                 2222  2222  0000  0000
```

	Before	**After**
Condition Code:	X	0

This page is intentionally left blank.

CONVERT PACKED DECIMAL TO LONG FLOAT (CVPDLF)

Instruction Description

The value of the second operand is converted from packed decimal to floating-point, and the result is placed in the first operand location.

Format: SS

CE		B	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first operand occupies 8 bytes and is formed according to the long floating-point field format.

The second operand occupies up to 16 bytes of storage, as specified by the operand 3 value, and has the packed fixed-point decimal format.

The third operand, halfword register hex F, contains the total and fractional digit count information for the number of decimal digits contained in the second operand. The total number of digits for operand 2 is contained, as a value between 1 and 31, in the leftmost byte of the halfword register. The number of fractional digits for operand 2 is contained as a value between 0 and 31 in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges or a specification exception is signaled. The number of integer digits in operand 2 is determined by subtracting the fractional digit count from the total digit count. The length of operand 2 in bytes, is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The number of digits specified are considered right adjusted in the operand 2 field. An even-value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the operand value.

The fixed-point decimal value is converted to floating-point as if to infinite precision. However, the precision provided by floating-point fields is not as great as that which can be provided by decimal fields. Long floating-point provides for unique representation of a maximum of 15 significant decimal digits of precision. The leftmost nonzero digit of the packed decimal number is considered the start of the significant digits of the number. Significant digits in the second operand beyond the first 15 for long floating-point may not be preserved in the result field and only serve to provide for rounding and uniqueness of conversion.

The result of this conversion is then normalized and rounded (according to the current floating-point rounding mode) to the significand length of the operand 1 field. The converted, normalized, and rounded result is then assigned to operand 1 in the long floating-point format.

Conversion of a zero value second operand results in a zero value of the same sign being assigned to operand 1.

Operand 2 is checked for valid decimal sign and digit codes. The data exception is signaled if any invalid values are encountered, and the operation is suppressed. If an even number of digits was specified, the leftmost digit position of the packed operand 2 field is not checked and is not used as part of the fixed-point decimal value.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow
- Floating-point inexact result
- Specification

CVPDLF Example

Op		E	B_1	D_1	B_2	D_2
CE		B	2	040	2	06C

0 Bits 8 12 16 20 32 36 47

Assembler: CVPDLF $D_1(B_1)$, $D_2(B_2)$

Machine: CE0B 2040 206C

B_1(2) and B_2(2): 800D 0C00 0000

R(F): 0703

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0040	xxxx	xxxx	xxxx	xxxx
800D 0C00 006C			0000	007D

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0040	BF7C	AC08	3126	E979
800D 0C00 006C			0000	007D

CONVERT PACKED DECIMAL TO SHORT FLOAT (CVPDSF)

Instruction Description

The value of the second operand is converted from packed decimal to floating-point, and the result is placed in the first operand location.

Format: SS

AE		B	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first operand occupies 4 bytes and is formed according to the short floating-point field format.

The second operand occupies up to 16 bytes of storage, as specified by the operand 3 value, and has the packed fixed-point decimal format.

The third operand, halfword register hex F, contains the total and fractional digit count information for the number of decimal digits contained in the second operand. The total number of digits for operand 2 is contained, as a value between 1 and 31, in the leftmost byte of the halfword register. The number of fractional digits for operand 2 is contained as a value between 0 and 31 in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges or a specification exception is signaled. The number of integer digits in operand 2 is determined by subtracting the fractional digit count from the total digit count. The length of operand 2, in bytes, is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The number of digits specified are considered right adjusted in the operand 2 field. An even-value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the operand value.

The fixed-point decimal value is converted to floating-point as if to infinite precision. However, the precision provided by floating-point fields is not as great as that which can be provided by decimal fields. Short floating-point provides for unique representation of a maximum of 7 significant decimal digits of precision. The leftmost nonzero digit of the packed decimal number is considered the start of the significant digit of the number. Significant digits in the second operand beyond the first 7 for short floating-point may not be preserved in the result field and only serve to provide for rounding and uniqueness of conversion.

The result of this conversion is then normalized and rounded (according to the current floating-point rounding mode) to the significand length of the operand 1 field. The converted, normalized, and rounded result is then assigned to operand 1 in the short floating-point format.

Conversion of a zero value second operand results in a zero value of the same sign being assigned to operand 1.

Operand 2 is checked for valid decimal sign and digit codes. The data exception is signaled if any invalid values are encountered, and the operation is suppressed. If an even number of digits was specified, the leftmost digit position of the packed operand 2 field is not checked and is not used as part of the fixed-point decimal value.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow
- Floating-point inexact result
- Specification

CVPDSF Example

Op		E	B_1	D_1	B_2	D_2
AE		B	2	058	2	06C

0 Bits 8 12 16 20 32 36 47

Assembler: CVPDSF $D_1(B_1)$, $D_2(B_2)$

Machine: AE0B 2058 206C

B_1(2) and B_2(2): 800D 0C00 0000

R(F): 0703

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0058	xxxx	xxxx		
800D 0C00 006C			0000	007D

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0058	BBE5	6042		
800D 0C00 006C			0000	007D

CONVERT PACKED TO BINARY (CVPB)

Instruction Description

The radix of the second operand is changed from decimal to binary, and the result is placed in the first-operand location.

Format: SS

F7		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The number is treated as a right-aligned, binary value both before and after conversion. The second operand occupies 8 bytes in storage and has the packed decimal format. The digit codes are checked for validity. Improper codes cause a data exception and the operation is terminated. The first operand occupies a word in storage and is formed using the signed binary format.

The maximum number that can be converted to a 32-bit, signed, binary integer is 2 147 483 647; the minimum number is -2 147 483 648. For any number outside this range, the operation is completed by placing the rightmost 32 bits in the first-operand location and causing a binary overflow exception.

Overflow: See *Operation.*

Sign Code: The sign code is checked for validity. An improper code causes a data exception and the operation is terminated.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Both operands must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed. The operands can overlap in storage.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Data
- Effective address overflow
- Specification

CVPB Example

Op F7		E 0	B_1 3	D_1 760	B_2 3	D_2 430

0 Bits 8 12 16 20 32 36 47

Assembler: CVPB $D_1(B_1)$, $D_2(B_2)$

Machine: F700 3760 3430

$B_1(3)$ and $B_2(3)$: 1810 2561 C000

Storage — Before

	0/8	2/A	4/C	6/E
1810 2561 C430	0000	0214	7483	647F
1810 2561 C760	xxxx	xxxx		

Storage — After

	0/8	2/A	4/C	6/E
1810 2561 C430	0000	0214	7483	647F
1810 2561 C760	7FFF	FFFF		

CONVERT PACKED TO ZONED (CVPZ)

Instruction Description

The format of the second operand is changed from packed to zoned, and the result is placed in the first-operand location.

Format: SS

F5	L_1	L_2	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The digits and sign of the second operand are placed unchanged in the first-operand location, using the zoned format. Zones with coding of 1111 are supplied for all bytes except the rightmost byte, which receives the sign of the packed operand. The operand sign and digits are not checked for valid codes.

The result is obtained as if the field were processed right to left. If necessary, the second operand is logically extended to the left with zero digits.

Overflow: If the first-operand field is too short to contain all significant digits of the second-operand field, decimal overflow occurs and the leftmost significant digits are lost.

Sign Code: See *Operation.*

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is to the right of the rightmost byte of the second operand by the number of bytes in the second operand minus two; otherwise the overlap is destructive and the results are unpredictable. If 1 or 2 bytes are converted, the rightmost byte of the two operands can coincide.

Program Exceptions:

- Address translation
- Addressing
- Decimal overflow
- Effective address overflow

CVPZ Example

Op F5	L_1 7	L_2 3	B_1 4	D_1 270	B_2 4	D_2 100	
0 Bits	8	12	16	20	32	36	47

Assembler: CVPZ $D_1(L_1, B_1)$, $D_2(L_2, B_2)$

Machine: F573 4270 4100

B_1(4) and B_2(4): 30B8 5693 C000

Storage — Before

	0/8	2/A	4/C	6/E
30B8 5693 C100	0210	261F		
30B8 5693 C270	xxxx	xxxx	xxxx	xxxx

Storage — After

	0/8	2/A	4/C	6/E
30B8 5693 C100	0210	261F		
30B8 5693 C270	F0F0	F2F1	F0F2	F6F1

CONVERT PACKED TO ZONED WITH DATA CHECKING (CVPZC)

Instruction Description

The format of the second operand is changed from packed to zoned, and the result is placed in the first-operand location.

Format: SS

E5	L_1	L_2	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The digits and sign of the second operand are placed unchanged in the first-operand location, using the zoned format. Zones with coding of 1111 are supplied for all bytes except the rightmost byte, which receives the sign of the packed operand. If the sign of the packed operand is positive, the preferred sign of 1111 is used. If the sign of the packed operand is negative, the preferred sign of 1101 is used, unless the digits are all zero, in which case a plus sign of 1111 is used. The operand sign and digits are checked for valid codes.

The result is obtained as if the field were processed right to left. If necessary, the second operand is logically extended to the left with zero digits.

The length of each operand is in digits.

Overflow: If the first-operand field is too short to contain all significant digits of the second-operand field, decimal overflow occurs and the leftmost significant digits are lost.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is to the right of the rightmost byte of the second operand by the number of bytes in the second operand minus two; otherwise, the overlap is destructive and the results are unpredictable. If 1 or 2 bytes are converted, the rightmost byte of the two operands can coincide.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Invalid segment group address

CVPZC Example

Op E5	L_1 7	L_3 6	B_1 4	D_1 270	B_2 4	D_2 100	
0 Bits	8	12	16	20	32	36	47

Assembler: CVPZC $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: E576 4270 4100

$B_1(4)$ and $B_2(4)$: 30B8 5693 C000

Storage — Before

	0/8	2/A	4/C	6/E
30B8 5693 C100	0210	261F		
30B8 5693 C270	xxxx	xxxx	xxxx	xxxx

Storage — After

	0/8	2/A	4/C	6/E
30B8 5693 C100	0210	261F		
30B8 5693 C270	F0F0	F2F1	F0F2	F6F1

CONVERT SHORT FLOAT TO BINARY (CVSFB)

Instruction Description

The value stored at the second operand location is converted from floating-point to binary and placed in the first operand location.

Format: *SS*

AE	M	8	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: Operand 1 has a signed binary format and is either 2, 4, or 8 bytes in length. The length of the operand is determined by an options mask.

Operand 2 is 4 bytes long, and has a short floating-point format. The data for this operand must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Operand 3 is a 4-bit options mask (bits 8 through 11) that controls the conversion operation. The format of the options mask is:

Bits	**Meaning**
8	Mode of rounding to be performed. 0 = Round using current floating-point rounding mode in effect. 1 = Round using decimal round algorithm.
9-11	Length of binary result (operand 1). 001 = 2 bytes. 011 = 4 bytes. 111 = 8 bytes. All other values are invalid.

The floating-point value of the second operand is converted to a fixed-point binary integer format. If necessary, the floating-point value is rounded to an integer value.

The rounding mode is specified by the options mask (bit 8 of operand 3). If floating-point rounding is specified, rounding is performed according to the current floating-point rounding mode in effect. If decimal rounding mode is specified, the current floating point rounding mode is overridden and the decimal round algorithm is performed. In this case, a value of 1/2 (a 1 bit) is added to the leftmost bit position of the fractional portion of the floating-point value, and that bit, and those bits to the right, are truncated from the resulting value.

The value assigned to operand 1 is formed as a right-aligned, binary integer value with an assumed binary point immedately to the right of its rightmost digit.

If the rounded integer portion of the floating-point value is zero, the first operand value is set to zero, and the sign is set positive, regardless of the sign of the second operand.

An invalid floating-point conversion exception is signaled for any number outside the range of integer values that can be contained in operand 1 (this includes NaNs and infinities).

The result obtained from overlapping operands is unpredictable.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must be on a fullword boundary; otherwise, a specification exception occurs.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Invalid floating-point conversion
- Specification

This page is intentionally left blank.

Programming Note: The following is a summary of the results for various combinations of operands.

Receiver	**Source**
0	±0
-B	-R
+B	+R
IFPC	±INF
IFPC	MNaN
IFPC	UNaN

Legend:

R	=	a real nonzero floating-point number
B	=	a nonzero binary number
MNaN	=	a masked NaN
INF	=	infinity
IFPC	=	an invalid floating-point conversion exception

The assignment of a real number (R), as the value of the binary field (B), is only successful if R is a value that can be contained within the value range of the binary field; otherwise, an invalid floating-point conversion may result.

CVSFB Example

Op	M_3	E	B_1	D_1	B_2	D_2
AE	3	8	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: CVSFB $D_1(B_1)$, $D_2(L_2B_2)$

Machine: AE38 4050 4060

B_1(4) and B_2(4): 0010 0200 0000

Storage – Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx		
0010 0200 0060	4D80	8080		

Storage – After

	0/8	2/A	4/C	6/E
0010 0200 0050	1010	1000		
0010 0200 0060	4D80	8080		

Condition Code: Not changed.

CONVERT SHORT FLOAT TO DECIMAL FORM (CVSFDF)

Instruction Description

The binary floating-point value specified by operand 5 is converted to a decimal form of a floating-point value (a decimal exponent and a decimal significand and placed into operand 1 (exponent) and operand 2 (significand) locations.

Format: SS

AE	M_3	C	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first operand specifies the decimal exponent and occupies from 3 to 16 bytes as specified by the operand 4 value. This operand is formed using the packed fixed-point decimal format.

The second operand specifies the decimal significand, and occupies a maximum of 16 bytes of storage as specified by the operand 4 value. This operand is formed using the packed fixed-point decimal format.

The third operand, bits 8 through 11, specifies an options mask to control the conversion operation.

Bits	Meaning
8	Mode of rounding to be performed. 0 = Round using current float rounding mode in effect. 1 = Round using decimal round algorithm.
9-11	Reserved.

The fourth operand, halfword register hex F, contains the digit lengths of the first and second operands. The total number of digits for operand 1 is contained as a value between 5 and 31 in the leftmost byte of the halfword register. The total number of digits for operand 2 is contained as a value between 1 and 31 in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges, or a specification exception is signaled. The length of operands 1 and 2 (in bytes) is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The number of digits specified are considered right adjusted in their respective fields. An even-value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the corresponding operand value.

The fifth operand, base register hex E, specifies the address of the binary floating-point number. The number occupies 4 bytes, and has the short floating-point field format.

The exponent (operand 1) and significand (operand 2) contain a decimal form of a floating-point number. The value of this number is:

Value = M * (10**E)
where:
M = the value of the significand operand
E = the value of the exponent operand
** denotes exponentiation
* denotes multiplication

The exponent is formed as a decimal integer value. The exponent, which gives the floating-point value its magnitude, contains a signed integer value that specifies a power of 10. The exponent has an assumed decimal point immediately to the right of its rightmost digit position.

The significand is formed as a decimal value with a single integer digit in its leftmost digit position and fractional digits in the digit positions to the right of the integer digit. The significand contains a signed decimal value that specifies decimal digits to give the floating-point value its precision. The significand has an assumed decimal point immediately to the right of its leftmost digit position.

The binary floating-point source is converted to a decimal form floating-point value as if to infinite precision. However, the precision provided for by floating-point fields is not as great as the precision provided for by decimal fields. Short floating-point provides for unique representation of a maximum of 7 significant decimal digits of precision. The significant digits of the significand start with the leftmost nonzero decimal digit and continue to the right out to the end of the significand value. The converted significand value is formed as a normalized value, the significant digits are left adjusted in the converted value, and the converted exponent is set accordingly. Significand digits beyond the leftmost 7 provide for uniqueness of the conversion and should be considered only as precise as the floating-point calculations that produced the sourve value.

The converted significand value is adjusted to the precision of the significand operand, if necessary, by using the rounding algorithm specified in the options mask operand. If the rounding algorithm causes a carry out of the leading integer digit position, the converted rounded significand value is shifted right one digit position and the converted exponent incremented by one to realign the significand back to having one leading integer digit. If floating-point rounding is selected, rounding is performed according to the current floating-point rounding mode in effect. If decimal rounding is selected, the current floating-point rounding mode is overridden and the decimal round algorithm is performed. In this case, a value of 5 is added to the converted significand in the leftmost digit position not provided for in operand 2, and that digit, and those digits to the right of it are truncated from the resulting significand value.

The result of this conversion is then assigned to the exponent and significand operands. For an exponent or significand operand with an even number of digits, the leftmost digit position of the packed field in the operand is set to binary 0.

If the binary floating-point number being converted contains a value of 0, the exponent operand is set to positive 0, and the significand operand is set to 0 with the sign of the binary floating-point number. A positive 0 is set with the preferred positive sign of hex F. A negative 0 is set with the preferred negative sign of hex D.

A decimal overflow exception cannot occur on the assignment of the exponent significand values.

When the binary floating-point number being converted contains a denormalized floating-point value, the first and second operand values are set with the correctly converted and rounded values; no exception is signaled.

When an infinity or NaN value is encountered in the second operand, the invalid floating-point conversion exception is signaled and the instruction operation is suppressed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlap between operands 1 and 2 is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid floating-point conversion
- Specification

Programming Note: The following is a summary of results for various combinations of operands.

Receivers	Source
-0*10**+0	-0
+0*10**+0	+0
-M*10**+E	-R<-1
+M*10**+E	+R>1
-M*10**-E	-R>-1
+M*10**-E	+R<1
IFPC	±INF
IFPC	MNaN
IFPC	UNaN

Legend:

R	=	a real nonzero, nondenormal floating-point number
E	=	the exponent, a nonzero decimal number
M	=	the significand, a nonzero decimal number
MNaN	=	a masked NaN
UNaN	=	an unmasked NaN
INF	=	infinity
IFPC	=	invalid floating-point conversion exception
**		denotes exponentiation
*		denotes multiplication

CVSFDF Example

Op	M_3	E	B_1	D_1	B_2	D_2
AE	0	C	2	064	2	07E

0 Bits 8 12 16 20 32 36 47

Assembler: CVSFDF $D_1(B_1)$, $D_2(B_2)$, M_3

Machine: AE0C 2064 207E

B_1(2) and B_2(2): 800D 0C00 0000

B(E): 800D 0C00 0300

R(F): 070F

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0064			xxxx	xxxx
800D 0C00 007E				xxxx
	xxxx	xx		
800D 0C00 0300	405E	C000		

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0064			0000	000F
800D 0C00 007E				3480
	4687	5F		
800D 0C00 0300	405E	C000		

CONVERT SHORT FLOAT TO PACKED DECIMAL (CVSFPD)

Instruction Description

The value of the second operand is converted from floating-point to packed decimal, and the result is placed in the first operand location.

Format: SS

AE	M_3	A	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The first operand occupies up to 16 bytes of storage, as specified by the operand 4 value, and is formed according to the packed fixed-point decimal format.

The second operand occupies 4 bytes, and has the short floating-point field format.

The third operand, bits 8 through 11, specifies an options mask to control the conversion operation.

Bits	Meaning
8	Mode of rounding to be performed. 0 = Round using current float rounding mode in effect. 1 = Round using decimal round algorithm.
9-11	Reserved.

The fourth operand, halfword register hex F, contains the total, and the fractional digit count information for the number of decimal digits contained in the first operand. The total number of digits for operand 1 is contained, as a value between 1 and 31, in the leftmost byte of the halfword register. The number of fractional digits for operand 1 is contained as a value between 0 and 31, in the rightmost byte of the halfword register. The specified digit lengths must be within the allowable ranges or a specification exception is signaled. The number of integer digits in operand 1 is determined by subtracting the fractional digit count from the total digit count. The length of operand 1, in bytes, is calculated by dividing the total digit count by 2 and adding 1 to the resulting quotient. The number of digits specified are considered right adjusted in the operand 1 field. An even-value digit length indicates the leftmost digit position of the packed field is not to be considered a digit position of the operand value.

The floating-point value is converted to a fixed-point packed decimal number as if to infinite precision. However, the precision provided by floating-point fields is not as great as that which can be provided by decimal fields. Short floating-point provides for unique representation of a maximum of 7 significant decimal digits of precision. The leftmost nonzero digit of the converted packed decimal number is the start of the significant digits of the number. Significant digits produced in the first operand beyond the first 7 for short floating-point provide for uniqueness of conversion and should be considered only as precise as the calculations that produced the floating-point number.

The result of this conversion is then rounded, if necessary, to match the fractional precision of the operand 1 field. The rounding algorithm performed is controlled by the third operand mask value. If floating-point rounding is selected, rounding is performed according to the current floating-point rounding mode in effect. If decimal rounding is selected, the current floating-point rounding mode is overridden and the decimal round algorithm is performed. In this case, a value of 5 is added to the converted number in the leftmost digit position not provided for in operand 1, and that digit, and those to the right of it, are truncated from the resulting sum.

The converted and rounded result is then assigned to operand 1 in the fixed-point packed decimal format for the number of digits specified by the total digit count for operand 1. If any even number of digits was specified, the leftmost digit position of the packed operand 1 field is set to binary 0.

If the converted and rounded result is 0, the first operand value is set to 0, and the sign is set positive, regardless of the sign of the second operand.

When a denormalized floating-point value is converted from the source operand, the first operand is set with the correctly rounded value, and no exception is signaled.

When any nonzero integer digits are truncated on the left in assigning the converted and rounded result to operand 1, or when an infinity value or a NaN value is encountered in the second operand, the invalid floating-point conversion exception is signaled, and the instruction operation is suppressed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid floating-point conversion
- Specification

Programming Note: The following is a summary of the results for various combinations of operands.

Receiver	**Source**
+0	±R0
-D	-R
+D	+R
IFPC	±INF
IFPC	MNaN
IFPC	UNaN

Legend

R	=	a real nonzero value converted and rounded form of the source floating-point number
R0	=	a real zero value converted and rounded form of the source floating-point number
D	=	a nonzero decimal number
MNaN	=	a masked NaN
UNaN	=	an unmasked NaN
INF	=	infinity
IFPC	=	invalid floating-point conversion exception

The assignment of a real number, R, as the value of the decimal field, D, is only successful if R is a value that can be contained within the value range of the decimal field; otherwise, an invalid floating-point conversion may result.

CVSFPD Example

Op	M_3	E	B_1	D_1	B_2	D_2
AE	0	A	2	068	2	060

0 Bits 8 12 16 20 32 36 47

Assembler: CVSFPD $D_1(B_1)$, $D_2(B_2)$, M_3

Machine: AE0A 2068 2060

$B_1(2)$ and $B_2(2)$: 800D 0C00 0000

R(F): 0703

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0068	xxxx	xxxx		
800D 0C00 0060	405E	C000		

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0068	0003	480F		
800D 0C00 0060	405E	C000		

CONVERT SHORT TO LONG FLOAT (CVSLF)

Instruction Description

The value of the second operand is converted from the short floating-point format to the long floating-point format, and the result is placed in the first operand location.

Format: SS

AE		7	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The first operand occupies 8 bytes in storage and is formed using the long floating-point field format.

The second operand occupies 4 bytes in storage and has the short floating-point field format.

When the second operand contains a normalized nonzero floating-point value, the significand value from the second operand is padded on the right with 0 bits to the long floating-point format significand length. The biased exponent value of the second operand is adjusted to the correct biased exponent value for the long floating-point format. This converted floating-point value is then assigned to the first operand according to the long floating-point field format. The sign bit value of operand 2 is assigned to the sign bit for operand 1.

When the second operand contains a value of 0, the first oprand is assigned a zero value of the same sign.

When the second operand contains an infinity floating-point value or a masked NaN value, the exponent is padded on the right with 1 bits to the long floating-point format exponent length, and the significand is padded on the right with 0 bits to the long floating-point format significand length. The padding occurs prior to their assignment into operand 1. The sign bit value of operand 2 is assigned to the sign bit for operand 1.

When the second operand contains an unmasked NaN value, the floating-point invalid operand condition is detected. For the case where the floating-point invalid operand condition is detected. For the case where the floating-point invalid operation exception is masked, operand 1 is assigned with a masked NaN value with the fraction value from the original unmasked NaN padded with zeros on the right out to the long floating-point format fraction length.

When the second operand contains a denormalized floating-point number, the result field is assigned the correctly converted and normalized value, and no exception is signaled.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point invalid operand
- Specification

CVSLF Example

Op		E	B_1	D_1	B_2	D_2
AE		7	2	040	2	060

0 Bits 8 12 16 20 32 36 47

Assembler: CVSLF $D_1(B_1)$, $D_2(B_2)$

Machine: AE07 2040 2060

$B_1(2)$ and $B_2(2)$: 800D 0C00 0000

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0040	xxxx	xxxx	xxxx	xxxx
800D 0C00 0060	405E	C000		

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0040	400B	D800	0000	0000
800D 0C00 0060	405E	C000		

CONVERT SNA TO CHARACTER (CVTSC)

Instruction Description

The operation converts the data at the second operand location from SNA (systems network architecture) compressed format. The conversion is controlled by information whose address is in the base register specified in the third operand. The result is placed in the first operand.

The operands are as follows:

Operand	**Description**
1	The base and displacement for the starting address of the result string that is to contain the converted data.
2	The base and displacement for the starting address of the source string that contains the data to be converted.
3	The base register that contains the address of the control information for the conversion operation to be performed.
4	Halfword register 14 specifies the length of the first operand (result string). A length of zero causes a specification exception.
5	Halfword register 15 specifies the length of the second operand (source string). A length of zero causes a specification exception.

The controls operand is a 14-byte string with an optional extension that specifies additional information to control the conversion operation. The controls operand has the following format:

Receiver Offset	Source Offset	Algorithm Modifier	Receiver Record Length	Record Separator Character	Prime Compression Character	Unconverted Receiver Record Bytes	Conversion Status	Unconverted Transparency String Bytes	Offset to Translate Table	
0	2 Bytes	4	5	6	7	8	9	11	12	14

	Record Separator Translate Table	
	0	63

Bytes	Description
0-1	Receiver offset
2-3	Source offset
4	Algorithm modifier
5	Receiver record length
6	Record separator character
7	Prime compression character
8	Unconverted receiver record bytes
9-10	Conversion status
11	Unconverted transparency string bytes
12-13	Offset to translate table
0-63	Record separator translate table

Upon input to the instruction, the *result offset* and the *source offset* fields specify the offsets at which bytes of the source field are processed and entered into the result field. The source and result offset fields are set to values which indicate how much of the conversion is complete when the instruction is interrupted or complete. An offset beyond the end of the related source or resultant operand causes a specification exception.

The *modifier* has the following valid values:

Bits **Meaning**

0 Decompression.

- 0 = Do not perform decompression.
- 1 = Perform decompression.

1-2 Processing Mode.

- 00 = String processing if bits 3 or 4 = 00; record processing if bits 3 or 4 ≠ 00. No record separators in source. Do not perform blank padding; do not perform data transparency conversion.
- 01 = Reserved.
- 10 = Record processing. Record separators in source. Do blank padding; do not perform data transparency conversion.
- 11 = Record processing. Record separators in source. Do blank padding; perform data transparency conversion.

3-4 Record Separator Processing.

- 00 = Do not put record separators into receiver.
- 01 = Move record separators from source to receiver. (Allowed only if bit 1 = 1.)
- 10 = Translate record separators from source to receiver. (Allowed only if bit 1 = 1.)
- 11 = Move record separator from controls to receiver.

5-7 Reserved. Must be zero.

An invalid modifier results in a specification exception.

The *receiver record length* specifies the fixed length for each record stored into the receiver. This length does not include the record separator character. A value of zero results in a specification exception. This parameter is used in record processing mode only; it is ignored in string processing mode.

The *record separator character* specifies the character that is to precede the converted form of each record in the receiver. It can have any value.

This parameter is used where:

- A missing record separator is detected in the source.
- The move record separator from controls to receiver function is specified (algorithm modifier bits 3-4).

The *prime compression character* specifies the prime compression character for decompression purposes; it can have any value. This parameter is ignored if decompression is not specified in the algorithm modifier.

The *unconverted receiver record bytes* specifies the number of bytes remaining in the current receiver record that have not been set with converted bytes. This parameter is ignored in string processing mode. In record processing mode, the following meanings apply.

- At start of execution:
  - A value of hex 00 means this is the start of a new record and the initial conversion step has not yet been performed; if a record separator is supposed to be placed into the receiver, this has not yet been done.
  - A nonzero value less then or equal to the receiver record length specified the number of bytes remaining in the current receiver record that have yet to be set with converted bytes. Validity is assumed and not checked. This value is used to determine the location of the next record boundary in the receiver. A specification exception occurs if this value is greater than the receiver record length.
- At end of execution:
  - This field is set equal to the number of bytes in the current receiver record not yet containing converted data.

The *conversion status* contains information for checkpointing the conversion status over successive executions of the instruction. It is set to the appropriate value at instruction termination. Bit definitions are:

Bits	Meaning	
0	0 =	No transparency string active.
	1 =	Transparency string active. The unconverted transparency string bytes value contains the remaining string length.
1-15	Reserved. Must be zero.	

The *unconverted transparency string bytes* specifies the number of bytes remaining to be converted for a partially processed transparency string.

If do not perform data transparency conversion is specified, this parameter is ignored.

When perform data transparency conversion is specified, this parameter has the following meanings:

- At start of execution:
  - When the conversion status byte indicates no transparency string active (bit 0 = 0), this value is ignored.
  - When the conversion status byte indicates transparency string active (bit 0 = 1), this value is a count of the remaining bytes to be converted for a transparency string in the source. Validity of this count is assumed; it is not checked.
  - A value of hex 00 means the count field for a transparency string is the first byte to be processed from the source.
  - A value of hex 01 through hex FF specifies the count of the remaining bytes to be converted for a transparency string.

- At end of execution this parameter is set, along with transparency string active, to describe a partially converted transparency string.
  - A value of hex 00 is set if the count field is the next byte to be processed for a transparency string.
  - A value of hex 01 through hex FF (specifying the number of remaining bytes to be converted) is set if the count field has already been processed.

The *offset to translate table* specifies the offset from the beginning of the controls operand to the record separator translate table. This parameter is ignored unless the translate record separators from source to receiver function is specified.

The *record separator translate table* provides for translation of the source record separator values being placed into the receiver.

This table is assumed to be 64 bytes in length and provides for translation of record separator values from hex 00 through hex 3F.

This table is used only when translate record separators from source is specified.

Format: SS

BE	B_3	7	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The operation begins by accessing the bytes of the source operand at the location specified by the source offset. The data is converted and stored in the receiver according to the following modes and optional functions.

String Processing Mode

String processing occurs when algorithm modifier bits 1, 3, and 4 are all equal to zero. The bytes accessed in the source are converted, decompressed (algorithm modifier bit 0 must be equal to one), and then stored in the receiver.

Decompression

The decompression function is always performed in string processing mode. This function converts strings of duplicate and nonduplicate characters in the compressed format back to their full length in the receiver. Decompressed data is built by concatenating one or more character strings as described by the compression strings in the source. If necessary, the processing of a partial decompressed record is performed.

Each character string to be built in the receiver is described in the source by a compression string. Compression strings are comprised of an SCB (string control byte) followed by prescribed bytes of data related to the character string to be built in the receiver.

The SCB has the following format and bit definitions:

Bits **Meaning**

0-1 Control.

00 = n nonduplicate characters are between this SCB and the next one. n is the value in the count field; possible values are 1-63 (decimal).

01 = Reserved.

10 = This SCB represents n deleted prime compression characters. n is the value in the count field; possible values are 1-63 (decimal). The next byte is the next SCB.

11 = This SCB represents n deleted duplicate characters. n is the value of the count field; possible values are 1-63 (decimal). The next byte contains a specimen of the deleted characters. The byte following the specimen character contains the next SCB.

Bits **Meaning**

2-7 Count.

The value n (in binary) in this field represents the count of the number of characters that have been deleted for a prime compression character string, a duplicate character string, or the number of characters to the next SCB for a nonduplicate string. A count value of zero is invalid and causes a conversion exception.

Strings of prime compression characters or duplicate characters described in the source record are repeated in the decompressed character string the number of times indicated by the SCB count value.

Strings of nonduplicate characters described in the source record are formed into a decompressed character string for the length indicated by the SCB count value.

If the end of the source is encountered prior to the end of a compression string, a conversion exception is signaled.

In string processing mode:

- Decompression is performed on a compression string basis with no record oriented processing implied. The conversion process for each compression string is completed by placing the decompressed character string into the receiver.
- The conversion process continues until the end of the source or receiver is reached.
  - When the end of the source is encountered, the instruction ends with a source exhausted condition code.
  - When the end of the receiver is encountered, the instruction ends with a receiver overrun condition code.
  - For either of the previous ending conditions, the controls operand is updated to describe the status of the conversion operation as of the last completely converted compression entry. Partial conversion of a compression entry is not performed.

Record Processing Mode

Record processing occurs when bit 1, bit 3, or bit 4 is equal to one in the algorithm modifier. Source bytes are accessed, converted, and placed into the receiver on a record basis.

The source offset locates the point at which processing should start on a full record or the point at which processing should be resumed on a partial record. If the unconverted receiver record bytes value is zero, source offset points to the start of a full record. If the unconverted receiver record bytes value is nonzero, source offset points to the location where processing of a partial record should be resumed. For resumption of processing of a partial record, the value in the conversion status byte indicates whether or not a transparency string is active.

The conversion process is started by completing the conversion of a partial source record, if necessary, before processing the first full source record.

A check is made (before storing the first byte of each record), to see if the receiver has room for another full record. If not, a receiver overrun condition is recognized and the instruction is terminated; the controls operand is updated to describe the last completely converted record. Partial conversion of a source record is not made. Source data is accessed prior to this check and this may result in a source exhausted condition or a conversion exception.

Record Separator Conversion

The record separator conversion function is always performed in record processing mode. This function can be performed with, or without, the optional decompression, data transparency conversion, and blank padding functions.

In record processing mode, a record separator is recognized in the source when a character value less than hex 40 is encountered; however, if decompression is not specified (algorithm modifier bit 0 = 0), a character whose value is hex 00 is ignored. If the perform data transparency conversion function is also specified (algorithm modifier bit 2 = 1), a character value of hex 35 is recognized as the start of a transparency string instead of a record separator.

This function controls the conversion of record separators into the receiver. The four possible options are (refer to algorithm modifier bits 3-4 definitions):

- Put no record separators in receiver; any record separator found in the source data is ignored and not placed into the receiver.

- Move record separators from source to receiver; any record separator found in the source is left as is and placed into the receiver.

- Translate record separators from source to receiver; the record separator from the source is translated via the translate table. The translated value is placed into the receiver as follows:
  - The translation is performed as in the Translate instruction. The source record separator value is used as an index into the translate table; the value at that location in the table (if not hex FF) is placed into the receiver as a record separator.
  - If the indexed translate table byte is equal to hex FF, it is recognized as an escape code and the instruction ends with an escape code encountered condition code. The controls operand is set to describe the conversion status as of the processing completed just prior to the conversion step for the record separator.

- Move record separator from controls to receiver; when a record separator is found in the source, the record separator value specified in the controls operand is placed in the receiver.

Missing Record Separator Handling: During the initial processing for a full record, the controls record separator character is used as the record separator value in the receiver and the specified record separator conversion function is not performed if all of the following conditions exist:

- The first byte of data is not a record separator

- Record separators are expected in the source (algorithm modifier bit 1 = 1)

- Record separators are to be placed in the receiver (algorithm modifier bits 3 or 4 = 1)

Data Transparency Conversion

The data transparency conversion function is performed only in record processing mode (along with blank padding); decompression can be performed with this function, but is not required.

This function correctly identifies record separators in the source even though the data has value that could be interpreted as record separators.

The source data is converted and stored into the receiver and is not transparent to the scan for a record separator value unless data transparency conversion is specified, and a source byte with hex 35 value is detected. Detection of this combination indicates a transparency string; source data transparent to record separator scanning is converted.

The hex 35 byte is the first byte of a 2-byte transparency control field (TRN). The second byte is a hexadecimal count (with allowable values of 1-255) that specifies the number of following bytes to be treated as transparent data. A transparency count of zero causes a conversion exception. This transparent data is not scanned for record separators. Only the transparent data is moved to the receiver; the TRN bytes are not moved.

A record in the receiver can be a mix of converted transparent and non-transparent source data; the first byte is always a record separator unless the algorithm modifier bits 3 and 4 specify do not put record separators into receiver (algorithm modifier bits 3 and 4 = 00).

If conversion continues until the length of converted data equals the specified receiver record length, the record is complete. An active transparent string is then handled as described in the definition of the unconverted transparency string bytes parameter.

Partial record processing is performed as described in the definition of unconverted receiver record bytes parameter.

Missing record separator and record separator conversion are performed as previously described.

If the end of source is encountered before the record is completed, the controls operand is updated to describe the partially converted record (and the partially converted transparency string, if appropriate), and the instruction ends with a source exhausted condition code.

Blank Padding

The blank padding function is performed in record processing mode along with record separator conversion. Decompression and data transparency conversion can be performed with this function, but are not required.

This function pads a receiver record with blanks to the size specified by the receiver record length. The padded blanks replace the trailing blanks truncated by the Convert Character-to-SNA instruction. The padded record can be produced from a partial or full record from the source.

The record separator for this record is accessed. Missing record separator and record separator conversion are performed as previously described.

Blank padding occurs if another record separator is detected before enough data has been processed to equal the receiver record length. Blanks are added to make the length of the converted data equal the receiver record length.

If the end of the source is encountered instead of another record separator, the data processed up to that point is placed into the receiver (no blank padding is done). The instruction ends with a source exhausted condition code. The controls operand is updated to describe the status of the partially converted record.

Decompression

The decompression function is performed one record at a time.

A conversion exception is signaled if a compression string describes a character string that would span a record boundary in the receiver.

Specified record separator conversion is performed as the first step (after decompression); missing record separators are handled as described under *Record Separator Conversion.*

The specified receiver record length applies to converted data only; it does not include a record separator.

Decompression continues until one of the following conditions occur:

- A record separator character for the next record is recognized and the source contains record separators is specified.

- The amount of decompressed data required to fill the receiver record has been processed.

- The end of the source is encountered.

Transparency strings encountered are not scanned for a record separator value.

The decompressed character strings, appended to the optional record separator for this record, form the decompressed record. If the end of the source is encountered, the data decompressed to that point, appended to the optional record separator for this record, forms a partial decompressed record.

Instruction Termination

The CVTSC instruction terminates in one of the following ways:

- The end of source operand is reached. This results in a source exhausted condition code.
- The end of the receiver is reached. This results in a receiver overrun condition code.
- A hex FF value is encountered in the record separator translate table. This results in a escape code encountered condition code.

At the completion of instruction execution:

- The source offset and receiver offset parameters are updated to point to the next bytes to be operated on in the source and receiver, respectively.
- If record processing is specified, the unconverted receiver record bytes parameter is updated to specify the number of bytes remaining in the current receiver record that have yet to be set.
- If perform data transparency conversion is specified, the conversion status byte is appropriately set, and the unconverted transparency string bytes are updated to describe the partially converted string.

Any form of overlap between the operands of this instruction yields unpredictable results.

Programming Notes

The CVTSC instruction does not provide support for compression entries in the source describing data that would span records in the receiver. SNA data from some systems may violate this restriction and, as such, be incompatible with this instruction. A provision can be made to avoid this incompatibility by performing the conversion through two invocations of this instruction. The first invocation would specify just decompression, with no record separator processing. The second invocation would specify record separator processing with no decompression. This technique separates the two functions, thus avoiding the incompatibility.

When decompression is not performed and the source is specified to contain record separators, source bytes of hex 00 are ignored. They are not transferred to the receiver and are not treated as record separators. When decompression is specified, source bytes of hex 00 are not ignored. If the source contains record separators and decompression is specified, source bytes of hex 00 are considered to be record separators.

This instruction can end with the escape code encountered condition. In this case, it is expected that the user of the instruction will do some special processing for the record separator causing the condition. In order to resume execution of the instruction, the user will have to set the appropriate value for the record separator into the receiver and update the controls operand offset values correctly to provide for restarting processing at the right points in the receiver and source operands.

For the special case of a tie between the source exhausted and receiver overrun conditions, the source exhausted condition is recognized first because when source exhausted is the resultant condition, the receiver may also be empty. In this case, the offset into the receiver operand may contain a value equal to the length specified for the receiver which would cause an exception to be detected on the next invocation of the instruction. The processing performed for the source exhausted condition should provide for this case, if the instruction is to be invoked multiple times with the same controls operand value. When the receiver overrun condition is the resultant condition, the source will always have room for the conversion.

This instruction is interruptible. If interrupted, information required to continue is stored in the controls operand the the instruction address register points to the instruction so that processing continues after the interrupt.

Overflow and Sign Code: Not applicable.

Condition Codes:

0 Source Exhausted
1 Receiver Overrun
2 --
3 Escape Code Encountered

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Conversion
- Effective address overflow
- Specification

CVTSC Example

Op	B_3	E	B_1	D_1	B_2	D_2
BE	D	7	3	6A8	4	644

0 Bits 8 12 16 20 32 36 47

Assembler: CVTSC $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: BED7 36A8 4644

B_1(3): 0001 236A 0000 (Base register for result)
B_2(4): 0001 136A 0000 (Base register for source)
B_3(13): 0001 036A 0000 (Address of control operand)
R(14): 000A (Length of result)
R(15): 000C (Length of source)

Storage – Before

	0/8	2/A	4/C	6/E
0001 036A 0000	0000	0000	C005	2850
	0000	00		
0001 136A 0644			02F0	F1C3
	F201	1785	0118	1785
0001 236A 06A8	xxxx	xxxx	xxxx	xxxx
	xxxx			

Storage – After

	0/8	2/A	4/C	6/E
0001 036A 0000	000A	0008	C005	2850
	0000	00		
0001 136A 0644			02F0	F1C3
	F201	1785	0118	1785
0001 236A 06A8	F0F1	F2F2	F250	5050
	5050			

	Before	After
Condition Code:	x	1

CONVERT ZONED TO PACKED (CVZP)

Instruction Description

The format of the second operand is changed from zoned to packed and the result is placed in the first-operand location.

Format: SS

F6	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The second operand is assumed to have the zoned format. All zones are ignored, except the zone over the rightmost digit, which is assumed to represent a sign. The sign is placed in the right 4 bits of the rightmost byte and the digits are placed adjacent to the sign and to each other in the remainder of the result field. The sign and digits are moved unchanged to the first-operand field and are not checked for valid codes.

The result is obtained as if the fields were processed right to left. If necessary, the second operand is logically extended to the left with zeros.

Overflow: If the first-operand field is too short to contain all significant digits of the second-operand field, decimal overflow occurs and the leftmost significant digits are lost.

Sign Code: See *Operation.*

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise, the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Decimal overflow
- Effective address overflow

CVZP Example

Op F6	L_1 4	L_2 7	B_1 3	D_1 124	B_2 3	D_2 154

0 Bits 8 12 16 20 32 36 47

Assembler: CVZP $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: F647 3124 3154

B_1(3) and B_2(3): 2881 5655 1000

Storage — Before

	0/8	2/A	4/C	6/E
2881 5665 1124			xxxx	xxxx
	xx			
2881 5665 1154			F0F5	F7F1
	F8F9	F4F2		

Storage — After

	0/8	2/A	4/C	6/E
2881 5665 1124			0057	1894
	2F			
2881 5665 1154			F0F5	F7F1
	F8F9	F4F2		

CONVERT ZONED TO PACKED WITH DATA CHECKING AND BLANK CONVERSION (CVZPB)

Instruction Description

The format of the second operand is changed from zoned to packed and the result is placed in the first-operand location.

Format: SS

D6	L_1	L_2	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The second operand is assumed to have the zoned format. All zones are ignored, except the zone over the rightmost digit, which is assumed to represent a sign. The updated sign is placed in the right 4 bits of the rightmost byte and digits are placed adjacent to the sign in the remainder of the result field. The sign and digits are checked for valid codes. If the rightmost byte of the zoned operand is hex 40, it is considered to have a valid plus sign. If the sign of the zoned operand is positive, the preferred sign of 1111 is used. If the sign of the zoned operand is negative, the preferred sign of 1101 is used, unless the digits are all zero, then a sign of 1111 is used.

The result is obtained as if the fields were processed right to left. If necessary, the second operand is logically extended to the left with zeros.

The length of each operand is in digits. When the digit count of the first operand is even, the first four bits of the leftmost byte are set to zero.

Overflow: If the first-operand field is too short to contain all significant digits of the second operand field, decimal overflow occurs and the leftmost significant digits are lost.

Sign Code: See *Operation.*

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is coincident with, or to the right of the rightmost byte of the second operand; otherwise, the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Effective address overflow

CVZPB Example

Op D6	L_1 7	L_3 7	B_1 3	D_1 124	B_2 3	D_2 154
0 Bits	8	12	16	20	32	36 47

Assembler: CVZPB $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: D677 3124 3154

B_1(3) and B_2(3): 2881 5655 1000

Storage — Before

	0/8	2/A	4/C	6/E
2881 5665 1124			xxxx	xxxx
	xx			
2881 5665 1154			F0F5	F7F1
	F8F9	F4F2		

Storage — After

	0/8	2/A	4/C	6/E
2881 5665 1124			0057	1894
	2F			
2881 5665 1154			F0F5	F7F1
	F8F9	F4F2		

CONVERT ZONED TO PACKED WITH DATA CHECKING (CVZPC)

Instruction Description

The format of the second operand is changed from zoned to packed and the result is placed in the first-operand location.

Format: SS

E6	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The second operand is assumed to have the zoned format. All zones are ignored, except the zone over the rightmost digit, which is assumed to represent a sign. The updated sign is placed in the right 4 bits of the rightmost byte and the digits are placed adjacent to the sign in the remainder of the result field. The sign and digits are checked for valid codes. If the sign of the zoned operand is positive, the preferred sign of 1111 is used. If the sign of the zoned operand is negative, the preferred sign of 1101 is used, unless the digits are all zero, in which case a plus sign of 1111 is used.

The result is obtained as if the fields were processed right to left. If necessary, the second operand is logically extended to the left with zeros.

The length of the operands is in digits. When the digit count of the first operand is even, the first four bits of the leftmose byte are set to zero.

Overflow: If the first-operand field is too short to contain all significant digits of the second-operand field, decimal overflow occurs and the leftmost significant digits are lost.

Sign Code: See *Operation.*

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise, the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Effective address overflow

CVZPC Example

Op	L_1	L_3	B_1	D_1	B_2	D_2
E6	7	7	3	124	3	154

0 Bits 8 12 16 20 32 36 47

Assembler: CVZPC $D_1(L_1, B_1)$, $D_2(L_2, B_2)$

Machine: E677 3124 3154

B_1(3) and B_2(3): 2881 5655 1000

Storage — Before

	0/8	2/A	4/C	6/E
2881 5665 1124			xxxx	xxxx
	xx			
2881 5665 1154			F0F5	F7F1
	F8F9	F4F2		

Storage — After

	0/8	2/A	4/C	6/E
2881 5665 1124			0057	1894
	2F			
2881 5665 1154			F0F5	F7F1
	F8F9	F4F2		

This page is intentionally left blank.

DEQUEUE MESSAGE (DQM)

Instruction Description

A send/receive message is dequeued from the send/receive queue designated by the second operand.

Format: SS

DA	B_1	I	B_2	D_2	B_3	D_3
0 Bits	8	12	16	20	32	36 47

Operation: The search type is specified by the I-field. The search key, which is treated as unsigned binary data, is specified by the third operand and must be of the length specified in the queue header. The messages searched are accessed sequentially, starting with the first message. The first message satisfying the search type is dequeued. B_1 is loaded with the address of the dequeued message. If no message satisfies the search type, or if the message list is empty, B_1 is not altered.

I-Field	**Search Type**
Bit 12	Message Key = Search Key (the third operand)
Bit 13	Message Key < Search Key (the third operand)
Bit 14	Message Key > Search Key (the third operand)
Bit 15	Not used

The search type is the logical inclusive OR of the I-bits specified. For a search type of binary 000x, no keys will satisfy the search type, therefore, this combination is invalid. A specification exception occurs and the operation is suppressed.

Note: A dequeue first operation is accomplished by setting the I-field to binary 111x. In this case, any search key provides the desired operation. However, because the third operand is accessed and used in the comparison, it is convenient to specify the third operand (the search key) as the header address to eliminate a potential address translation exception. The hardware forces a zero for the length field in the header and the key value is ignored for specifications of I = binary 111x. Also, a check is not made for a page crossing in the key field if enqueue first or enqueue last is specified.

Overflow and Sign Code: Not applicable.

Condition Code:

0	One or more messages remaining after successful dequeue
1	No messages remaining after successful dequeue
2	--
3	Not dequeued

Carry: Not applicable.

Boundary Requirements: The search key specified by the third operand must be fullword aligned and cannot cross a page boundary.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRM descriptor
- Descriptor access: Monitored SRQ descriptor
- Effective address overflow

DQM Example

Op	B_1	I	B_2	D_2	B_3	D_3
DA	4	8	5	000	6	000

0 Bits 8 12 16 20 32 36 47

Assembler: DQM B_1, D_2(B_2), D_3(B_3)

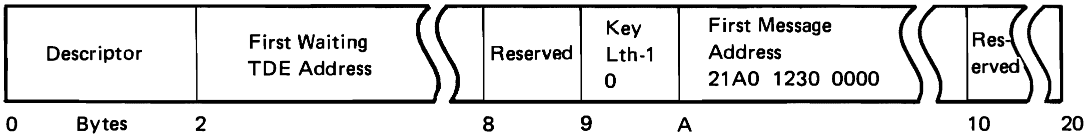
Machine: DA48 5000 6000

	Before			After		
B_1(4):	xxxx	xxxx	xxxx	21A0	1230	0000
B_2(5):	0020	3240	0000	0020	3240	0000
B_3(6):	0000	EF20	0000	0000	EF20	0000

Before

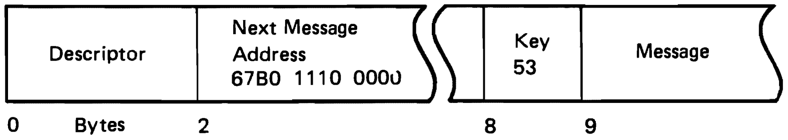
SRQ

0020 3240 0000


Descriptor
First Waiting TDE Address
Reserved
Key Lth-1 0
First Message Address 21A0 1230 0000
Res-erved
0 Bytes 2
8
9
A
10
20

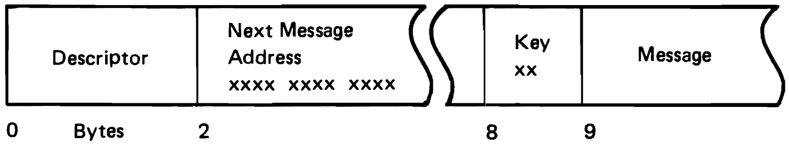
SRM

21A0 1230 0000



SRM

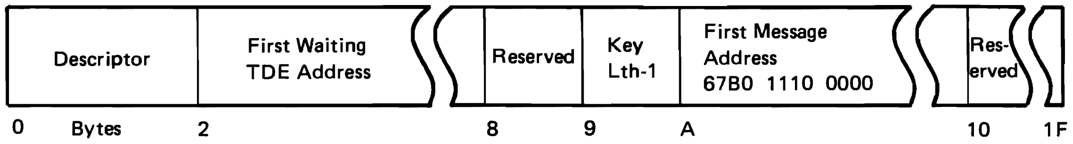
67B0 1110 0000



After

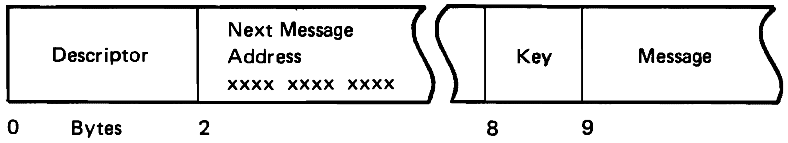
SRQ

0020 3240 0000


Descriptor
First Waiting TDE Address
Reserved
Key Lth-1
First Message Address 67B0 1110 0000
Res-erved
0 Bytes 2
8
9
A
10
1F

SRM

67B0 1110 0000



DEQUEUE TASK DISPATCHING ELEMENT (DQTDE)

Instruction Description

The TDE addressed by the first-operand is dequeued from the SRQ (send/receive queue), wait list, SRC (send/receive counter) wait list, or TDQ (task dispatching queue) designated by the second operand.

Format: RS

6D	B_1	1	B_2	D_2

0 Bits 8 12 16 20 31

Operation: No search key is used.

Overflow and Sign Code: Not applicable.

Condition Code:

0	One or more TDEs remaining after successful dequeue
1	No TDEs remaining after successful dequeue
2	--
3	Not dequeued

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRQ descriptor
- Descriptor access: Monitored TDE descriptor (if the second operand is an SRQ)
- Effective address overflow
- Invalid descriptor
- Specification

DQTDE Example

Op	B_1	E	B_2	D_2
6D	3	1	4	000

0 Bits 8 12 16 20 31

Assembler: DQTDE B_1, D_2(B_2)

Machine: 6D31 4000

	Before			After		
B_1(3):	0321	B031	0000	0321	B031	0000
B_2(4):	0001	D5C1	0000	0001	D5C1	0000

Before

TDQ

0001 D5C1 0000	Descriptor	First TDE Address 0321 B031 0000
	0	2 8

TDE

0321 B031 0000	Descriptor	Next TDE Address 0321 BC00 0000
	0	2 8

TDE

0321 BC00 0000	Descriptor	Next TDE Address xxxx xxxx xxxx
	0	2 8

After

TDQ

0001 D5C1 0000	Descriptor	First TDE Address 0321 BC00 0000
	0	2 8

TDE

0321 BC00 0000	Descriptor	Next TDE Address xxxx xxxx xxxx
	0	2 8

DIAGNOSE (DIAG)

Instruction Description

This instruction provides a way to test the I/O channel disconnect line by turning off the valid page bit in a specified I/O RAR (resolved address register) and a way to sample the system power status.

Format: SI

6D	0	7	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The first operand occupies a fullword in storage and has the following format:

Byte	Description
0	Must be hex 20 or hex 80.
1	Specified I/O RAR (reserved if byte 0 is hex 20).
2-3	Reserved.

I/O RARs begin at VLS (VAT local storage) location hex 100, so by coding hex B1 in byte 1 of the first operand, VLS (1B1) is modified.[1]

If byte 0 does not equal hex 20 or hex 80 or if byte 0 is hex 80 and byte 1 specifies I/O RAR greater than hex DF on Models 3, 4, and 5, the operation is suppressed. If byte 0 is hex 80, the valid page bit is reset in the specified I/O RAR. The position of the valid page bit is indicated below:

Model	Bit
3, 4, and 5	17
6, 7, and 8	15

If byte 0 is hex 20, the system power status is returned in base register 5 as follows:

0 = Not valid
1 = System under utility power
2 = System under UPS power
3 = System power fluctuating

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand must begin on a fullword boundary; otherwise, a specification exception occurs and the operation is suppressed.

DIAG Example

Op		E	B_1	D_1
6D	0	7	3	D00

0 Bits 8 12 16 20 31

Assembler: DIAG $D_1(B_1)$

Machine: 6D07 3D00

B_1(3): 00AF 0210 0000

Storage – Before and After

	0/8	2/A	4/C	6/E
00AF 0210 0D00	80B1	00		

	Before	After
VLS (1B1)	XXXX EXXX	XXXX AXXX

[1] VLS (1b1) is location 1B1 in the VAT (virtual address translator) local storage. The VAT local storage array is used to buffer the necessary information required during virtual address translation. For additional information see the *IBM System/38 Processing Unit Theory-Maintenance* manuals, for Models 3, 4, and 5, SY31-0524, for Model 7, SY31-0649.

DISABLE TASK DISPATCHING (DTD)

Instruction Description

This instruction disables task dispatching by setting the task dispatcher mask (byte hex 22, bit 7 of LSR [local storage register]) off and stops the task interval timer. This LSR is in HMC (horizontal microcode).

Format: RR

0D		1
0 Bits	8	12 15

Operation: No other task can be executed until the task dispatcher mask is turned on by the Enable Task Dispatching instruction. Program exceptions are handled by the machine check handler while the task dispatcher mask is off.

A machine check will be reported if a Receive Message, Receive Count, Dispatch Task Dispatching Queue, or Supervisor Linkage instruction is executed while the task dispatcher mask is off.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

Programming Note: When task dispatching is disabled, the task interval timer will not be decremented. The second interval timer, the clock comparator, and the time-of-day clock will continue to function.

DTD Example

Op 0D		E 1
0 Bits	8	12 15

Assembler: DTD

Machine: 0D01

The following bit is reset by this instruction: LSR byte 22 (FLG0), bit 7 (flag task switch blocked).

DISPATCH TASK DISPATCHING QUEUE (DTDQ)

Instruction Description

See *Operation.*

Format: SI

6D		6		
0 Bits	8	12	16	20 31

Operation: If the top TDE (task dispatching element) of the TDQ is the current TDE, the following occur: the PEM (program event monitor) mode is enabled or disabled according to the setting of byte hex C, bit 6 of the current TDE; task switch trace is set according to byte hex C, bit 2; SVLM1 operation is set according to byte hex C, bit 3; the SVLM (supervisor linkage mask) status is set according to byte hex C, bit 7; and the next sequential instruction is executed. Otherwise the task dispatcher is invoked.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

DTDQ Example

Op 6D		**E** 6		
0 Bits	8	12	16	20 31

Assembler: DTDQ

Machine: 6D06 0000

DIVIDE HALFWORD STORAGE (DHS)

Instruction Description

The dividend (first operand) is divided by the divisor (second operand) and replaced by the quotient and remainder.

Format: SS

DD		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The dividend is a 32-bit signed-binary value occupying a word of storage at the first-operand location. It is replaced by a 16-bit, signed remainder and a 16-bit signed quotient occupying the first and second halfwords, respectively. The divisor is a 16-bit signed integer.

When the relative magnitude of the dividend and divisor is such that the quotient cannot be expressed by a 16-bit signed integer, a binary divide exception occurs and the operation is suppressed. This includes attempts to divide by zero.

Overflow: Not applicable.

Sign Code: The sign of the quotient is determined by the rules of algebra. The remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The halfword storage operand must start on a halfword boundary and the word storage operand must start on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Binary divide
- Effective address overflow
- Specification

DHS Example

Op		E	B_1	D_1	B_2	D_2
DD		0	3	A40	3	A80

0 Bits 8 12 16 20 32 36 47

Assembler: DHS $D_1(B_1)$, $D_2(B_2)$

Machine: DD00 3A40 3A80

B_1(3) and B_2(3): 3456 BACD 8000

Storage — Before

	0/8	2/A	4/C	6/E
3456 BACD 8A40	0001	1000		
3456 BACD 8A80	0006			

Storage — After

	0/8	2/A	4/C	6/E
3456 BACD 8A40	0002	2D55		
3456 BACD 8A80	0006			

DIVIDE LONG FLOAT (DLF)

Instruction Description

The first operand is divided by the second operand (two-operand format) or the second operand is divided by the third operand (three-operand format), and the quotient is placed in the first operand location. No remainder is saved.

Format: SS

CE	B_3	4	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: A two-operand or three-operand format may be specified. A two-operand format is used if base register 0 is specified for the third operand. The three-operand format is used if one of the base registers hex 1 through hex F is specified for the third operand.

Floating-point division uses exponent subtraction and significand division. The difference between the signed (unbiased) exponents of the dividend and the divisor operands is used as the signed exponent of the intermediate quotient when both the dividend and the divisor are normalized.

When the dividend is denormalized and the divisor is normalized and nonzero, the difference between the signed exponents of the dividend and divisor operands less 1 is used as the signed exponent of the intermediate quotient.

All dividend and divisor significand digits participate in forming the significand of the quotient. The intermediate quotient is calculated as if to infinite precision.

Normalizing the intermediate quotient is never necessary, but a right shift of one digit position can be called for, which causes the intermediate quotient significand to be shifted right one digit position.

The intermediate quotient is rounded, if necessary, according to the rounding mode specified in the task dispatching element.

When the dividend is 0 and the divisor is a finite number or infinity, the quotient is 0.

If the divisor is denormalized and the dividend is normalized and not normal 0, an invalid operand exception is recognized.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the quotient. The source operands are checked for this value in order of their specification with the masked not-a-number with the larger fraction value being provided as the quotient.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the quotient is determined by the rules of algebra. This amounts to the exclusive OR of the divisor and dividend signs and applies to quotients of 0 or infinity value as well as for normal finite results.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: All operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Floating-point zero divide
- Specification

Programming Note: The following is a summary of the results for various combinations of operands.

Quotient	First Operand (Divisor)	Second Operand (Dividend)
Invalid operand exception	+0 or -0	+0 or -0
Divide by zero	+Real number ≠ 0 or -real number ≠ 0	+0 or -0
+0	+0	+Real number ≠ 0
+0	-0	-Real number ≠ 0
-0	-0	+Real number ≠ 0
-0	+0	-Real number ≠ 0
+0 (see note)	+Real number ≠ 0	+Larger real number ≠ 0
+0 (see note)	-Real number ≠ 0	-Larger real number ≠ 0
-0 (see note)	-Real number ≠ 0	+Larger real number ≠ 0
Invalid operand exception	+Infinity or -infinity	+Infinity or -infinity
+Infinity	+Infinity	+Real number ≠ 0
+Infinity	+Infinity	+0
+Infinity	-Infinity	-Real number ≠ 0
+Infinity	-Infinity	-0
-Infinity	-Infinity	+Real number ≠ 0
-Infinity	-Infinity	+0
-Infinity	+Infinity	-Real number ≠ 0
-Infinity	+Infinity	-0
+0	+Real number ≠ 0	+Infinity
+0	+0	+Infinity
+0	-Real number ≠ 0	-Infinity

Note: For a small value real number that is not equal to 0 and a larger value real number that is not equal to 0, a masked floating-point underflow which yields a 0 rather than a denormalized result can occur.

Quotient	First Operand (Divisor)	Second Operand (Dividend)
+0	-0	-Infinity
-0	-Real number ≠ 0	+Infinity
-0	-0	+Infinity
-0	+Real number ≠ 0	-Infinity
-0	+0	-Infinity
Masked not-a-number	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-number	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operand exception	Unmasked not-a-number	Any
Invalid operand exception	Any	Unmasked not a number

Legend:

Any = Any floating-point value.

DLF Example

Op	B_3	E	B_1	D_1	B_2	D_2
CE	3	4	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: DLF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: CE34 4050 4060

B_3(3): 0010 0200 0070

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	4894	AC90	0000	0000
0010 0200 0070	4442	3000	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4442	3000	0000	0000
0010 0200 0060	4894	AC90	0000	0000
0010 0200 0070	4442	3000	0000	0000

Condition Code: Not changed.

DIVIDE PACKED (DP)

Instruction Description

The dividend (first operand) is divided by the divisor (second operand) and replaced by the quotient and remainder.

Format: SS

F4	L_1	L_2	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The quotient is placed leftmost in the first-operand field. The remainder is placed rightmost in the first-operand field and has a length equal to the divisor length. Together, the quotient and remainder occupy the entire dividend field.

The dividend, divisor, and remainder are all signed integers, right-aligned in their fields. The quotient is a signed integer, but is left-aligned in its field. The digit codes are checked for validity; invalid codes cause data exceptions, and the operation is terminated.

When division by zero is attempted, a decimal zero divide exception occurs and the operation is suppressed.

Overflow: When the quotient is larger than the number of digits allowed, an overflow occurs and the rightmost significant digits are lost.

Sign Code: The sign of the quotient is determined by the rules of algebra from the dividend and divisor signs. The remainder has the same sign as the dividend except that a zero quotient or a zero remainder is always positive. The processor uses the preferred signs for the quotient and remainder as follows: a positive sign is encoded as 1111 (hex F); a negative sign is encoded as 1101 (hex D).

The sign codes are checked for validity; invalid codes cause a data exception, and the operation is terminated.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The divisor and dividend fields can overlap only if their rightmost bytes coincide. Improperly overlapping fields cause a data exception, and the operation is terminated.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Decimal zero divide
- Effective address overflow
- Specification

Note: The length of the quotient field is L_1 minus L_2 bytes. When the divisor length is larger than 7 (15 digits and sign) or larger than or equal to the dividend length, a specification exception occurs.

DP Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
F4	3	2	4	3B0	4	3D0

0 Bits 8 12 16 20 32 36 47

Assembler: DP $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: F432 43B0 43D0

B_1(4) and B_2(4): 000A 1234 5000

Storage — Before

	0/8	2/A	4/C	6/E
000A 1234 53B0	0000	256F		
000A 1234 53D0	0003	0F		

Storage — After

	0/8	2/A	4/C	6/E
000A 1234 53B0	8F00	016F		
000A 1234 53D0	0003	0F		

DIVIDE PACKED LONG (DPL)

Instruction Description

The dividend (first operand) is divided by the divisor (second operand) and replaced by the quotient and remainder.

Format: SS

FA	L_1	L_2	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Note: The Divide Packed Long instruction is implemented in vertical microcode (VMC) and is treated as an implicit SVL by the IMP processor. The op code is used as the index into the SVL table, as described in the section on SVLs in Chapter 6.

Operation: L_2 specifies 1 less than the length in bytes of the divisor. The divisor can contain a maximum of 31 digits and sign.

The quotient is placed leftmost in the first-operand field, and can contain a maximum of 31 digits and sign, corresponding to a maximum of 15 for L_1. The remainder is placed rightmost in the first-operand field and can contain a maximum of 31 digits and sign, corresponding to a maximum of 15 for L_2. Together, the quotient and remainder occupy the entire first-operand field.

When division by zero is attempted, a decimal zero divide exception occurs, and the operation is suppressed.

Digit codes are checked for validity; invalid codes cause a data exception, and the operation is terminated.

The dividend, divisor, and remainder are all signed integers, right-aligned in their fields. The quotient is a signed integer, but is left-aligned in its field.

Overflow: Not applicable.

Sign Code: The sign of the quotient is determined by the rules of algebra from the dividend and divisor signs. The remainder has the same sign as the dividend except that a zero quotient or a zero remainder is always positive. The processor uses the preferred signs for the quotient and remainder as follows: a positive sign is encoded as 1111 (hex F); a negative sign is encoded as 1101 (hex D).

The sign codes are checked for validity: invalid signs cause a data exception, and the operation is terminated.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The divisor and dividend fields may overlap only if their rightmost bytes coincide; improperly overlapping fields cause a data exception, and the operation is terminated.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal zero divide
- Effective address overflow

Note: L_1 specifies 1 less than the length in bytes of the dividend plus the length by which the first-operand storage area exceeds the length of the divisor. The dividend is taken from the first L_1+1 bytes of the first operand. The dividend can contain a maximum of 31 digits and sign, corresponding to a maximum of 15 for L_1. The rightmost L_2 bytes of the first operand are ignored unless they contain the divisor (due to overlapping operands). The maximum length of the first operand (L_1+L_2+2) is 32 bytes.

DPL Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
FA	4	0	3	F30	3	FB0

0 Bits 8 12 16 20 32 36 47

Assembler: DPL $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: FA40 3F30 3FB0

B_1(3) and B_2(3): 00FA 12AB C000

Storage — Before

	0/8	2/A	4/C	6/E
00FA 12AB CF30	2793	4766	2Fxx	
00FA 12AB CFB0	3D			

Storage — After

	0/8	2/A	4/C	6/E
00FA 12AB CF30	0931	1588	7D1F	
00FA 12AB CFB0	3D			

DIVIDE SHORT FLOAT (DSF)

Instruction Description

The first operand is divided by the second operand (two-operand format) or the second operand is divided by the third operand (three-operand format), and the quotient is placed in the first operand location. No remainder is saved.

Format: SS

AE	B_3	4	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: A two-operand or three-operand format may be specified. A two-operand format is used if base register 0 is specified for the third operand. The three-operand format is used if one of the base registers hex 1 through hex F is specified for the third operand.

Floating-point division uses exponent subtraction and significand division. The difference between the signed (unbiased) exponents of the dividend and the divisor operands is used as the signed exponent of the intermediate quotient when both the dividend and the divisor are normalized.

When the dividend is denormalized and the divisor is normalized and nonzero, the difference between the signed exponents of the dividend and divisor operands less 1 is used as the signed exponent of the intermediate quotient.

All dividend and divisor significand digits participate in forming the significand of the quotient. The intermediate quotient is calculated as if to infinite precision.

Normalizing the intermediate quotient is never necessary, but a right shift of one digit position can be called for, which causes the intermediate quotient significand to be shifted right one digit position.

The intermediate quotient is rounded, if necessary, according to the rounding mode specified in the task dispatching element.

When the dividend is 0 and the divisor is a finite number or infinity, the quotient is 0.

If the divisor is denormalized and the dividend is normalized and not normal 0, an invalid operand exception is recognized.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the quotient. The source operands are checked for this value in order of their specification with the masked not-a-number with the larger fraction value being provided as the quotient.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the quotient is determined by the rules of algebra. This amounts to the exclusive OR of the divisor and dividend signs and applies to quotients of 0 or infinity value as well as for normal finite results.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: All operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Floating-point zero divide
- Specification

Programming Note: The following is a summary of the results for various combinations of operations.

Quotient	**First Source (Divisor)**	**Second Source (Dividend)**
Invalid operand exception	+0 or -0	+0 or -0
Divide by zero exception	+Real number ≠ 0 or -real number ≠ 0	+0 or -0
+0	+0	+Real number ≠ 0
+0	-0	-Real number ≠ 0
-0	-0	+Real number ≠ 0
-0	+0	-Real number ≠ 0
+0 (see note)	+Real number ≠ 0	+Larger real number ≠ 0
+0 (see note)	-Real number ≠ 0	-Larger real number ≠ 0
-0 (see note)	-Real number ≠ 0	+Larger real number ≠ 0
-0	+Real number ≠ 0	-Larger real number ≠ 0
Invalid operand exception	+Infinity or -infinity	+Infinity or -infinity
+Infinity	+Infinity	+Real number ≠ 0
+Infinity	+Infinity	+0
+Infinity	-Infinity	-Real number ≠ 0
+Infinity	-Infinity	-0
-Infinity	-Infinity	+Real number ≠ 0
-Infinity	-Infinity	+0
-Infinity	+Infinity	-Real number ≠ 0
-Infinity	+Infinity	-0
+0	+Real number ≠ 0	+Infinity
+0	+0	+Infinity
+0	-Real number ≠ 0	-Infinity
+0	-0	-Infinity
-0	-Rea; number ≠ 0	+Infinity
-0	-0	+Infinity

Note: For a small value real number that does not equal 0 and a large value real number that does not equal 0, a masked floating-point underflow which yields a 0 rather than a denormalized result can occur.

Quotient	First Source (Divisor)	Second Source (Dividend)
-0	+Real number ≠ 0	-Infinity
-0	+0	-Infinity
Masked not-a-number	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-number	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operand exception	Unmasked not-a-number	Any
Invalid operand exception	Any	Unmasked not-a-number

Legend:
Any = Any floating-point value.

DSF Example

Op	B_3	E	B_1	D_1	B_2	D_2
AE	3	4	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: DSF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: AE34 4050 4060

B_3(3): 0010 0200 0070

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	4100	0000		
0010 0200 0070	BF80	0000		

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	C100	0000		
0010 0200 0060	4100	0000		
0010 0200 0070	BF80	0000		

Condition Code: Not changed.

DIVIDE WORD STORAGE (DWS)

Instruction Description

The dividend (first operand) is divided by the divisor (second operand) and replaced by the quotient and remainder.

Format: SS

ED		0	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The dividend is a 64-bit, binary value occupying 8 bytes of storage at the first-operand location. It is replaced by a 32-bit signed remainder and a 32-bit signed quotient occupying the first and second words, respectively. The divisor is a 32-bit signed integer.

When the relative magnitude of the dividend and divisor is such that the quotient cannot be expressed by a 32-bit signed integer, a binary divide exception occurs, and the operation is suppressed. This includes attempts to divide by zero.

Overflow: Not applicable.

Sign Code: The sign of the quotient is determined by the rules of algebra. The remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Both operands must start on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Binary divide
- Effective address overflow
- Specification

DWS Example

Op ED		E 0	B_1 3	D_1 EF0	B_2 3	D_2 F40	
0 Bits	8	12	16	20	32	36	47

Assembler: DWS $D_1(B_1)$, $D_2(B_2)$

Machine: ED00 3EF0 3F40

B_1(3) and B_2(3): 1C4F 3AB9 4000

Storage — Before

	0/8	2/A	4/C	6/E
1C4F 3AB9 4EF0	0000	0000	0000	A34F
				(41807)
1C4F 3AB9 4F40	FFFF	FFED		
		(−19)		

Storage — After

	0/8	2/A	4/C	6/E
1C4F 3AB9 4EF0	0000	0007	FFFF	F768
		(+7)		(−2200)
1C4F 3AB9 4F40	FFFF	FFED		
		(−19)		

EDIT PACKED DECIMAL (EDPD)

Instruction Description

The format of the source field (the second operand) is changed from packed to zoned decimal format, and is modified under control of the edit-mask field (the third operand).

Format: RS

63	L_3	B_3	D_3
0 Bits	8	16	20 31

Operation: The edited result is placed in the result field (the first operand). The address of the first significant result character is placed in base register hex F.

The first and second-operand addresses and their associated length codes are obtained implicitly from registers. Base register 14 points to the leftmost byte of the result field, and base register 15 points to the leftmost byte of the source field. Byte register 10 (r10) specifies one less than the number of bytes in the result field. Byte register 11 (r11) specifies the number of decimal digits exclusive of sign in the source field. The third-operand address (B_3, D_3) points to the leftmost byte of the edit-mask field. The length field (L_3) specifies one less than the number of bytes in the edit-mask field.

The maximum length of the source field is 31 decimal digits. If the source field length is zero, the sign is the only source data processed. The maximum length of the edit-mask field is 256 bytes, while the maximum valid length of the result field is 254 bytes.

The source field is in packed decimal format. The edit mask contains control characters and data character strings. Both the edit mask and the source fields are processed left to right. The edited result from this processing is placed in the result field left to right.

If the length of the source field is even, the high-order 4 bits of the source field are ignored and not checked for validity, all other source digits and signs are checked for validity and a data exception is indicated when invalid. Any overlapping of these fields will yield unpredictable results. After validity checking, the sign of a source field with a value of zero (or digit length of zero) is considered to be plus (1111).

There are 10 types of control characters which may be found in the edit mask: hex AA through hex B3. In addition to these control characters, if the first character of the edit-mask field is less than hex 40, the value of that character is used instead of hex AE as an end-of-string delimiter. This allows the use of characters with a value of hex AE within a string. Two of these control characters indicate the beginning of a type of field, two control characters indicate the beginning of a type of character string, and one is used to indicate the end of the character string. The other five control characters indicate that a digit from the source field should be checked and that appropriate action be taken.

There is a significance indicator used in the execution of this instruction. At the start of the execution of this instruction, this indicator is set to the *off* state. It remains in this state until a nonzero source digit is encountered in the source field, or until one of the four unconditional digits (hex AA through hex AD) or an unconditional string (hex B3) is encountered in the edit mask.

When significance is detected, the selected floating string specified by hex B1 is overlaid into the result field immediately to the left of the first significant result character.

When the significance indicator is set to the *on* state, the first significant result character has been reached. The state of the significance indicator determines whether the fill character or a digit from the source field is to be inserted into the result field for conditional digits and characters in conditional strings specified in the edit-mask field. The fill character will be a hex 40 until it is replaced by the first character following the floating string control character, hex B1.

When the indicator is in the *off* state:

- A conditional digit control character in the edit mask causes the fill character to be moved to the result field.

- A character in a conditional string in the edit mask causes the fill character to be moved to the result field.

When the indicator is in the *on* state:

- A conditional digit control character in the edit mask causes a source digit to be moved to the result field.

- A character in a conditional string in the edit mask is moved to the result field.

The control characters found in the edit-mask field are:

- End-of-string character (EOSC)

  This control character indicates the end of a string of characters and must be present, even if the string is null. The value of this character is either that of the first character of the edit-mask field, if that character is less than hex 40, or it is hex AE.

- Start-of-static-field control character

  – Hex AF: This control character indicates the start of a static field. A static field is used to indicate that one of two mask character strings immediately following this character is to be inserted into the result field, depending upon the algebraic sign of the source field. The string to be inserted into the result field, if the sign in the source field is positive, is the first string in the field. The second string in this field is the string to be inserted into the result field if the sign in the source field is negative.

  – Static field format:

    Hex AF . . . positive string . . . EOSC . . . negative string . . . EOSC (end-of-string control character)

- Start-of-floating-string control character

  – Hex B1: This control character indicates the start of a floating string field. The first character of the field is used as the fill character.

    Following the fill character are two optional strings, delimited by the EOSC. If the sign in the source field is positive, the first string in the field is to be inserted into the result field. The second string is to be inserted into the result field if the sign in the source field is negative.

    The string selected for insertion into the result field appears immediately to the left of the first significant result character, and is called a floating string. If significance is never set, neither string is placed in the result field. Conditional source digit positions (hex B2 control characters) must be provided in the edit mask immediately following the hex B1 field to accommodate the larger of the two floating strings, or a length conformance exception will be presented. For each of these hex B2 control characters, the fill character is inserted into the result field and source digits are not consumed.

  – Floating-string field format:

    Hex B1 fill character . . . positive string . . . EOSC negative string . . . EOSC, hex B2 . . .

- Start-of-conditional-string control character

  – Hex B0: This control character indicates the start of a conditional string. The string contains any character and is delimited by EOSC (the end-of-string control character). This string, or fill characters replacing it, is inserted into the result field based on the state of the significance indicator. When the significance indicator is in the *off* state, a fill character for every character in the conditional string is placed in the result field. When the significance indicator is in the *on* state, the characters in the conditional string are placed in the result field.

  – Conditional string format:

    Hex B0 . . . conditional string . . . EOSC

- String-of-unconditional-string control character:

  - Hex B3: This control character turns on the significance indicator and indicates the start of an unconditional strıng. This string consists of any character and is delimited by the EOSC (end-of-string control character). This string is unconditionally inserted into the result field regardless of the state of the significance indicator.

    If the significance indicator is *off* when a hex B3 control character is encountered, the appropriate floating string is inserted (overlaid) into the result field prior to (to the left of) the hex B3 unconditional string.

  - Unconditional string format:

    Hex B3 . . . unconditional string . . . EOSC

  Control characters that correspond to digits in the source field

  - Hex B2: This control character specifies that the corresponding source field digit or floating string (hex B1) field digit is considered a conditional digit. This means that either the source digit or the fill character is placed in the result field based on the state of the significance indicator. When the significance indicator is in the *off* state, the fill character is placed in the result field. When the significance indicator is in the *on* state, the source digit is placed in the result field. When a source digit is moved to the result field, the hex F zone receives the source digit. When significance (that is, a nonzero source digit) is detected, the floating string is placed to the left of significance.

    Control characters hex AA, hex AB, hex AC, and hex AD will independently turn on the significance indicator. If the significance indicator is turned off when one of these control characters is encountered, the appropriate floating string is inserted (overlaid) into the result field prior to (to the left of) the source digit.

  - Hex AA: This control character specifies that the corresponding source field digit is unconditionally placed in the result field. The zone portion is set to a hex F. The source digit is placed in the four low-order result bits.

  - Hex AB: This control character specifies that the corresponding source field digit is unconditionally placed in the result field. The zone portion of the digit is set to the preferred positive sign bits 1111 (hex F) if the sign of the source field is positive, or to the negative sign bits 1101 (hex D) if the sign of the source field is negative.

  - Hex AC: This control character specifies that the corresponding source field is unconditionally placed in the result field. The zone portion of the digit is set to the preferred positive sign bits 1111 (hex F) only if the sign of the source field is positive. If it is not positive, the source sign field is moved to the result zone field.

  - Hex AD: This control character specifies that the corresponding source field digit is unconditionally placed in the result field. The zone portion of the digit is set to the preferred negative sign 1101 (hex D) only if the sign of the source field is negative. If it is not negative, the source field sign will be moved to the zone position of the result byte.

The *Table of Valid Edit Conditions and Results* provides an overview of the result obtained with the valid edit conditions and sequences.

Conditions				Results
Mask Character (Hex)	**Significance Indicator Before/After**	**Source Digit**	**Source Sign**	**Result Character**
AF	NI[1]/NC[2]	NI	Positive	Positive string inserted
	NI/NC	NI	Negative	Negative string inserted
AA	Off/On	0-9	Positive	Positive floating string overlaid; hex F, source digit
	Off/On	0-9	Negative	Negative floating string overlaid; hex F, source digit
	On/On	0-9	NI	Hex F, source digit
AB	Off/On	0-9	Positive	Positive floating string overlaid; hex F, source digit
	Off/On	0-9	Negative	Negative floating string overlaid; hex D, source digit
	On/On	0-9	Positive	Hex F, source digit
	On/On	0-9	Negative	Hex D, source digit
AC	Off/On	0-9	Positive	Positive floating string overlaid; hex F, source digit
	Off/On	0-9	Negative	Negative floating string overlaid; sign, source digit
	On/On	0-9	Positive	Hex F, source digit
	On/On	0-9	Negative	Sign, source digit
AD	Off/On	0-9	Positive	Positive floating string overlaid; sign, source digit
	Off/On	0-9	Negative	Negative floating string overlaid; hex D, source digit
	On/On	0-9	Positive	Sign, source digit
	On/On	0-9	Negative	Hex D, source digit
B0	Off/Off	NI	NI	Insert fill character for each hex B0 string character
	On/On	NI	NI	Insert hex B0 character string

[1] NI: Not important

[2] NC: No change

Conditions				Results
Mask Character (Hex)	Significance Indicator Before/After	Source Digit	Source Sign	Result Character
B1 (including necessary B2s)	Off/NC[2]	NI[1]	NI	Insert the fill character for each hex B2 that corresponds to a character in the longer of the two floating strings
B2 (not for a B1 field)	Off/Off	0	NI	Insert fill character
	Off/On	1-9	Positive	Overlay positive floating string and insert hex F, source digit
	Off/On	1-9	Negative	Overlay negative floating string and insert hex F, source digit
	On/On	0-9	NI	Hex F, source digit
B3	Off/On	NI	Positive	Overlay positive floating string and insert hex B3 character string
	Off/On	NI	Negative	Overlay negative floating string and insert hex B3 character string
	On/On	NI	NI	Insert hex B3 string

[1] NI: Not important
[2] NC: No change

Overflow: Not applicable.

Sign Code: See *Operation.*

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Unpredictable results will occur if the first, second, or third operands are overlapped. The source field length must not exceed 31 decimal digits; otherwise a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Data
- Edit digit count
- Edit mask syntax
- Effective address overflow
- Length conformance
- Specification

Programming Notes:

1. A source length equal to zero implies that only the sign is processed (examined).
2. Base registers 14 and 15 and byte registers 10 and 11 are not altered by this instruction.
3. The result field may be partially modified if EDPD is interrupted before processing has completed. In this case, the EDPD operation will be restarted upon completion of interrupt processing.
4. Any character is a valid fill character, including hex AE.
5. Hex AF, hex B1, hex B0, and hex B3 strings must be terminated by the EOSC, even if they are null strings.
6. If a floating string (hex B1) field has not been encountered (specified) when the significance indicator is turned on, the floating string is considered to be a null string and is therefore not used to overlay into the result field.

The following is a truth table which indicates the valid ordering of control characters in an edit-mask field.

Control Character Y

Hex AA, AB, AC, AD

			AF	B0	B1	B2	B3
		0	0	2	2	2	0
	AF	0	0	0	0	0	0
Control Character X	B0	1	0	0	2	0	1
	B1	1	0	1	3	1	1
	B2	1	0	0	2	0	1
	B3	0	0	2	2	2	0

Condition	**Definition**
0	Both X and Y can appear in the edit-mask field in either order.
1	Y cannot precede X.
2	X cannot precede Y.
3	Both control characters (two hex B1s) cannot appear in the edit mask field.

Note: Violation of the above rules will result in an edit-mask syntax program exception.

EDPD Example

Op	L_3	B_3	D_3
63	20	4	342

0 Bits 8 16 20 31

Assembler: EDPD $D_3(L_3,B_3)$

Machine: 6320 4342

B_3(4):	011A	3247	0000	(Base register for mask)
B(14):	02BC	4431	0680	(Address of result)
B(15):	02BC	86AA	B012	(Address of source)
r(10):			10	(Length of result-1)
r(11):			0B	(Number of digits in source field)
L_3:	21			(Number of bytes in edit-mask field -1)

Storage — Before

	0/8	2/A	4/C	6/E
011A 3247 0342 (Edit mask)		B140	5BAE	5BAE
	B2B2	B2B2	B06B	AEB2
	B2B2	B06B	AEB2	B2B2
	B34B	AEAA	AAAF	4040
	AEC4	C2AE		
02BC 4431 0680 (Result field)	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
02BC 86AA B012 (Source field)		0000	1234	567D

Storage — After

	0/8	2/A	4/C	6/E
011A 3247 0342 (Edit mask)		B140	5BAE	5BAE
	B2B2	B2B0	6BAE	B2B2
	B2B0	6BAE	B2B2	B2B3
	4BAE	AAAA	AF40	40AE
	C4C2	AE		
02BC 4431 0680 (Result field)	4040	4040	5BF1	F26B
	F3F4	F54B	F6F7	C4C2
02BC 86AA B012 (Source field)		0000	1234	567D

Result will print as: b̸b̸b̸b̸b̸$12,345.67DB

Edit Mask Characters Source is Negative	Source Digit	Significance Indicator (Before/After)	Character(s) Placed in Result	Description
B1405BAE5BAEB2	Not Used	Reset/reset	40	Floating string format specifies hex 40 fill character and floating $ (hex 5B)
B2	0	Reset/reset	40	Conditional digit
B2	0	Reset/reset	40	Conditional digit
B2	0	Reset/reset	40	Conditional digit
B06BAE	Not Used	Reset/reset	40	Conditional string
B2	0	Reset/reset	40	Conditional digit
B2	1	Reset/set	5BF1	Conditional digit negative floating string overlays previous fill character (5B overlays 40)
B2	2	Set/set	F2	Conditional digit
B06BAE	Not Used	Set/Set	6B	Conditional string
B2	3	Set/set	F3	Conditional digit
B2	4	Set/set	F4	Conditional digit
B2	5	Set/set	F5	Conditional digit
B34BAE	Not Used	Set/set	4B	Unconditional string
AA	6	Set/set	F6	Unconditional digit
AA	7	Set/set	F7	Unconditional digit
AF4040AEC4C2AE	Not Used	Set/set	C4C2	Static field-negative string to result

ENABLE TASK DISPATCHING (ETD)

Instruction Description

This instruction enables task dispatching by setting the task dispatcher mask (byte hex 22, bit 7 of LSR [local storage register]) and invoking the task dispatcher.

Format: RR

0D		0

0 Bits 8 12 15

Operation: Control may not be immediately returned to the next sequential instruction of the function issuing the Enable Task Dispatching instruction if a higher priority task TDE (task dispatching element) is on the TDQ (task dispatching queue).

This instruction also restarts the task interval timer if no task switch occurs and the current task is timed. If a task switch occurs, the task dispatcher sets the task interval timer depending upon whether the new task is timed or untimed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

Programming Note: The Enable Task Dispatching instruction does not check whether or not the task dispatcher mask is already set prior to the execution of this instruction.

ETD Example

Op 0D		E 0

0 Bits 8 12 15

Assembler: ETD

Machine: 0D00

The following bit is set by this instruction: LSR byte 22 (FLG0), bit 7 (flag task switch blocked).

This page is intentionally left blank.

ENQUEUE MESSAGE (EQM)

Instruction Description

The SRM (send/receive message) addressed by B_1 is checked for validity and, if valid, is enqueued to the message list of the send/receive queue designated by the second operand.

Format: RS

6C	B_1	I_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The enqueuing method is designated by the I-field. The message list is searched, in sequence, beginning with the first message. The new message (the first operand) is enqueued before the first message that satisifies the search type. If the list is empty, the new message is enqueued first. If the search type is not satisified, the new message is enqueued last. Search keys begin in byte 8 of the SRM, have a length specified in the queue header, and are treated as unsigned binary data.

I-Field	**Search Type**
Bit 12	Searched Message Key = The first operand Message Key
Bit 13	Searched Message Key < The first operand Message Key
Bit 14	Searched Message Key > The first operand Message Key
Bit 15	Not used

The search type is the logical OR of the I-bits specified. Therefore, I = binary 000x results in enqueue last and I = binary 111x in enqueue first.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRM descriptor
- Descriptor access: Monitored SRQ descriptor
- Effective address overflow
- Invalid descriptor
- Specification

Note: The key length specification in the queue header is key length minus 1. Therefore, if enqueue first or enqueue last is specified, the key/text portion of the SRM must be at least 1 byte long. Also, no check is made for a page crossing in the key field, if enqueue first or enqueue last is specified.

EQM Example

Op	B_1	I_3	B_2	D_2
6C	4	8	5	000

0 Bits 8 12 16 20 31

Assembler: EQM B_1, $D_2(B_2)$, I_3

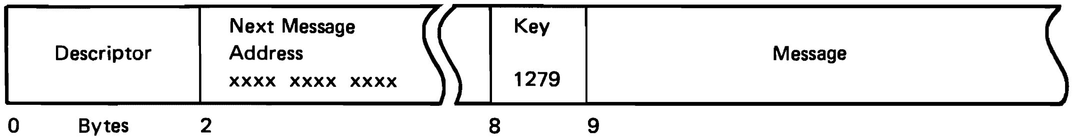
Machine: 6C48 5000

$B_1(4)$: 21A0 1240 0000

$B_2(5)$: 0020 32A0 0000

SRM

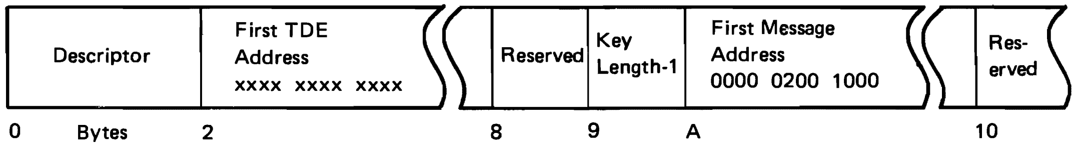
21A0 1240 0000



Before

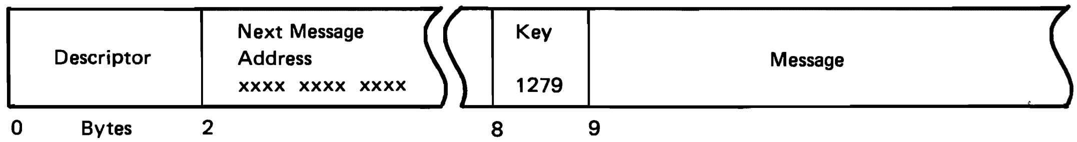
SRQ

0020 32A0 0000


Descriptor
First TDE Address xxxx xxxx xxxx
Reserved
Key Length-1
First Message Address 0000 0200 1000
Res- erved
0 Bytes 2
8
9
A
10

SRM

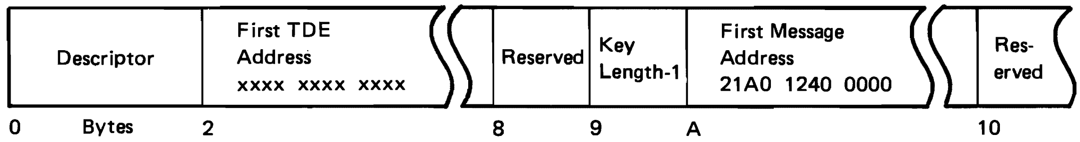
0000 0200 1000



After

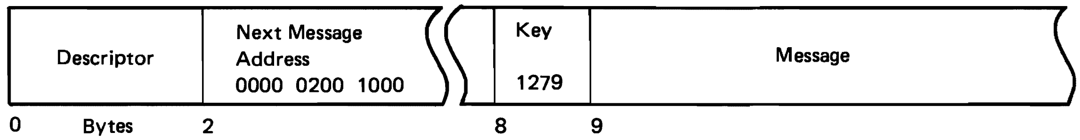
SRQ

0020 32A0 0000


Descriptor
First TDE Address xxxx xxxx xxxx
Reserved
Key Length-1
First Message Address 21A0 1240 0000
Res- erved
0 Bytes 2
8
9
A
10

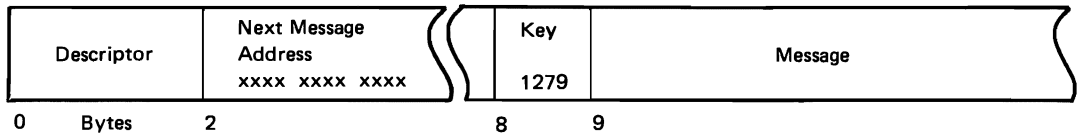
SRM

21A0 1240 0000



SRM

0000 0200 1000



ENQUEUE TASK DISPATCHING ELEMENT (EQTDE)

Instruction Description

The TDE (task dispatching element) addressed by B_1 is checked for validity and, if valid, is enqueued to the TDQ (task dispatching queue), SRQ (send/receive queue) wait list, or SRC (send/receive counter) wait list designated by the second operand.

Format: RS

6D	B_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: Enqueuing is in key sequence; low key first, last within key value. TDE bytes hex 16-1B, the address of the current queue, are set to the second-operand address.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRQ descriptor
- Descriptor access: Monitored TDE descriptor (if the second operand is an SRQ)
- Effective address overflow
- Invalid descriptor
- Specification

EQTDE Example

Op	B_1	E	B_2	D_2
6D	3	0	4	350

0 Bits 8 12 16 20 31

Assembler: EQTDE B_1, D_2(B_2)

Machine: 6D30 4350

B_1(3): 13A2 1442 0550

B_2(4): 21A3 A983 0000

TDE

13A2 1442 0550

Descriptor	Next TDE Address xxxx xxxx xxxx	Priority 0000 0123
0 Bytes	2	8 C

Before

TDQ

21A3 A983 0350

Descriptor	First TDE Address 0000 0300 4000
0 Bytes	2 8

TDE

0000 0300 4000

Descriptor	Next TDE Address xxxx xxxx xxxx	Priority 0000 0124
0 Bytes	2	8 C

After

TDQ

21A3 A983 0350

Descriptor	First TDE Address 13A2 1442 0550
0 Bytes	2 8

TDE

13A2 1442 0550

Descriptor	Next TDE Address 0000 0300 4000	Priority 0000 0123
0 Bytes	2	8 C

TDE

0000 0300 4000

Descriptor	Next TDE Address xxxx xxxx xxxx	Priority 0000 0124
0 Bytes	2	8 C

EXAMINE PRIMARY DIRECTORY ENTRY (EPDE)

Instruction Description

The primary directory entry pointed to by the first operand is examined to determine if the frame with which the entry is associated can be reused. The result is returned via condition code.

Format: SI

83		3	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The primary directory entry is first checked for validity. The diagram on the next page outlines the operation of the EPDE instruction

The primary directory entry is identified by the first operand, which occupies 2 bytes in storage. Bits 0-15 of the first operand are used as the primary directory index value. These bits are shifted left 4 bits to convert them from an index to an offset. Bits 12-15 become zeros.

The high-order 4 bits of the primary directory index identified by the first operand are not used.

Overflow and Sign Code: Not applicable.

Condition Code:

0 The page is not pinned, but it does not satisfy the criteria for reuse.
1 The page is not pinned, and it satisfies the criteria for reuse. It has been removed.
2 The page is pinned either by a user or storage management.
3 The page satisfies the criteria for reuse, but the change bit is on.

Carry: Not applicable.

Boundary Requirements: The first operand must be halfword aligned; otherwise, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

EPDE Example

Op		E	B_1	D_1
83		3	F	002

0 Bits 8 12 16 20 31

Assembler: EPDE $D_1(B_1)$

Machine: 8303 F002

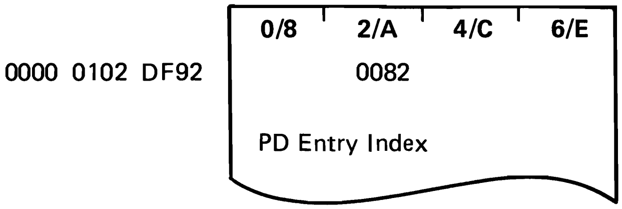
Before and After

B_1(F): 0000 0102 DF90

Primary Directory Address: 0000 0103 0000

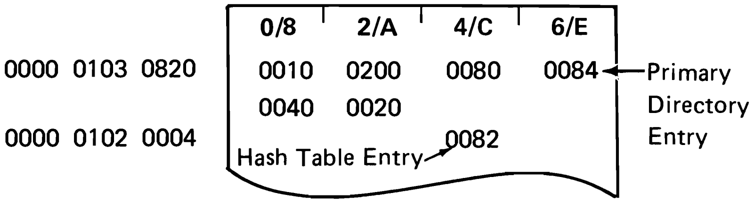
Hash Table Address: 0000 0102 0000

Storage — Before and After



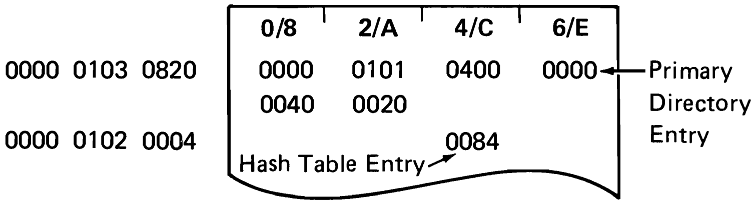
	0/8	2/A	4/C	6/E
0000 0102 DF92		0082		

Storage — Before



	0/8	2/A	4/C	6/E
0000 0103 0820	0010	0200	0080	0084
	0040	0020		
0000 0102 0004			0082	

Storage — After



	0/8	2/A	4/C	6/E
0000 0103 0820	0000	0101	0400	0000
	0040	0020		
0000 0102 0004			0084	

	Before	After
Condition Code:	x	1

Operation Diagram:

```
IF the PD entry is valid (bit 40=1),
    THEN IF the page is not pinned (bits 64-71 and bits 75-79=0),
        THEN IF the page has not been referenced (bit 41=0),
            THEN IF the purge indicator (bit 73) is a 1,

                            -or-

                    the usage code (bit 90) is a 1,
            THEN IF the page has been changed (bit 42=1),
                    THEN the condition code is set to 3.

                ELSE the PD I/O use bit (bit 44) is checked. If the page is being used by I/O (bit 44=1), the
                page is removed from the I/O resolved address registers. The PD entry virtual address is
                hashed and the chain is searched. If the entry is not found on this chain, a specification
                exception is recognized and the operation is terminated. If the entry is on the chain, the SID
                and PID entries (bit 0-39) are updated as follows: Bits 0-23 are forced to hex 00 0001; bits
                24-39 are updated by shifting the index of the PD entry left by 1, inserting a 0 in the vacated
                low-order bit position, and storing the result in bits 24-39. Bits 40-63 are forced to zeros. The
                entry is then removed from the PD chain and the condition code is set to 1.

            ELSE the purge bit is set ON (bit 73=1), the condition code is set to 0, and the instruction is
                terminated.

        ELSE the purge indicator (bit 73) and reference bit (bit 41) are set to binary 0.

        IF the directory entry is for a V=R address,
            THEN a specification exception is recognized and the operation is terminated.
        ELSE if the lookaside buffer contains an entry for the page associated with the directory entry, the
            change bit in the loookaside buffer entry is ORed into the change bit (bit 42) of the PD entry,
            then removed from the lookaside buffer. The condition code is set to 0.

    ELSE the condition code is set to 2.

ELSE
    IF the page is not pinned (bits 75-79=0) by storage management.
        THEN IF the virtual address is V=R,
                THEN the condition code is set to 1.
             ELSE the condition code is set to 0.
     ELSE the condition code is set to 2.
```

EXCLUSIVE OR BYTE (XB)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand location.

Format: RS

79	r_1	2	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result =	0
1	Result ≠	0
2	--	
3	--	

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

XB Example

Op	r_1	E	B_2	D_2
79	7	2	3	310

0 Bits 8 12 16 20 31

Assembler: XB r_1, $D_2(B_2)$

Machine: 7972 3310

B_2(3): 1131 1132 2000

Storage — Before and After

	0/8	2/A	4/C	6/E
1131 1132 2310	FA			

	Before	After
r_1(7):	AF	55
Condition Code:	x	1

EXCLUSIVE OR BYTE IMMEDIATE (XBI)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand location.

Format: SI

9A	I_2	B_1	D_1
0 Bits	8	16	20 31

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result =	0
1	Result ≠	0
2	--	
3	--	

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

XBI Example

Op 9A	I_2 32	B_1 3	D_1 980
0 Bits	8	16	20 31

Assembler: XBI $D_1(B_1)$, I_2

Machine: 9A32 3980

$B_1(3)$: 0000 4250 A000

Storage — Before

	0/8	2/A	4/C	6/E
0000 4250 A980	CE			

Storage — After

	0/8	2/A	4/C	6/E
0000 4250 A980	FC			

	Before	**After**
Condition Code:	x	1

EXCLUSIVE OR BYTE REGISTER (XBR)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand register.

Format: RR

1A	r_1	r_2

0 Bits 8 12 15

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

XBR Example

Op	r_1	r_2
1A	6	7

0 Bits 8 12 15

Assembler: XBR r_1, r_2

Machine: 1A67

	Before	**After**
r_1 (6):	2D	12
r_2 (7):	3F	3F
Condition Code:	x	1

EXCLUSIVE OR BYTE REGISTER IMMEDIATE (XBRI)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand register.

Format: RI

4A	r_1	0	I_2	

0 Bits 8 12 16 24 31

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

XBRI Example

Op	r_1	E	I_2	
4A	3	0	B9	

0 Bits 8 12 16 24 31

Assembler: XBRI r_1, I_2

Machine: 4A30 B900

	Before	After
r_1(3):	42	FB
Condition Code:	x	1

EXCLUSIVE OR CHARACTER (XC)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand location.

Format: SS

CA	L_1	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32 36	47

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Each operand field is processed left to right.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

XC Example

Op CA	L_1 03	B_1 4	D_1 800	B_2 4	D_2 810
0 Bits	8	16	20	32 36	47

Assembler: XC $D_1(L_1B_1)$, $D_2(B_2)$

Machine: CA03 4800 4810

$B_1(4)$ and $B_2(4)$: 1801 1802 1000

Storage — Before

	0/8	2/A	4/C	6/E
1801 1802 1800	C5C6	C7C8		
1801 1802 1810	C1C2	C3C4		

Storage — After

	0/8	2/A	4/C	6/E
1801 1802 1800	0404	040C		
1801 1802 1810	C1C2	C3C4		

	Before	After
Condition Code:	x	1

EXCLUSIVE OR HALFWORD (XH)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand register.

Format: RS

80	R_1	5	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	1
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

XH Example

Op	R_1	E	B_2	D_2
80	8	5	3	330

0 Bits 8 12 16 20 31

Assembler: XH R_1, $D_2(B_2)$

Machine: 8085 3330

B_2(3): 0633 0634 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
0633 0634 0330	0636			

	Before	After
R_1(8):	0632	0004
Condition Code:	x	1

EXCLUSIVE OR HALFWORD REGISTER (XHR)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand register.

Format: RR

2A	R_1	R_2

0 Bits 8 12 15

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

XHR Example

Op 2A	**R_1** 9	**R_2** A

0 Bits 8 12 15

Assembler: XHR R_1, R_2

Machine: 2A9A

	Before	**After**
R_1 (9):	BB76	44FE
R_2 (A):	FF88	FF88
Condition Code:	x	1

EXCLUSIVE OR HALFWORD REGISTER IMMEDIATE (XHRI)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand register.

Format: RI

5A	R_1	0	I_2

0 Bits 8 12 16 31

Operation: Operands are treated as logical quantities, and the connective exclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit positions in the two operands are unlike; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	=	0
1	Result	≠	0
2	--		
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

XHRI Example

Op	**R_1**	**E**	**I_2**
5A	C	0	F0F0

0 Bits 8 12 16 31

Assembler: XHRI R_1, I_2

Machine: 5AC0 F0F0

	Before	**After**
R_1(C):	A2A2	5252
Condition Code:	x	1

This page is intentionally left blank.

EXECUTE (EX)

Instruction Description

The single instruction at the second-operand address is modified by the contents of the byte register specified by r_1, and the resulting instruction is executed.

Format: RS

7E	r_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: Bits 8-15 of the instruction at the second-operand address are ORed with the bits of the register specified by r_1, except when register zero is specified, which indicates that no modification takes place. The ORing does not change either the contents of the register or the instruction in storage, and it is effective only for the interpretation of the instruction to be executed.

The execution and exception handling of the subject instruction are exactly as if the subject instruction were obtained in normal sequential operation, except for the instruction address and the instruction length code. The instruction address is increased by the length of the Execute instruction in order to form the updated instruction address.

When the subject instruction is another Execute instruction, an execute exception occurs and the operation is suppressed. The second-operand address must be even; otherwise a specification exception occurs.

Overflow and Sign Code: Not applicable.

Condition Code: The code may be set by the subject instruction.

Carry: Not applicable.

Boundary Requirements: The second operand must begin on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Execute
- Specification

EX Example

Op	r_1	E	B_2	D_2
7E	1	0	A	000

0 Bits 8 12 16 20 31

Assembler: EX r_1, $D_2(B_2)$

Machine: 7E10 A000

B_2(A): 0000 3820 0000

	Before	After
r_1(1):	99	99

Storage — Before and After

	0/8	2/A	4/C	6/E
0000 3820 0000	9B66	C003		

Op	I_2	B_1	D_1
9B	66	C	003

0 Bits 8 16 20 31

Assembler: MVBI $D_1(B_1)$, I_2

Machine: 9B66 C003

B_1(C): 0000 8916 0000

Bits 8–15	66
Bits r_1(1)	99
Result	FF

The MVBI instruction is executed as if it were:

Op	I_2	B_1	D_1
9B	FF	C	003

0 Bits 8 16 20 31

If the EX instruction is located at 0000 3820 0000, the next instruction to be executed is at address 0000 3820 0004.

EXTRACT TAGS (EXTAG)

Instruction Description

This instruction saves the tags when the page is written to auxiliary storage. The pointer tag bits associated with the block of storage addressed by the second operand are stored in the halfword addressed by the first operand.

Format: SS

A4	L_2	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32	36 47

Operation: Tags are extracted from the second operand, one bit for each quadword, and placed in the first operand. The second operand remains unchanged after the operation. L_2 applies only to the number of bytes of the second operand minus 1.

The first operand is a halfword in storage with each bit (starting with the leftmost bit) containing the tagged indication for each of the quadwords of the second operand. Any partial quadword at the end of the second operand would reset the corresponding bit of the first operand. Any unused bits of the first operand are reset. The quadword containing the first operand is untagged by this operation.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand must start on a halfword boundary and the second operand must start on a quadword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

EXTAG Example

The example shows 2 EXTAG instructions being used to extract the tags from PAGE1 and put them in the two halfwords at FIELD1 and FIELD1+2.

Op	L_2	B_1	D_1	B_2	D_2
A4	FF	4	000	5	000

0 Bits 8 16 20 32 36 47

Assembler: EXTAG $D_1(B_1),D_2(L_2,B_2)$

Machine: A4FF 4000 5000

Op	L_2	B_1	D_1	B_2	D_2
A4	FF	4	002	5	100

0 Bits 8 16 20 32 36 47

Machine: A4FF 4002 5100

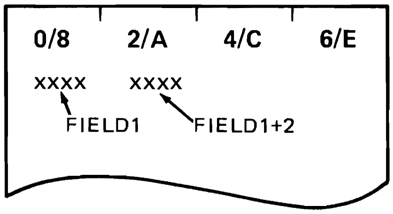
B_1(4): 020A B76C 0000

B_2(5): 020A B62A 0000 (This is the address of PAGE1.)

Assume that PAGE1 contains a tagged pointer in the quadwords at addresses 020A B62A 0030 and 020A B62A 0040 but no other pointers exist on the page.

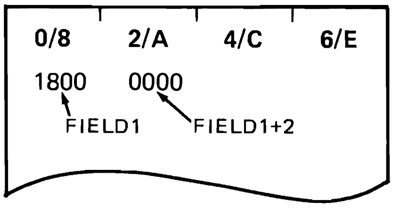
Storage — Before

020A B76C 0000



Storage — After

020A B76C 0000



FREE HOLD RECORD (FHR)

Instruction Description

This instruction is designed to be a continuation of a Free Hold Record First instruction that was interrupted. Therefore, the execution is almost identical except for some initial conditions. The FHR instruction assumes that the hold record chain has been marked *busy* by a previous Free Hold Record First instruction, and instead of using an object address and hash table to locate the first hold record to check, the FHR instruction assumes the address of a hold record is loaded in a register.

The FHR instruction also checks for pending interruptions during execution. When the FHR instruction terminates, it is necessary to check the condition code to determine if the instruction was interrupted.

Format: RS

57	B_1	B_3	B_2	D_2	
0 Bits	8	12	16	20	31

Operation: The first-operand hold records are checked to see if the second-operand free request can be granted. The outcome of this check is as follows: to grant the free request, to signal an error condition, or to indicate that further hold records on the chain must be checked. In the first and last cases, the first-operand base register is loaded with the address of the appropriate hold record.

A 4096-byte HHT (hold hash table) is accessed. Its address is given in bytes hex 8A-8F of the CAT. This HHT address is set by IMPL to point to the first byte in the page.

This instruction functions the same as the Free Hold Record First instruction except for the following differences. The address of the last hold record previously checked (by a prior execution of a FHRF or a FHR instruction) is found in the first-operand base register. Instead of hashing the third object address and going through HHT, the address of the first hold record address is calculated by multiplying bytes hex A and B of the present hold record by 16 and then concatenating the 20 bit result with the 28 high order bits of the available hold record entry in bytes 92 through 97 of the control address table. This hold record is not the first on the chain, so an end of chain exception is not invoked by an empty chain. Also, its busy flag is not checked; thus, a descriptor access busy exception cannot occur in this instruction.

Because this instruction checks groups of hold records starting within the object hold chain, it does not set the chain busy flag in the first hold record of the chain when the condition code 3 exit is used (the flag is assumed to have been set by the execution of a preceding Free Hold Record First instruction). However, when the condition code 0 exit is used, it must go back to the first hold record in the chain and reset the record's busy flat (bit 5 of the first byte).

Finally, when condition code 3 is set, the address loaded into the first-operand base register points to the alst hold record checked. This is similar to the Free Hold Record First or Grand Hold Record instruction, which loads the address of the next hold record to be checked.

Overflow and Sign Code: Not applicable.

Condition Code:

0 Result free was allowed.
1 --
2 --
3 Continue searching hold chain.

Carry: Not applicable.

Boundary Requirements: The second-operand hold request block must be halfword aligned and the first-operand hold record must be quadword aligned. If either requirement is violated, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Chain conflict
- Effective address overflow
- End of chain
- Second chain search
- Specification

Programming Note: If a program exception occurs, the busy flag is not reset.

FHR Example

See the Free Hold Record First instruction example.

57	B	C	D	006
0 Bits	8	12	16	20 31

Assembler: FHR B_1,B_3,D_2(B_2)

Machine: 57BC D006

FREE HOLD RECORD FIRST (FHRF)

Instruction Description

The FHRF instruction removes entries from a list of *hold records*, which contain a 6-byte object address, a *hold* byte, and the identification of the TDE (task dispatching element) that requested the hold on the object. The list of hold records is searched until a record matching the input data is found. Then, if the maching entry is not monitored, it is removed. If no match can be found, an end-of-chain exception occurs.

The starting point for the search of the hold records is found by hashing the 6-byte object address. An entry in a hash table points to the first hold record to be checked. Then each hold record contains the address of the next hold record to be checked. As each hold record is checked, its 6-byte address is placed in a register. The last entry on a chain is indicated by a flag bit in the hold record. When a hold record is removed from the chain, it is placed on the chain of available hold records. The address of the hash table and the start of the chain are located in the control address table. While this instruction is executing, it periodically checks for pending interruptions.

Format: RS

47	B_1	B_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The first hold records in the hold record chain of the third-operand object are checked for one of the following purposes: to free a hold record, to signal that no matching record can be found to be freed, or to indicate that further hold records on the chain must be checked. In the first case, the first-operand base register is loaded with the address of the freed hold record. In the third case, the first-operand base register is loaded with the address of the last checked hold record in the chain.

A 4096-byte HHT (hold hash table) is accessed. Its address is given in bytes hex 8A-8F of the control address table. This HHT address is set by IMPL to point to the first byte in the page.

The third-operand register contains an object address. This 6-byte effective address is hashed to create a 2-byte index into a hold hash table. The 2-byte hash table entry selected (when multiplied by 16 and concatenated to the right of the 28 high-order segment bits of the available hold record address) addresses the first hold record in the chain for the third-operand object address and its hash synonyms. If the chain is empty (contains no hold records), the hold hash table entry is all zeros. In this case, the free requested by the second-operand cannot be granted. An end-of-chain program exception is recognized, the first-operand register contents are unchanged, and the operation is nullified.

If the hold hash table entry is not all zeros, the addressed hold record is accessed. The 2-byte hold hash table entry is multiplied by 16 and concatenated to the right of the high-order 28 bits of the AHR (available hold record) address. This new value points to the first hold record on the chain (the start of the hold record area). The chain busy flag (bit 5 of the first byte) of the hold record is checked. If the chain busy flag is set, the first-operand register is unchanged, a descriptor access busy exception occurs and the operation is nullified. If the chain busy flag is not set, hold records are checked for a matching record; the figure below indicates the number of records to be checked. A matching record or end-of-chain flag set in any hold record prohibits checking additional hold records. If no matching records are found in the first or previous check, a check is made for pending external interrupts. If none are pending, additional groups of records are checked. If an interrupt is pending, it is handled the same as in a page fault. The instruction finishes as follows.

	Records Checked	
Models	**Initial**	**Additional**
3, 4, and 5	9	13
6, 7, and 8	14	20

The condition code is set to 3, the first-operand register is loaded with the address of the last hold record checked (note this difference from operation of the Grant Hold Record First instruction), the chain is busy flag (bit 5 of the first byte) in the first hold record for this object chain is set, and the operation is completed.

For a matching hold record to be found, three fields in the hold record are checked. The 6-byte object address field (bytes 2-7) must match the third-operand register contents; the 1-byte hold field (byte 1) must match the second-operand hold field (byte 1); and the 2-byte TDE identifier field (bytes 8-9) must match the TDE identifier located in bytes hex 94-95 of the current TDE. If all three match, this hold record can be removed from the object hold chain and returned to the available hold record chain. See *Freeing a Hold Record*, later in this instruction description.

Primary Chain Search

If the object address does not match, the EOC (end-of-chain) flag (byte 0, bit 6) is checked. If the EOC flag is set, no matching hold record can be found. The first operand base register is unchanged, an EOC exception occurs, and the operation is nullified. If the EOC flag is reset, the processor checks for pending external interrupts. If there are no pending interrupts, searching continues with the next hold record on the primary chain (pointed to by the index in bytes 10 and 11 of the current hold record).

If an object address matches, the other two checks are made (on the hold record and the TDE identifier). If either of these does not match, the head of the secondary chain bit (byte 0, bit 0) is checked. If this bit is reset, no secondary chain exists and no other holds could have been placed on this object. The first operand base register is unchanged, an EOC exception occurs, and the operation is nullified. If the head of secondary chain bit is set, the secondary chain must be searched.

Secondary Chain Search

The secondary chain is searched in the same manner as the primary chain. If a compare is not found, the end-of-secondary chain flag bit (hold record bytes C and D) is checked. If this is the last hold record on the secondary chain, no matching hold record can be found. The first operand base register is not changed, an EOC exception occurs, and the operation is nullified. If the current hold record is not the EOC, a check is made for pending external interrupts. If none are pending, the secondary chain search continues with the next hold record pointed to by bytes C and D.

Freeing a Hold Record

If the monitor flag is set, the first operand base register is loaded with the address of the monitored hold record, the chain busy flag is set, a chain conflict exception occurs, and the operation is nullified. See *Programming Note.* If the monitor flag is reset, the matching hold record is freed, the condition code is set to 0, and the operation is completed.

Programming Note: Two other values are passed in the TDE. If the monitored hold record is at the top of chain, the TDE contains the hash table entry offset for this chain. If the monitored hold record is not at the top of the chain, the TDE contains the index of the hold record just prior to the monitored hold record.

To be freed, a matching hold record must be removed from the object hold chain and returned to the available hold record chain with some of its fields initialized and both chains updated to reflect the changes. The flag byte, the TDE identifier field, and the second chain pointer and cumulative hold field of the freed record are reset. The address of the AHR (available hold record) chain from the control address table is converted to a hold record index and the resultant 2-byte value is loaded into the object chain field of the freed record. The address of the freed record is loaded into the control address table AHR chain entry. The value in the current TDE's hold count field is decreased by 1.

Freeing from the Secondary Chain

If this is the last hold record on the secondary chain, which is indicated by secondary chain flag, the previous hold record has its secondary chain field set to zero. If this hold record is the head of the secondary chain, the head of the secondary chain flag is reset, and the cumulative hold fields are updated. See *Cumulative Hold Updates* later in this instruction description.

If the hold record to be freed is not last on the secondary chain, it is removed from the chain by moving its backward pointer to the backward pointer field of the next record. Then the secondary chain field is moved to the secondary chain field of the previous hold record and the cumulative hold fields are updated.

Freeing from the Primary Chain

If the freed hold record is the head of the secondary chain, the head of the secondary flag is reset. The second hold record on the secondary chain becomes the head of the seconday chain. The secondary chain index of the freed hold record is moved to the hold record chain index of the previous hold record on the primary chain, or into the appropriate hash hold table entry if the freed record was the first on the object chain. The new head of secondary chain hold record has its head-of-secondary chain flag set, unless it is the end of the secondary chain. The end-of-primary chain flag on the freed hold record is checked, and if it is set, the hold record that has just been marked the end-of-secondary chain is also marked the end-of-primary chain. If the EOC flag is reset, the hold record chain index of the freed hold record are moved to the new head of secondary chain.

If the freed hold record is not the head of secondary chain and the end-of-primary chain flag is set, the EOC flag is set in the preceding hold record in its chain. If it was first on the chain, the object chain's HHT (hold hash table) entry is set to zero. If there is no secondary chain and the freed hold record was not at the end of the primary chain, the object chain field of the record is moved into the object chain field of the preceding hold record on the object chain or into the appropriate hash table entry if the freed record was the first on the object chain.

Cumulative Hold Updates

The cumulative hold field is an OR (noninclusive) of the remaining holds on the secondary chain. When a hold record is removed from the secondary chain, all the preceding holds on that secondary chain may need their cumulative hold fields updated. A check is made to see if the removal of the hold affects the cumulative hold value. The hold of the freed hold record is used as an AND mask with its cumulative hold field. The result is then compared with the hold, and if they are equal, the chain's cumulative hold fields need not be updated.

If the cumulative hold field must be updated, the cumulative hold of the hold record to be freed is moved to the previous hold record. If the freed hold record was last on the secondary chain, this field is set to zero.

If the updated hold record is the head of the secondary chain, all the records on the secondary chain are up to date. If the updated hold record is not the head of the secondary chain, it may not be necessary to continue updating the cumulative hold fields. The hold fields of the currently updated hold record and the freed hold record are compared. If the result of ANDing these two values equals the hold of the hold record freed, the OR value does not change and there is no need to continue the updates. If they are not equal, this hold record's hold field and cumulative hold are ORed and placed in the cumulative hold field of the previous hold record and processing continues as stated at the beginning of this paragraph.

Programming Note: All activity on the secondary chain is currently implemented in VMC, but the operation is nullified. Control is passed by a second chain search exception when it is determined that a second chain must be searched. The chain is marked busy and parameters are passed through the checkpoint area in the TDE. Bytes hex C2 and hex C3 in the HMC checkpoint area hold the index to the hold record at the beginning of the secondary chain. The second and third operands are unchanged; however, the first operand may have been altered if the primary chain search was interrupted. A freed hold record is placed on the available hold record chain with the Return Available Hold Record instruction.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Requested free was allowed.
1	--
2	--
3	Continue searching hold chain.

Carry: Not applicable.

Boundary Requirements: The second-operand hold request block must be halfword aligned; otherwise, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Chain conflict
- Descriptor access: Busy
- Effective address overflow
- End of chain
- Second chain search
- Specification

FHRF Example

A hold of hex 02 is to be dropped for an object located at hex 8001 1803 0000. Three holds exist on the same primary chain: two from another TDE and one from the current TDE which is to be freed.

The address of the obejct to be freed has been loaded into register hex C and register hex D points to an area that contains the 6-byte object (usually an object but may be a group of bits or bytes) address and the 2-byte hold request block.

After executing the FHRF instruction, storage would look like:

Op	B_1	B_3	B_2	D_2
47	B	C	D	006

0 Bits 8 12 16 20 31

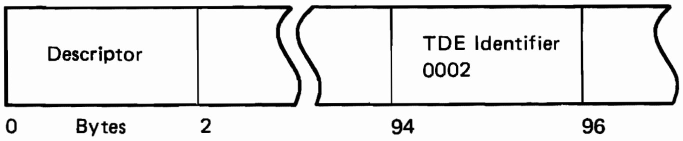
Assembler: FHRF B_1, B_3, $D_2(B_2)$

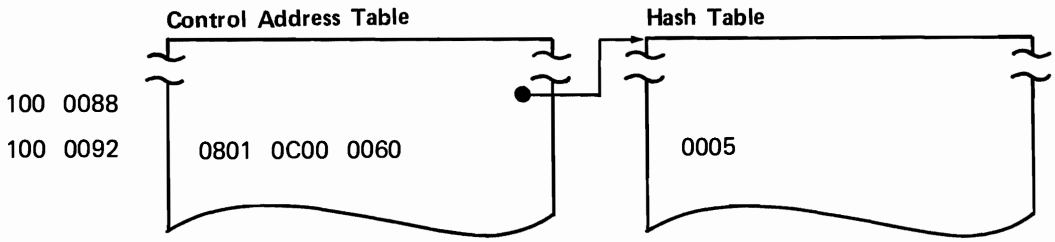
Machine: 47BC D006

	Before	After	
B_1(B):	xxxx xxxx xxxx	0801 0C00 0050	(TDE that requested hold)
B_2(D):	0801 D200 2200	0801 D200 2200	(pointer to hold request block)
B_3(C):	0801 1803 0000	0801 1803 0000	(object address)

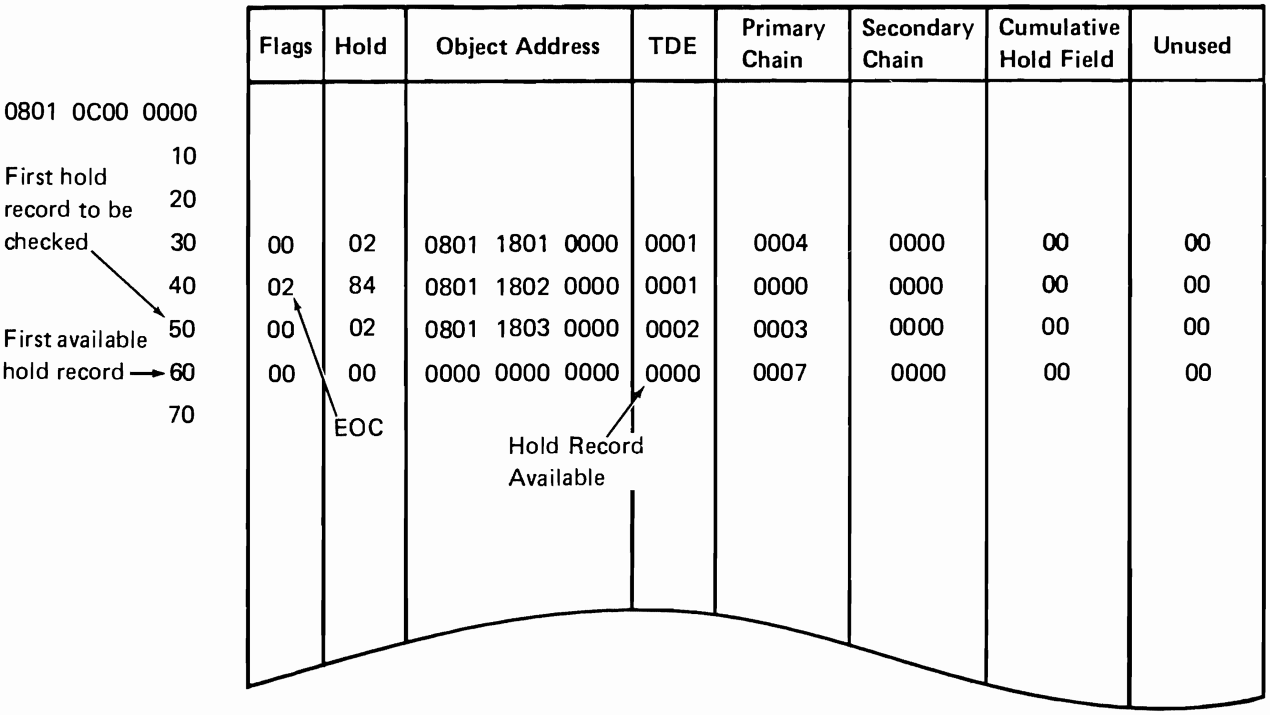
Storage — Before

TDE



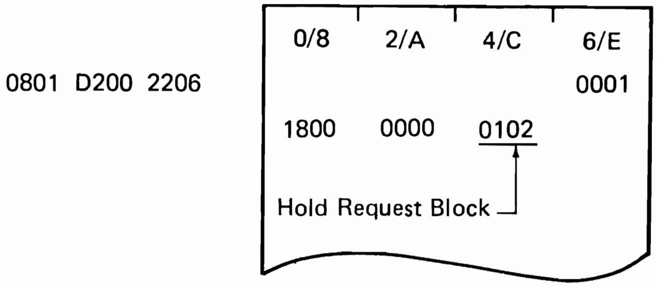


Hold Records—Before



	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	00	02	0801 1801 0000	0001	0004	0000	00	00
40	02	84	0801 1802 0000	0001	0000	0000	00	00
50	00	02	0801 1803 0000	0002	0003	0000	00	00
60	00	00	0000 0000 0000	0000	0007	0000	00	00
70								

Storage

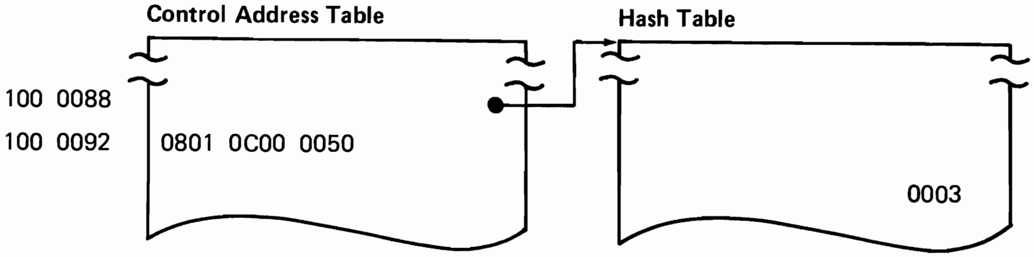


	0/8	2/A	4/C	6/E
0801 D200 2206				0001
	1800	0000	0102	

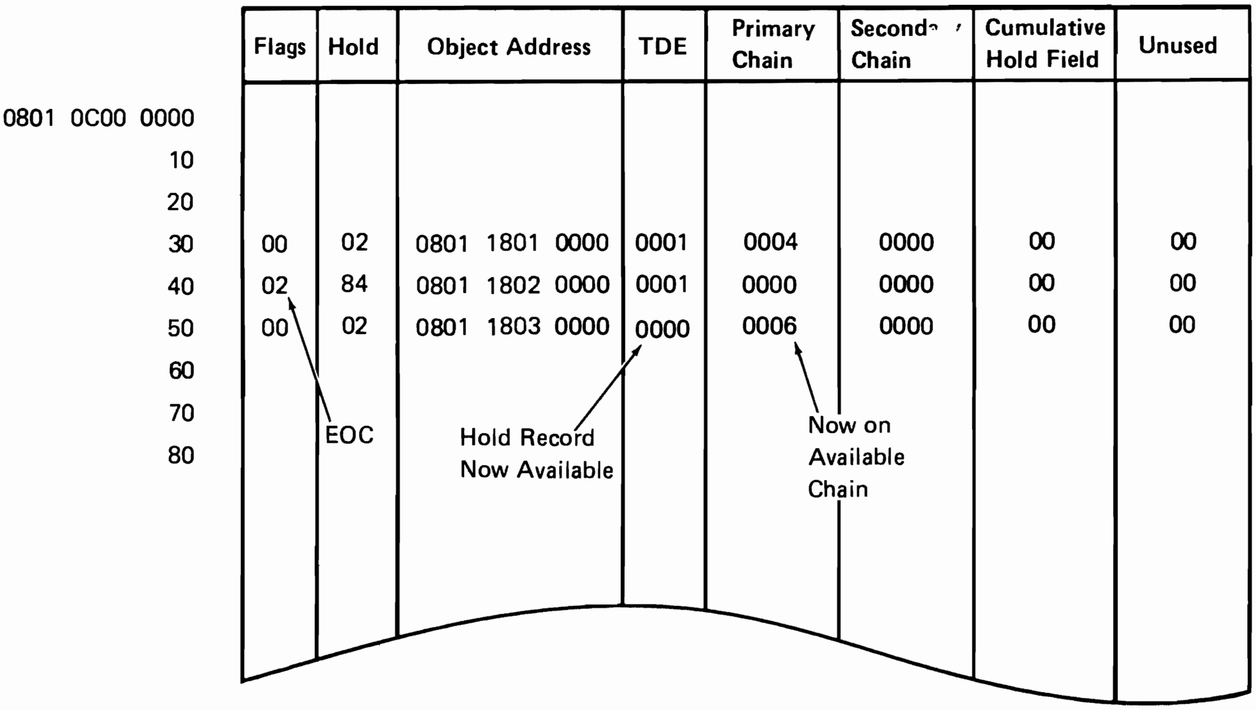
The hold record at address hex 8001 0C00 0050 was taken off the chain. This is indicated by the TDE requesting the hold now showing as not used (hex 000).

Note: Any of the hold records may be in any one of 2048 possible chains.

Storage — After



Hold Records — After



	Flags	Hold	Object Address	TDE	Primary Chain	Second⌐ ′ Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	00	02	0801 1801 0000	0001	0004	0000	00	00
40	02	84	0801 1802 0000	0001	0000	0000	00	00
50	00	02	0801 1803 0000	0000	0006	0000	00	00
60								
70								
80								

EOC

Hold Record Now Available

Now on Available Chain

FUNCTION CALL DOUBLE (FNC2)

Instruction Description

The function call mechanism is used to call the function routine selected by the index in the I-instruction field.

Format: SS

A8		I	B_1	D_1	B_2	D_1	
0 Bits	8	12	16	20	32	36	47

Operation: The index is used to access an entry in the FRAT (function routine address table) to determine where the function routine is located. Registers 0, 1, and 2 and the IAR are saved in the current stack entry prior to that routine.

Using the address found in base register 3, the current stack entry is accessed to determine if its storage capability will allow for the storage of up to 128 bytes. That capability is calculated by subtracting the offset of the current stack entry from the first halfword of the stack entry (that halfword pointing to the next stack entry).

The stack entry is used to save the following:

- Base register 0 in bytes 122 (hex 7A) through 127 (hex 7F)
- Base registers 1 through 2 in bytes 24 (hex 18) through 35 (hex 23)
- Updated IAR (instruction address register) in bytes 120 (hex 78) and 121 (hex 79)

After the registers are saved into the stack entry, the effective address of the first operand is loaded into base register 1, the effective address of·the second operand is loaded into base register 2, and the IAR and base register 0 are loaded with data from the function routine address table. The IAR is set to the 2-byte instruction address, and the base register 0 is set to the 6-byte function address, according to the following procedure: the I-field is multiplied by 10 (hex A); that product is added to the address of the FRAT, located at bytes 162-167 (hex A2-A7) in the CAT (control address table). The resultant address locates a 10-byte entry for a function routine. The first 2 bytes of the indexed FRAT entry are loaded into the IAR, and the next 6 bytes are loaded into base register 0. The last 2 bytes are ignored. A branch is then taken to the address that was formed by concatentating the upper 4 bytes of base register 0 and the 2 bytes of the IAR.

The effective addresses of the first and second operands are computed and checked for effective address overflow exceptions. No attempt is made to access the first and second operands, and they remain unchanged in storage.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: If the new values of the IAR and/or base register 0 are not halfword aligned, a specification exception is recognized and the instruction is suppressed. If the FRAT is not halfword aligned or the stack entry is not doubleword aligned, a specification exception is recognized and the operation is suppressed. If the FRAT crosses a segment boundary, an effective address overflow exception occurs and the instruciton is suppressed.

Program Exceptions:

- Addressing
- Address translation
- Effective address overflow
- Specification:

  FRAT not halfword aligned
  FRAT ENTRY IAR value not halfword aligned
  FRAT entry B0 value not halfword aligned
  Stack entry not doubleword aligned
  Stack entry less than 128 bytes long

FNC2 Example

Op	I_3	B_1	D_1	B_2	D_2
A8	00	7	456	4	A00

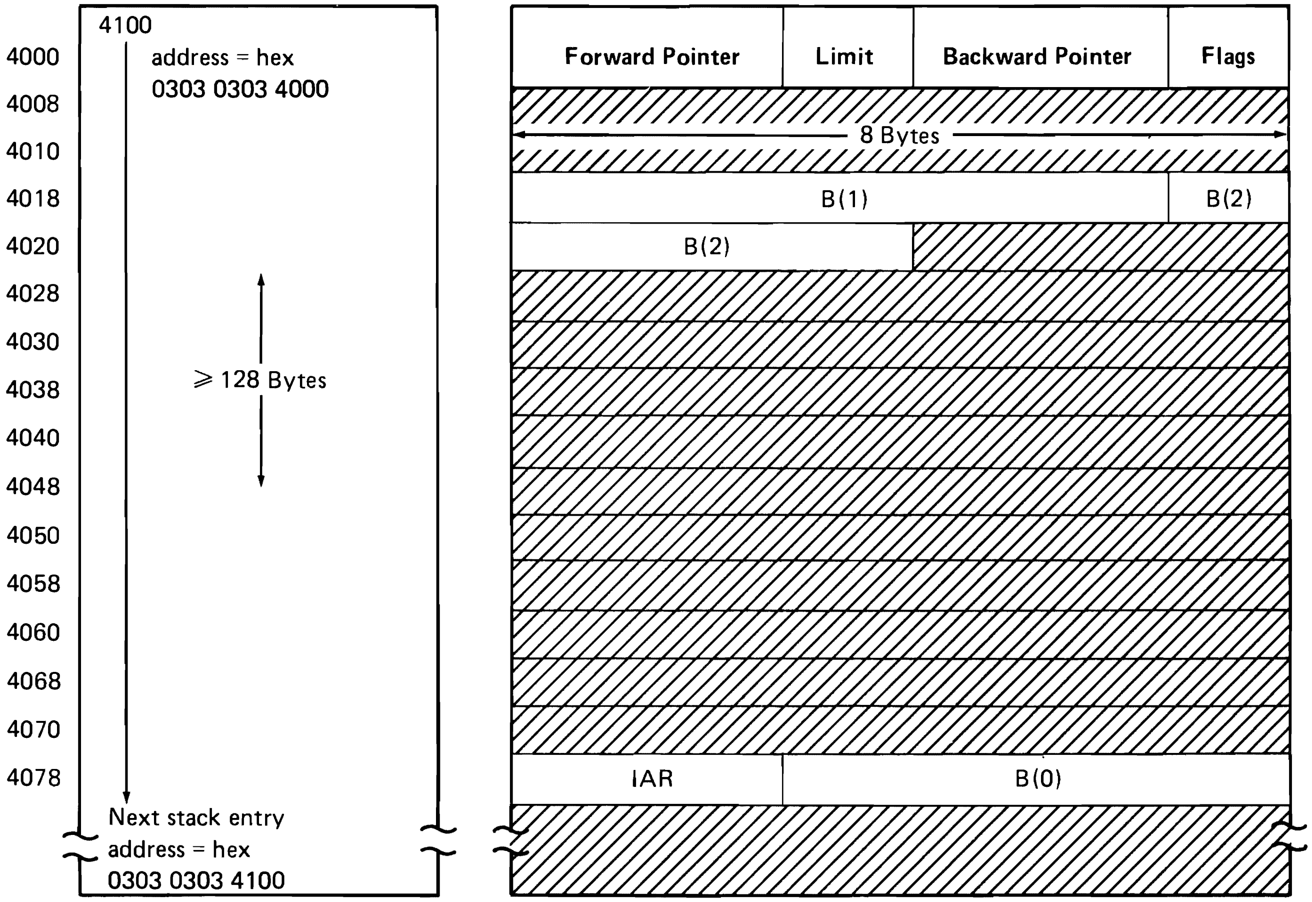
0 Bits 8 16 20 32 36 47

Assembler: FNC2 $D_1(B_1), D_2(B_2), I_3$

Machine: A800 7456 4A00

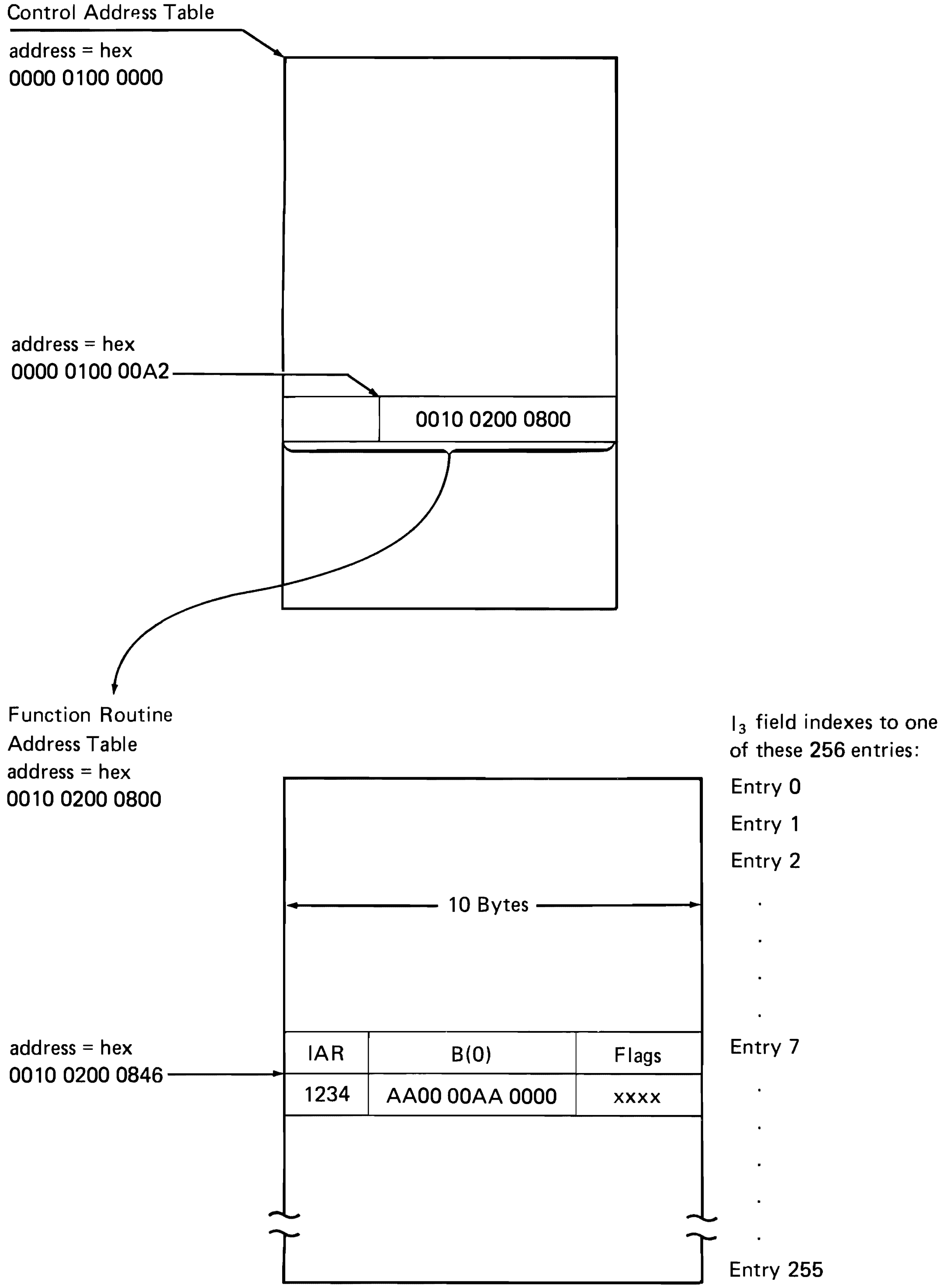
	Before			**After**		
B(0):	0000	1111	0000	AA00	00AA	0000
B(1):	0101	0101	1000	0707	0707	1567
B(2):	0202	0202	2000	0404	0404	4E44
B(4):	0404	0404	4444	0404	0404	4444
B(7):	0707	0707	1111	0707	0707	1111
IAR:	0800			1234		

Stack Entry Before Data **Doubleword Boundary**



Stack Entry After Data

4018	0101 0101 1000 0202
4020	0202 2000
4028	
4030	
4038	
4040	
4048	
4050	
4058	
4060	
4068	
4070	
4078	0806 0000 1111 0500



IAR	B(0)	Flags
1234	AA00 00AA 0000	xxxx

This page is intentionally left blank.

GRANT HOLD RECORD (GHR)

Instruction Description

The GHR instruction is designed to be a continuation of a Grant Hold Record First instruction that was interrupted. Therefore, the execution is almost identical except for some initial conditions. The GHR instruction assumes that the hold record chain has been marked *busy* by a previous Grant Hold Record First instruction and instead of using an object address and the hash table to locate the first hold record to check, the GHR instruction assumes the address of a hold record is loaded in a register.

The GHR instruction also checks for pending interrupts during execution. When the GHR instruction terminates, it is necessary to check the condition code to determine if execution completed or if the operation was interrupted and must be continued again by reexecuting the GHR instruction.

Format: RS

56	B_1	B_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The first-operand hold record and succeeding hold records are checked to see if the second operand hold request can be granted. The outcome of this check is: to grant the hold request, to signal that a conflict exists, or to indicate that further hold records on the chain of the third-operand object must be checked. In all three cases, the first-operand base register is updated to the address of the appropriate hold record.

A 4096-byte HHT (hold hash table) is accessed. Its address is in bytes hex 8A-8F of the control address table. This HHT address is set at IMPL (initial microprogram load) to point to the first byte in the page. The first AHR (available hold record) is also accessed using the AHR address in bytes hex 92-97 of the control address table.

The instruction functions the same as the Grant Hold Record First instruction except for the following differences. The address of the first hold record to be checked is found in the first-operand base register instead of by hashing the third-operand object address and going thrrough the hold hash table. This first hold record is not the first on the chain, so its busy flag is not checked; thus no descriptor access busy exception can occur in this instruction.

Since this instruction checks groups of hold records starting within the object hold chain, it does not set the chain busy flag in the first hold record of the chain when condition code 3 is set (the flag is assumed to have been set by the execution of a preceding Grant Hold Record First instruction). However, when condition code 0 exit is used, it goes back to this first hold record in the chain and resets the record's busy flag (bit 5 of the first byte).

Overflow and Sign Code: Not applicable.

Condition Code:

0 Requested hold was allowed.
1 --
2 --
3 Continue searching hold chain.

Carry: Not applicable.

Boundary Requirements: The second-operand hold request block must be halfword aligned; the first available hold record must be quadword aligned; and the first-operand hold record must be quadword aligned. If any of these requirements is violated, specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Chain conflict
- Effective address overflow
- End of chain
- Second chain search
- Specification

Programming Note: If a program exception occurs, the busy flag is not reset.

GHR Example

56	B	C	D	006
0 Bits	8	12	16	20 31

Assembler: GHR B_1,B_3,D_2(B_2)

Machine: 56BC D006

See the GHRF instruction example.

GRANT HOLD RECORD FIRST (GHRF)

Instruction Description

The GHRF instruction conditionally adds entries to a list of *hold records* that contain a 6-byte object address, a *hold* byte, the identification of the TDE (task dispatching element) that requested the hold on the object, and the chain address to the next hold record on the hold record chain. The addition of new entries is conditional because each grant request specifies a *test* byte as well as a *hold* byte. If an existing hold record has a hold on the specified object matching any bit in the test byte, and the existing hold was requested by a different TDE, then an exception occurs and the new request is not granted.

The starting point for the search of the hold record list is found by hashing the 6-byte object address specified in the third operand. An entry in a hold hash table points to the first hold record to be checked. Each hold record contains the index to the next hold record to be checked. As each hold is checked, its 6-byte address is placed in a register. The last entry on a chain is indicated by a flag bit in the hold record. The address of the hold hash table is located in the control address table. When a new hold record is to be added to a chain, the record is obtained from a chain of available hold records. The starting point for this chain is an address in the control address table, and the entries are chained the same way as hold records.

While this instruction is executing it periodically checks for pending interrupts. If an interrupt is pending, the condition code is set (the busy flag is set for this hold record chain) and execution of the GHRF instruction ends. Execution can then be continued by the Grant Hold Record instruction.

Format: RS

46	B_1	B_3	B_2	D_2	
0 Bits	8	12	16	20	31

Operation: The hold records in the hold record chain of the third-operand object are checked. The outcome of this check is: to grant the second-operand hold request, to signal that a conflict exists for that request (unless hold request block byte 1 = 0), or to indicate that further hold records on the chain must be checked. In all three cases the first-operand base register is loaded with the address of the appropriate hold record. If the hold request block byte 1 is zero, the first-operand base register is unchanged.

The address of the 4096-byte hold hash table is accessed in bytes hex 8A-8F of the control address table. This HHT (hold hash table) address was set by IMPL (initial microprogram load) to point to the first byte in the page. The first AHR (available hold record) is accessed using the AHR address given in bytes hex 92-97 of the control address table.

The third-operand register contains an object address. The 6-byte effective address is hashed to create a 2-byte index into the hold hash table. If the chain is empty (contains no hold records) the hold hash table entry is all zeros. In this case, the hold requested by the second operand can be granted. (See *Granting a Hold* for how a hold is granted).

If the hold hash table entry is not all zeros, the indexed hold record is accessed. The 2-byte hold hash table entry is multiplied by 16 and concatenated to the right of the high-order 28 bits of the AHR (available hold record) address, found in the control address table, to point to the first hold record on the chain. These bits of the AHR point to the start of the hold record area. The hold record's chain-busy flag (byte 0, bit 5) is checked. If the chain busy flag is set (indicating chain busy), the first operand register is unchanged, a descriptor-access busy occurs, and the operation is nullified.

If the chain busy flag is not set, hold records are checked for a hold conflict; the figure below indicates the number of records that can be checked. A conflict or end of chain in any hold record prohibits checking additional hold records. If the initial or previous groups of hold records has been checked without determining if the requested hold can be granted, a check is made for pending interrupts. If none are pending, additional groups of records are checked. If an interrupt is pending, it is handled the same as in a page fault. The instruction finishes as follows.

	Records Checked	
Models	Initial	Addition
3, 4, and 5	9	13
6, 7, and 8	14	20

The condition code is set to 3, the first-operand register is loaded with the address of the next hold record in the chain, the chain busy flag (bit 5 of the first byte) in the first hold record for this object chain is set, and the operation is completed.

The hold chains are set up in a dual chain structure. The primary chain consists of those hold records with hash synonyms. The secondary chain has all hold records with equal object addresses.

Primary Chain Search

Each hold record is checked as follows. The 6-byte object address field of the hold record (bytes 2-7) is compared with the third-operand register contents. If this field does not match (indicating the hold record is intended for a hold hash synonym of this object), the hold EOC flag (bit 6 of the first byte) is checked. If this EOC flag is set; the hold requested by byte 1 of the second-operand hold request block may be granted (see *Granting a Hold* in this instruction description). If the EOC flag is reset, a check is made to see if pending interrupts must be checked. If it is, pending interrupts are checked as described in the preceding paragraphs. If not, the next hold record in this chain (whose record index is found in bytes hex A and hex B of the current hold record) is accessed and checked as described earlier in this paragraph.

If the comparison of the object address fields of the preceding paragraph results in a match, the hold test field (byte 0 from the second-operand hold request block) is compared with the hold field (byte 1) of the current hold record. If any corresponding bits in these fields are both ones, a potential conflict exists and the hold field is compared to hex F8. If equal, the hold is an exclusive hold and no other holds can be placed on this object; a conflict has arisen (see *Conflicts* in this instruction description). If a potential conflict exists and the hold is not an exclusive hold, the TDE identifier field (byte 8-9) of the hold record is compared with the TDE ID field in bytes hex 92-97 of the current TDE. If these TDE IDs do not match, a conflict has arisen.

If no conflict was detected, the head of secondary chain bit is checked. If this hold record is not the head of secondary chain, the hold may be granted (see *Granting a Hold* in this instruction description). If there is a secondary chain, the cumulative hold field is checked with the hold test field. If any corresponding bits are set, a potential conflict exists on the secondary chain and the secondary chain must be searched. If there is no potential conflict on the cumulative hold field, the hold may be granted (see *Granting a Hold* in this instruction description).

Secondary Chain Search

The secondary chain is searched using the chain field (bytes hex C and hex D) of the hold record. Object addresses do not need to be checked because they are all equal. The hold test field (byte 0) of the second operand hold request block is compared with the hold field (byte 1) of the current hold record. If any corresponding bits in these fields are set, a potential conflict exists and the TDE identifier field is compared with the TDE identifier field of the current TDE. If the TDE identifiers do not match; a conflict has arisen. If the TDE identifiers match, it might not be necessary to continue searching the secondary chain. This hold record's cumulative hold field is checked. If there are no corresponding bits in these fields that are both set, the hold can be granted. If any corresponding bits in these fields are set, a potential conflict exists and the second chain search continues with the hold record pointed to by bytes hex C and hex D of the current hold record. If the end-of-secondary chain is reached (bytes hex C and hex D equal 0) before a conflict arises or a hold is granted, a specification exception is presented.

Granting a Hold

When it is determined that the requested hold may be granted, the HRB hold field (byte 1 of the hold request block) is checked. If it is all zeros, the new hold record is not created and chained, the TDE count field in bytes hex 96-97 of the TDE is not incremented by 1, and the first-operand register is not loaded. However, the condition code is set to zero and the operation is completed. If the HRB hold field is not all zeros, the requested hold is granted as follows. A hold record is obtained from the AHR (available hold record) chain using the AHR address contained in the CAT. If this is the last AHR (its EOC flag set), the first-operand register is unchanged, an end-of-chain program exception occurs, and the operation is nullified.

If this is not the last available chain record, the acquired hold record fields are filled in with pertinent data and the hold record is chained into the front of the object hold chain. Byte 0 (flags) of the acquired hold record is reset. Byte 1 (HRHOLD) is filled from HRBHOLD (byte 1 of the hold request block). Bytes 2-7 (HROBJ) are filled from the contents of the third-operand base register. Bytes 8-9 (HRTDE) are filled from the task identifier halfword field (bytes hex 94-95 from the current TDE). Bytes hex A-B (HRCHN) are filled from the hash table entry for this object chain. The HRCHN value from the acquired hold record is multiplied by 16 and the resultant 20 bit value is loaded into the low order 20 bits of the AHR (available hold record) address field in the control address table. The address of this acquired hold record is converted to a 2-byte record index and is moved to the hold hash table entry for this object chain.

If no other record exists, the new hold record is placed at the top of the chain by linking the rest of the chain to it. The appropriate hash table entry value is placed into this new record (bytes hex A and hex B). The hash table entry is then filled with the index pointing to this new hold record. If other hold records already exist for this object, the new hold record is marked head-of-secondary chain (byte 0, bit 0 is set). The secondary chain is linked to this hold record by placing the index (address) to the previous head-of-secondary chain into bytes hex C and hex D of the new hold record. The new hold record's index (address) is placed into bytes hex A and hex B of the hold record that was previously the head of the secondary chain. (This hold record's head-of-secondary chain bit is reset.) The cumulative hold field is generated, by ORing the hold and cumulative hold of the previous head of this secondary chain, and then placed into bytes hex E of the new hold record. This previous head-of-secondary chain is then removed from the primary chain by moving its chain pointer in bytes hex A and hex B and the EOC bit (byte 0, bit 6) to the same bytes in the previous hold record on the primary chain. However, if this previous head-of-secondary chain was the first hold record on the primary chain, then the value that was originally in bytes hex A and hex B is moved to bytes hex A and hex B of the acquired hold record, updating the primary chain pointer again. If this new hold record is the only hold record in its primary hash chain (hash table entry was zero), the EOC flag is set. The TDE hold count field in bytes hex 96 and hex 97 of the TDE is incremented by 1, the first operand base register is loaded with the address of the new hold record, and the condition code is set to 0.

Conflicts

When a conflict has arisen, the chain busy flag (byte 0, bit 5 in the first hold record in the chain) is set, the first operand register is loaded with the address of this conflicting hold record, a chain conflict exception is signaled, and the operation is nullified.

Programming Note: All activity on the secondary chain is currently implemented in VMC. The chain is marked busy and parameters are passed through the checkpoint area in the TDE. A second chain search exception is issued and the operation is nullified.

Checkpoint Area (Hex Bytes)	**Contents**
C2 and C3	Hash table entry offset.
C4 and C5	An index to the hold record just prior to the hold record at the beginning of the secondary chain. (A value of 0 indicates the head of secondary chain is also first on the primary chain.)
C6 and C7	An index pointing to an available hold record. (The hold record is removed from the available hold record list when the second chain search exception is present.)

Overflow and Sign Code: Not applicable.

Condition Code:

0	Requested hold was allowed.
1	--
2	--
3	Continue searching hold chain.

Carry: Not applicable.

Boundary Requirements: The second-operand hold request block must be halfword aligned, and the first available hold record must be quadword aligned. If one of these requirements is not met, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Chain conflict
- Descriptor access: Busy
- Effective address overflow
- End of chain
- Second chain search
- Specification

GHRF Example

A hold of hex 02 is being requested for hex 8001 1803 0000. The test byte specifies hex 01, so only a previous hold of hex 01 is considered a conflict. Two holds from a different TDE already exist on the same hold record primary chain.

The value for the hold has been loaded into register hex C. Register hex D points to the 6-byte value that precedes the hold request block.

Op 46	B_1 B	B_3 C	B_2 D	D_2 006
0 Bits	8	12	16	20 31

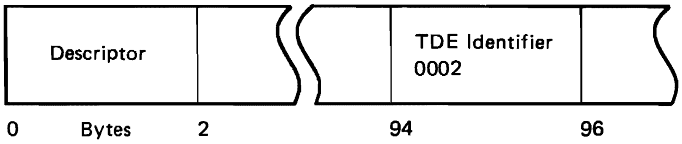
Assembler: GHRF B_1, B_3, D_2(B_2)

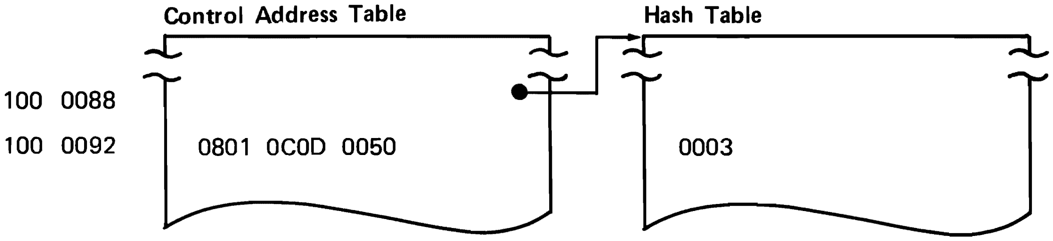
Machine: 46BC D006

	Before			After		
B_1(B):	xxxx	xxxx	xxxx	0801	0C00	0050
B_2(D):	0801	D200	2200	0801	D200	2200
B_3(C):	0801	1803	0000	0801	1803	0000

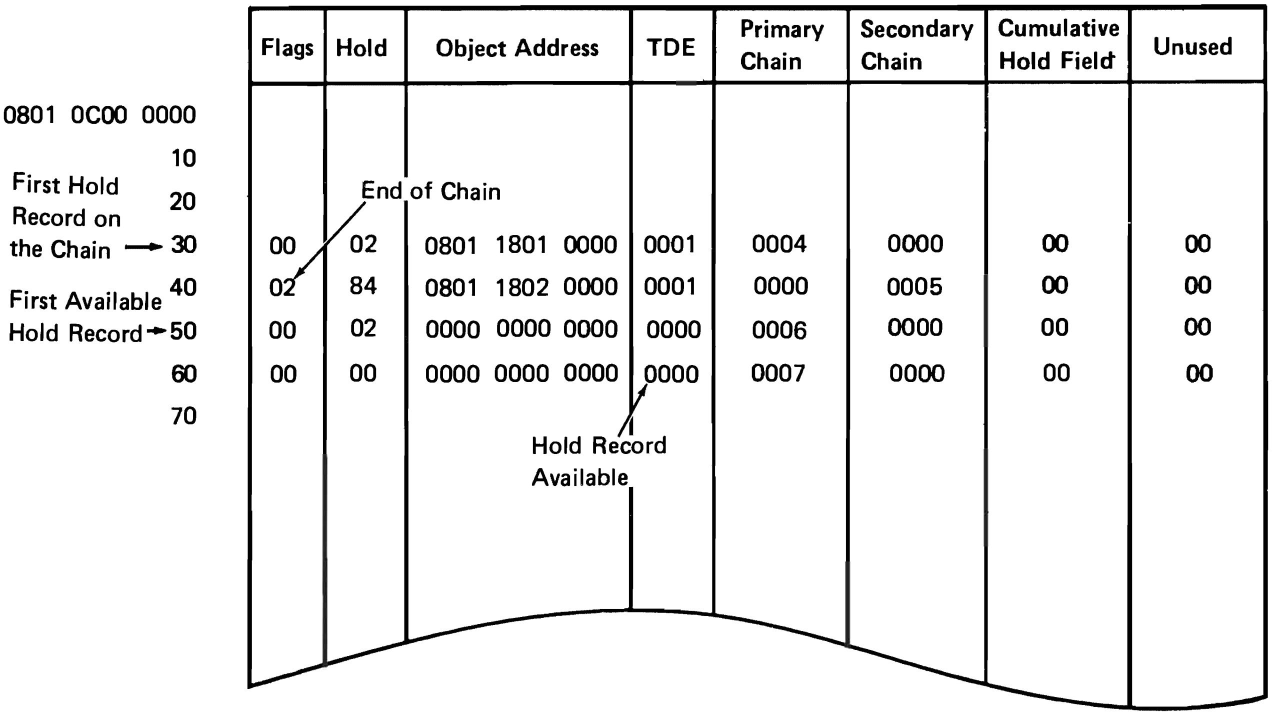
Storage — Before

TDE



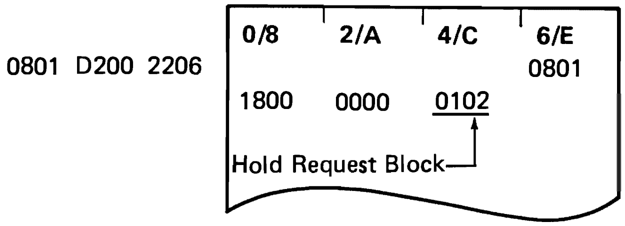


Hold Records — Before



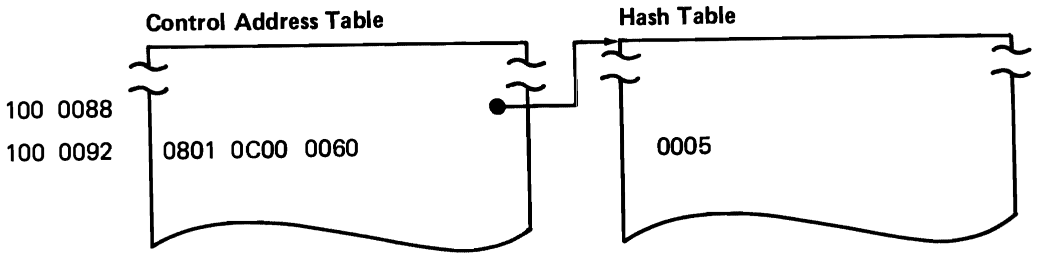
	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
First Hold Record on the Chain → 30	00	02	0801 1801 0000	0001	0004	0000	00	00
40	02	84	0801 1802 0000	0001	0000	0005	00	00
First Available Hold Record → 50	00	02	0000 0000 0000	0000	0006	0000	00	00
60	00	00	0000 0000 0000	0000	0007	0000	00	00
70								

Storage



	0/8	2/A	4/C	6/E
0801 D200 2206				0801
	1800	0000	0102	

Storage — After



Hold Records — After

	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	00	02	0801 1801 0000	0001	0004	0000	00	00
40	02	84	0801 1802 0000	0001	0000	0000	00	00
50	00	02	0801 1803 0000	0002	0003	0000	00	00
60	00	00	0000 0000 0000	0000	0007	0000	00	00
70								
80								

HASH AND VERIFY VIRTUAL ADDRESS (HVVA)

Instruction Description

The HVVA instruction provides support for linking virtual addresses (so the virtual address translator may resolve them) and for pinning and unpinning pages (to prevent storage management from stealing them).

Format: SS

D8		I_3	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation:

The instruction has varying results depending on the contents of the second operand, which is treated as a 6-byte address in storage.

- If the storage operand contains a V=R address, the condition code is set to 3 and the instruction ends. The immediate byte of the instruction is ignored.

- If the storage operand contains a virtual address, the address is hashed by the VAT (virtual address translator) microcode. The resulting 2-byte hash table class is placed in the left 2 bytes of the first operand.

- If the virtual address represented by the storage operand is not found by the VAT microcode, condition code 2 is set and the instruction ends. The immediate byte is ignored.

- If the virtual address represented by the storage operand is successfully translated, the PD (primary directory) entry valid bit is checked. If it is off, condition code 1 is set. The immediate byte is ignored. The PD entry identifier is formed and set in the right 2 bytes of the result.

- If the virtual address represented by the pointer is successfully translated and the PD entry valid bit is on, condition code zero is set. The immediate byte is interrogated. The PD entry identifier is formed and set in the right 2 bytes of the result. The action taken for the immediate byte is as follows:

I_3 Bits		
12 =	**13 =**	**Action**
0	0	Instruction ends; pin count in PD entry unchanged
1	0	Pin count in PD entry incremented
1	1	Pin count in PD entry decremented

Bits 14 and 15 are not used. The pin count is 1 byte, unsigned.

After execution of the HVVA instruction the bits of the first operand have the following meanings:

Bits	Contains
0-15	Hash table entry index value
16-31	Primary directory entry index

If the primary directory entry is not present, bits 16-31 are unchanged.

Overflow: A machine check occurs if the increment or decrement operation to the pin count causes an overflow or underflow respectively.

Sign Code: Not applicable.

Condition Code:

0 Primary directory entry exists and is valid
1 Primary directory entry exists and is invalid
2 Primary directory entry does not exist
3 The second operand is a V=R address

Carry: Not applicable.

Boundary Requirements: The first operand must begin on a word boundary and the second operand must begin on a halfword boundary. Otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: The PD entry may be found by multiplying the second halfword by 16, ignoring any bits carried out of the halfword, and adding the resulting halfword to the primary directory address in the control address table. The address of the hash table entry may be found by multiplying the first halfword by 2, ignoring any bit carried out of the halfword, and adding the result to the hash table address in the control address table. (In both cases the condition code must indicate that the halfword[s] are valid.)

The hash index 1 for address hex C001 CE00 0400 is formed. The result of hashing 2 is saved in the first 2 bytes of the result. Hash table index hex 0050 3 indicates the first PD entry currently containing a virtual address of that hash index.

PD entry index hex 00CE 4 does not contain the requested address but contains a similar one that hashes to the same hash index.

Since the next PD entry index 5 field is not hex 0000 (end-of-chain), at least one more member of the hash class is present. Index hex 00D1, turns out to be the address searched for. If index hex 00D1 had not been hex C001 CE00 0400 and had contained *end-of-chain* for its next PD entry index, the HVVA microcode would have (1) concluded that no more PD entries contained members of hash class hex 0050 and that hex C001 CE00 0400 was not resident at this time, and (2) set the condition code to 2 and ended the instruction.

The second 2 bytes of the result 6 area are set to the primary directory entry found.

Since the valid bit 7 (leftmost bit of the sixth byte of the code address) is on, the page is addressable. If it was off, the condition code would have been set to one and the instruction ended.

Since the immediate field says *increment pin count* and the page is resident and valid, the pin count is incremented from zero to 1. The condition code is set to zero. The instruction completes.

HVVA Example

Op		I_3	B_1	D_1	B_2	D_2
D8		8	4	000	5	000

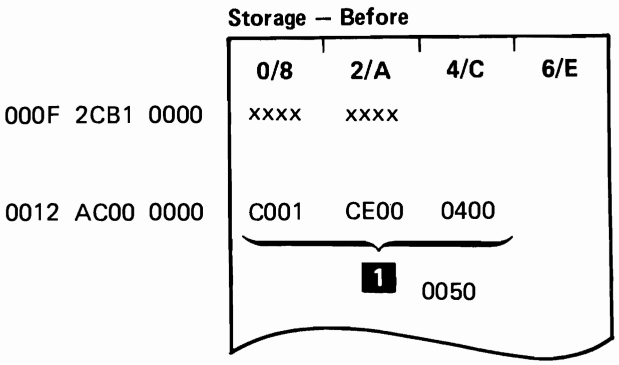
0 Bits 8 12 16 20 32 36 47

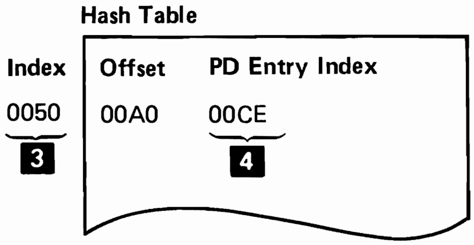
Assembler: HVVA $D_1(B_1), D_2(B_2), I_3$

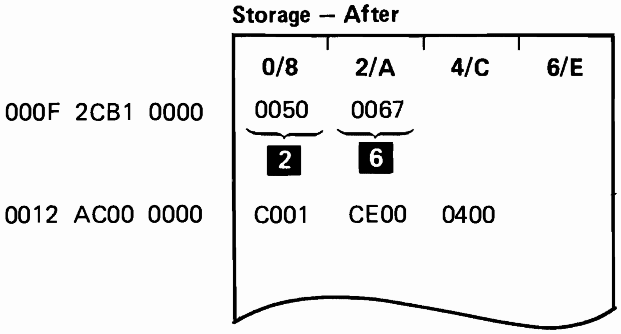
Machine: D808 4000 5000

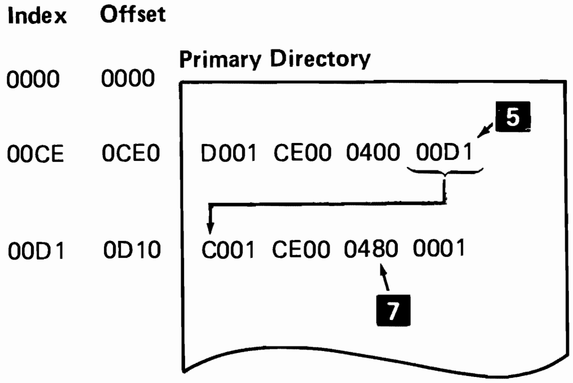
B_1(4): 000F 2CB1 0000

B_2(5): 0012 AC01 0000









	Before	After
Condition Code:	x	0

INSERT TAGS (INTAG)

Instruction Description

Each quadword of the first operand is tagged if the corresponding bit of the second operand is set.

Format: SS

A5	L_1	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Operation: L_1 applies only to the number of bytes of the first-operand minus 1.

The second operand is a halfword in storage with each bit (starting with the leftmost bit) containing the tagged indication for each of the quadwords of the first operand. When a zero bit is encountered, the corresponding quadword is untagged. When a 1 bit is encountered, the corresponding quadword is tagged. Any unused portion of the second operand is ignored. Any partial quadword at the end of the first operand is always untagged regardless of the tagged indication of the second operand.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must start on a halfword boundary and the first operand must start on a quadword boundary. Otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

INTAG Example

Op	L_1	B_1	D_1	B_2	D_2
A5	07	4	000	5	000
0 Bits 8		16 20		32 36	47

Assembler: INTAG $D_1(L_1,B_1),D_2(B_2)$

Machine: A507 4000 5000

B_1(4): 020A B76C 0000

B_2(5): 020A B62A 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
020A B62A 0000	AF9F			

As a result of this operation the words addressed by the first operand have their tag bits set as follows:

020A B76C 0000 04 08 0C	1
10 14 18 1C	0
20 24 28 2C	1
30 34 38 3C	0
40 44 48 4C	1
020A B76C 0050 54 58 5C	1
60 64 68 6C	1
70 74 78 7C	1
80 84 88 8C	1
90 94 98 9C	0
020A B76C 00A0 A4 A8 AC	0
B0 B4 B8 BC	1
C0 C4 C8 CC	1
D0 D4 D8 DC	1
E0 E4 E8 EC	1
020A B76C 00F0 F4 F8 FC	1

INVALIDATE PRIMARY DIRECTORY ENTRY (IPDE)

Instruction Description

The IPDE instruction provides support for main storage management to steal page frames. Conceptually, pages are either resident (in main storage) or nonresident (in auxiliary storage only). However, while a page is in the process of being paged in from secondary storage or being eliminated from main storage, the page must be marked in some manner to account for such things as multiple address translation exceptions to the same page by concurrent processes or for attempts by processes to reference a page being stolen by storage management.

Support for these conditions is provided by the notion of a valid PD (primary directory) entry. When a virtual address is inserted in the PD entry, the valid bit within that PD entry is left off until the page is read from auxiliary storage (or otherwise set to the correct contents). Any attempts to reference the address in this period generates address translation exceptions (page faults) just as if no PD entry contained the address.

The address translation exception handler, however, by executing the HVVA instruction, distinguishes no PD entry from a PD entry with the valid bit off. This allows special case handling for multiple processes with concurrent address translation exceptions (since only one may actually insert addressability in the PD entry and perform the I/O).

The valid bit may be turned on with any computational instruction, such as the OR Byte Immediate instruction. Turning it off, however, requires this instruction for three reasons: the use by I/O registers, pinning, and the LB (lookaside buffer). An I/O register is internally implemented such that an internal 4-byte rather than a 6-byte form of the address is used. First, the 4-byte form addresses storage directly without going through virtual address translation when references stay within the current page. That is, the I/O register pretranslates the address. Thus, invalidating a page requires destroying addressability of any I/O registers addressing the subject page. Second, a page that is pinned by another user must always remain addressable and be addressed in the same page frame. Thus, if this page is pinned, this instruction cannot be allowed to complete successfully. Finally, the hardware LB must be purged of the entry for this PD entry, if present. This internal buffer is not addressable directly by any IMP instructions.

The operand for this instruction is a PD entry identifier, which, when multiplied by 16 (ignoring any bits carried out of the halfword) and added to the base address of the PD, addresses the PD entry to be invalidated.

Format: SI

83		5	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The pin count, bits 64-71 in the PD entry identified by the operand, is compared to zero. If it equals zero, the valid bit is then reset. In addition, if a copy of this directory entry is in the LB, the change bit in the LB entry is ORed into bit 42 of the PD entry. The entry is then removed from the LB. In addition, one of the PD I/O use bits is checked. If the page was being used by I/O (bit 44=1), the page is removed from the I/O resolved address register. If the pin count does not equal zero, the PD is not changed and the LB is not checked.

Bits 0-15 of the operand are used as the PD index value. If the index value specifies a directory entry containing a V=R address, a specification exception occurs and the operation is suppressed. A specification exception will not be presented if the index value specifies a directory entry beyond the range of directory entries.

The high-order 4 bits (bits 0-3) of the PD index identified by the first operand are not used. The high-order bit (bit 0) of the hash table entry index is not used to index the hash table.

Overflow and Sign Code: Not applicable.

Condition Code:

0	PD entry invalidated, pin count = 0
1	PD entry not invalidated, pin count ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements: The operand occupies 2 bytes in storage and must begin on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: The PD entry identifier is returned in bits 16-31 of the result from the Hash and Verify Virtual Address instruction.

IPDE Example

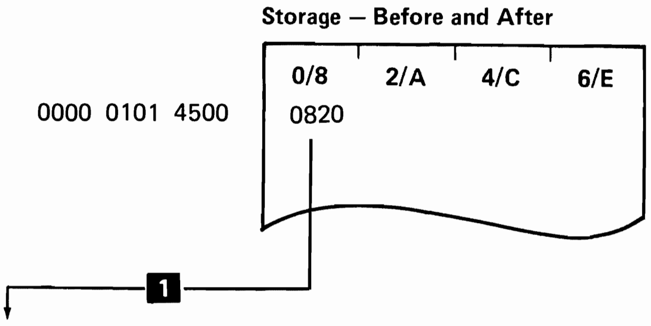
Op		E	B$_1$	D$_1$
83		5	4	320

0 Bits 8 12 16 20 31

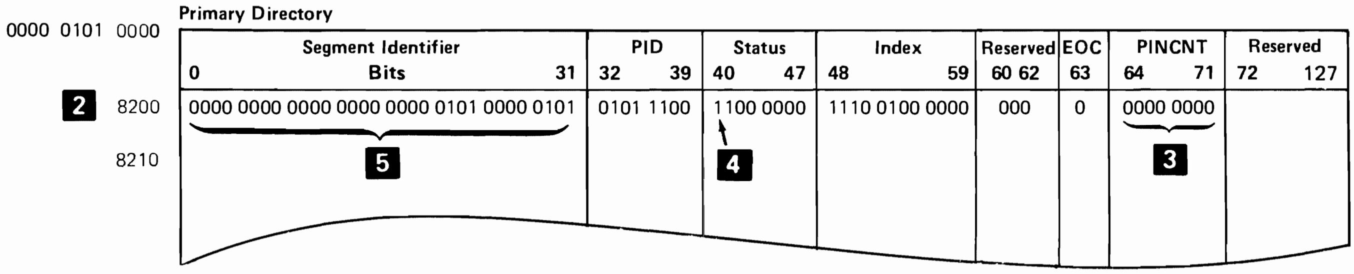
Assembler: IPDE D$_1$(B$_1$)

Machine: 8305 4320

B$_1$(4): 0000 0101 41E0



PD Base Address: 0000 0101 0000



Segment Identifier 0 Bits 31	PID 32 39	Status 40 47	Index 48 59	Reserved 60 62	EOC 63	PINCNT 64 71	Reserved 72 127
0000 0000 0000 0000 0000 0101 0000 0101	0101 1100	1100 0000	1110 0100 0000	000	0	0000 0000	

The first operand addresses the PD entry index value. The processor multiplies the PD entry index value by 16, ignoring any bits carried out by the halfword. This halfword offset is then added 1 to the PD base address found in the control address table in order to address the PD entry 2.

The pin count 3 is interrogated. If it had been nonzero, condition code one would have been set and the instruction ended. Since it is zero, the valid bit 4 in the PD entry status field is set to zero and the virtual address 5 is used to interrogate the lookaside buffer and the I/O registers. If the lookaside buffer entry is present (in the processor), the change bit is ORed into the PD entry status field and the lookaside buffer entry is removed. All I/O registers pointing to the subject page are invalidated.

	Before	After
Condition Code:	1	0

JUMP ON BITS OFF (JBF)

Instruction Description

The state of the first-operand bits selected by a mask is used to determine whether the jump is taken. A mask of zero results in no jump.

Format: RI

5F	r_1	0	I_2	J_3

0 Bits 8 12 16 24 31

Operation: The I_2 byte (immediate data) is used as an 8-bit mask. The bits of the mask are made to correspond one for one with the bits of the byte register designated by r_1.

A set mask bit indicates that the register bit is to be tested. When the mask bit is reset, the register bit is ignored. When the selected register bits are all reset, the updated instruction address is incremented by the 8-bit jump displacement, J_3. The 8-bit jump displacement is added to bits 8-15 of the updated instruction address with both operands treated as unsigned binary quantities. Otherwise instruction sequencing proceeds with the updated instruction address.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: If the sum of the updated instruction address and the jump displacement cause a carry, an effective address overflow exception occurs and the operation is suppressed.

Boundary Requirements: The final instruction address must begin on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

JBF Example

Op	r_1	E	I_2	J_3
5F	5	0	AF	2A

0 Bits 8 12 16 24 31

Assembler: JBF r_1, J_3, I_2

Machine: 5F50 AF2A

	Before	Updated	After
r_1(5):	50	–	50
IAR:	3500	3504	352E

JUMP ON BITS ON (JBN)

Instruction Description

The state of the first-operand bits selected by a mask is used to determine whether the jump is taken.

Format: RI

5E	r_1	0	I_2	J_3
0 Bits	8	12	16	24 31

Operation: A mask of zero results in no jump.

The I_2 byte (immediate data) is used as an 8-bit mask. The bits of the mask are made to correspond one for one with the bits of the byte register designated by r_1.

A set mask bit indicates that the register bit is to be tested. When the mask bit is reset, the register bit is ignored. When all of the selected register bits are all set, the updated instruction address is incremented by the 8-bit jump displacement, J_3. The 8-bit jump displacement is added to bits 8-15 of the updated instruction address with both operands treated as unsigned binary quantities. Otherwise, instruction sequencing proceeds with the updated instruction address.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: If the sum of the updated instruction address and the jump displacement cause a carry, an effective address overflow exception occurs and the operation is suppressed.

Boundary Requirements: The final instruction address must begin on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

JBN Example

Op	r_1	E	I_2	J_3
5E	5	0	AF	2A
0 Bits	8	12	16	24 31

Assembler: JBN r_1, J_3, I_2

Machine: 5E50 AF2A

	Before	Updated	After
r_1(5):	AF	–	AF
IAR:	1500	1504	152E

This page is intentionally left blank.

JUMP ON CONDITION (JC)

Instruction Description

The jump displacement in J_2 is added to the updated instruction address if the condition code is as specified by M_1; otherwise, normal instruction sequencing proceeds with the updated instruction address.

Format: RI

4E		M_1		J_2
0 Bits	8	12	16	24 31

Operation: M_1 is used as a 4-bit mask. The 4 bits of the mask correspond, left to right, with the four condition codes (0, 1, 2, and 3). The jump is taken whenever the condition code has a corresponding set mask bit.

The jump address is formed by adding the 8-bit jump displacement, J_2, to bits 8-15 of the updated instruction address with both operands considered as binary unsigned quantities.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The updated instruction address must begin on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed. If the sum of the jump offset and the updated instruction offset crosses a segment boundary, an effective address overflow exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: The IMP instruction assembler uses the following extended mnemonics:

Extended Mnemonic	Meaning	Standard Mnemonic	Mask Code (Hexadecimal)
JH	Jump High	JC	2
JL	Jump Low	JC	4
JE	Jump Equal	JC	8
JNH	Jump Not High	JC	D
JNL	Jump Not Low	JC	B
JNE	Jump Not Equal	JC	7
JP	Jump Plus	JC	2
JM	Jump Minus	JC	4
JZ	Jump Zero	JC	8
JNP	Jump Not Plus	JC	D
JNM	Jump Not Minus	JC	B
JNZ	Jump Not Zero	JC	7
JO	Jump Ones	JC	1
JM	Jump If Mixed	JC	4
JZ	Jump If Zeros	JC	8
JNO	Jump If Not Ones	JC	E

JC Example

Op 4E		M 1		J_2 C0

0 Bits 8 12 16 24 31

Assembler: JC M_1, J_2

Machine: 4E01 00C0

	Before	Updated	After
IAR:	A200	A204	A2C4
Condition Code:	3	–	3

LOAD (L)

Instruction Description

The second operand is placed unchanged into the register designated by the first operand.

Format: RS

94	B_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

L Example

Op	B_1	E	B_2	D_2
94	4	0	3	250

0 Bits 8 12 16 20 31

Assembler: L B_1, D_2 (B_2)

Machine: 9440 3250

	Before			After		
B_1 (4):	xxxx	xxxx	xxxx	0000	ABCD	EF00
B_2 (3):	0020	2A00	0000	0020	2A00	0000

Storage — Before and After

	0/8	2/A	4/C	6/E
0020 2A00 0250	0000	ABCD	EF00	

LOAD ADDRESS (LA)

Instruction Description

The address specified by B_2 and D_2 is loaded into the base register specified by B_1.

Format: RS

53	B_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The address computation follows the rules for address arithmetic.

No storage references for operands take place, and the address is not inspected for an addressing exception.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: If the sum of the offset portion of B_2 and the displacement results in a carry, an effective address overflow exception occurs and the operation is suppressed.

Boundary Requirements: None.

Program Exception: Effective address overflow

LA Example

Op	B_1	E	B_2	D_2
53	3	0	4	B00

0 Bits 8 12 16 20 31

Assembler: LA B_1, D_2(B_2)

Machine: 5330 4B00

	Before			After		
B_1(3):	xxxx	xxxx	xxxx	0000	EB00	0B00
B_2(4):	0000	EB00	0000	0000	EB00	0000

LOAD AND VERIFY TAGS (LVT)

Instruction Description

The LVT instruction provides support for loading the address value of a pointer into a base register. Verifications are performed on the specified storage to ensure that it does contain a pointer. Checks are performed to ensure that:

- Boundary alignment is a 16-byte (quadword) multiple.

- The hardware tags are on.

- The pointer type is one of those specified as allowable on the instruction.

- The pointer is resolved to an object. A pointer may contain an initial value, requiring it to be resolved.

- The object addressed by the pointer has not been destroyed.

Upon passing all of these verifications, the address value within the pointer is loaded into the base register providing addressability to the object for VMC instructions. The condition code is set as a result of the instruction to indicate the type of pointer that was accessed.

Format: RS

64	B_1	I_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The address located in bytes hex A-F of the second operand is placed in the first-operand location.

The following validity checks are made on the second operand:

- The second operand must be tagged.

- Bit 2 of the second operand must be zero.

- Bytes 8 and 9 of the second operand must match the halfword in storage located at the address determined by taking bytes hex A, B, and C of the second operand and concatenating hex 00 0004 on the right.

The type of pointer (specified by bits 0-1 of the second operand) is verified to match with that allowed by I_3.

Bits 0-1 specify the pointer type as follows:

Second Operand

Bits 0-1	**Decode**	**Description**
00	1000	System pointer
01	0100	Instruction pointer
10	0010	Space pointer
11	0001	Data pointer

I_3	**Description**
0---	System pointer *not* allowed
1---	System pointer allowed
-0--	Instruction pointer *not* allowed
-1--	Instruction pointer allowed
--0-	Space pointer *not* allowed
--1-	Space pointer allowed
---0	Data pointer *not* allowed
---1	Data pointer allowed

To verify the pointer match, the AND of the I_3 field with the decode of bits 0-1 must be nonzero.

If any of the above checks fails, a verify exception occurs and the operation is suppressed.

Overflow and Sign Code: Not applicable.

Condition Code:

0	System pointer
1	Instruction pointer
2	Space pointer
3	Data pointer

Carry: Not applicable.

Boundary Requirements: The second operand is a quadword and must start on a quadword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification
- Verify

LVT Example

Op	B_1	I_3	B_2	D_2
64	5	3	2	0C0

0 Bits 8 12 16 20 31

Assembler: LVT B_1, D_2(B_2), I_3

Machine: 6453 20C0

	Before			After		
B_1(5):	xxxx	xxxx	xxxx	0001	2C00	5148
B_2(2):	0001	5005	0000	0001	5005	0000
Condition Code:		x			2	

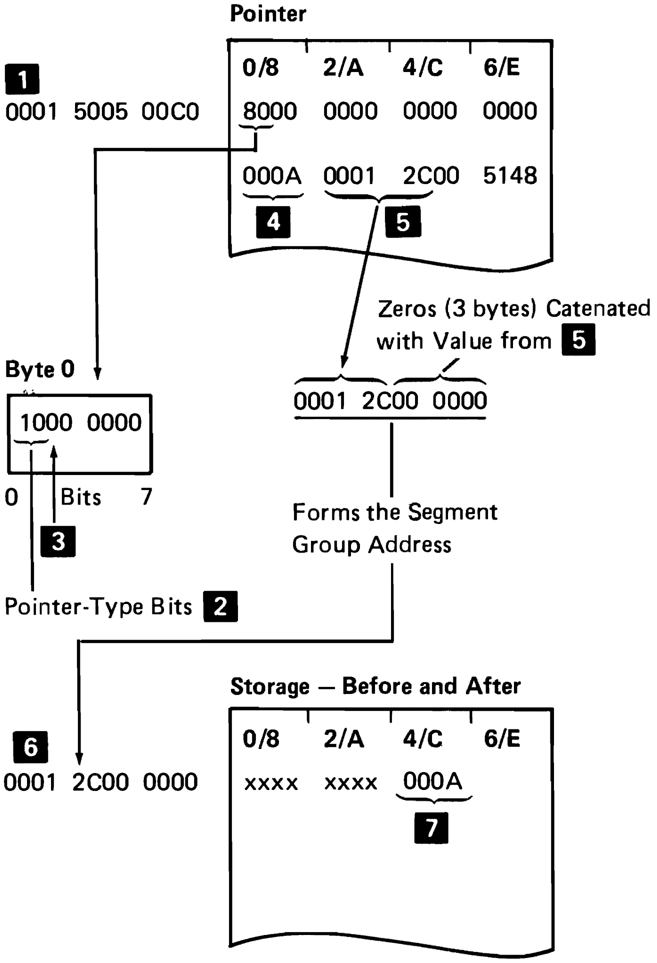
The pointer is accessed through the base plus the displacement 1 (the second operand) specified in the instruction. The type of pointer 2 is verified to match with that allowed by the I_3 field of the instruction. In the example I_3 = 3 (space pointer allowed and data pointer allowed). If the type is not allowed, a verify exception occurs.

The initial value indicator, bit 2, of the pointer 3 is checked for a value of zero. If not zero, a verify exception occurs. A value of 1 can only occur for system and data pointers and indicates that they contain a pointer to a name and must be resolved to the actual object. This resolution is done by a default exception handler for the verify exception.

The segment group extender value 4 in the pointer is checked against the extender value 7 in the header of the segment group 6 identified by bytes A-C 5 of the pointer. If not equal, a verify exception occurs. (Equal means that the segment group is still being used to contain the same object that it contained when the pointer was built.)

The first oprand is updated with the address value from bytes A-F of the pointer.

The condition code is set to indicate the type of pointer accessed. In this example, it is set to 2 to indicate a space pointer.



LOAD BYTE (LB)

Instruction Description

The second operand is placed unchanged into the register designated by the first-operand register.

Format: RS

74	r_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

LB Example

Op	r_1	E	B_2	D_2
74	5	0	3	210

0 Bits 8 12 16 20 31

Assembler: LB r_1, D_2 (B_2)

Machine: 7450 3210

B_2(3): 001B CDE0 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
001B CDE0 0210	A5			

	Before	After
r_1(5):	xx	A5

LOAD BYTE REGISTER (LBR)

Instruction Description

The second operand is placed in the first-operand register.

Format: RR

14	r_1	r_2

0 Bits 8 12 15

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

LBR Example

Op	r_1	r_2
14	3	4

0 Bits 8 12 15

Assembler: LBR r_1, r_2

Machine: 1434

	Before	After
r_1(3):	xx	FF
r_2(4):	FF	FF

LOAD BYTE REGISTER IMMEDIATE (LBRI)

Instruction Description

The second operand is placed into the register designated by the first operand.

Format: RI

44	r_1	0	I_2	

0 Bits 8 12 16 24 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

LBRI Example

Op	r_1	E	I_2	
44	5	0	A9	

0 Bits 8 12 16 24 31

Assembler: LBRI r_1, I_2

Machine: 4450 A900

	Before	After
r_1 (5):	xx	A9

LOAD HALFWORD (LH)

Instruction Description

The second operand is placed into the register designated by the first operand.

Format: RS

84	R_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

LH Example

Op	**R_1**	**E**	**B_2**	**D_2**
84	5	0	3	C00
0 Bits	8	12	16	20 31

Assembler: LH R_1, D_2 (B_2)

Machine: 8450 3C00

B_2(3): 00AF 0210 0000

Storage – Before and After

	0/8	**2/A**	**4/C**	**6/E**
00AF 0210 0C00	ABCD			

	Before	**After**
R_1(5):	xxxx	ABCD

LOAD HALFWORD REGISTER (LHR)

Instruction Description

The second operand is placed in the first-operand register.

Format: RR

24	R_1	R_2

0 Bits 8 12 15

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

LHR Example

Op	R_1	R_2
24	3	4

0 Bits 8 12 15

Assembler: LHR R_1, R_2

Machine: 2434

	Before	After
R_1(3):	xxxx	ABCD
R_2(4):	ABCD	ABCD

LOAD HALFWORD REGISTER IMMEDIATE (LHRI)

Instruction Description

The second operand is placed into the register designated by the first operand.

Format: RI

54	R_1	0	I_2

0 Bits 8 12 16 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

LHRI Example

Op	R_1	E	I_2
54	3	0	ABCD

0 Bits 8 12 16 31

Assembler: LHRI R_1, I_2

Machine: 5430 ABCD

	Before	After
R_1(3):	xxxx	ABCD

LOAD HASH TABLE ENTRY ADDRESS (LHTEA)

Instruction Description

The address of the hash table entry indexed by the second operand is loaded into the base register specified by the first operand.

Format: RS

83	B_1	2	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The second operand is used as the 2-byte hash table index value. The address of the hash table entry indexed by the second operand is formed by shifting the index 1 bit to the left, converting it from an index to an offset (bit 15 becomes zero). That offset is added to the original address of the hash table and loaded into the base register specified by the first operand. No storage reference is made for the hash table entry, and the high-order bit (bit 0) of the hash table entry index identified by the second operand is not used.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand occupies 2 bytes in storage and must begin on a halfword boundary; otherwise, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

LHTEA Example

Op	B_1	E	B_2	D_2
83	8	2	4	074

0 Bits 8 12 16 20 31

Assembler: LHTEA B_1, D_2(B_2)

Machine: 8382 4074

	Before	After
B_1(8):	xxxx xxxx xxxx	0000 0102 0C42
B_2(4):	0000 0100 2480	0000 0100 2480

Hash Table Address: 0000 0102 0000

Storage – Before and After

	0/8	2/A	4/C	6/E
0000 0102 24F4			0621	

LOAD MULTIPLE (LM)

Instruction Description

A set of registers is loaded from the locations designated by the second-operand address.

Format: RS

95	B_1	I_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The first-operand field identifies the first register to be loaded, and I_3 specifies the number of additional registers to be loaded.

The storage area from which the contents of the registers are obtained starts at the location specified by the second-operand address and continues through as many locations as needed.

The registers are loaded in the ascending order of their addresses, starting with the register specified by the first operand. The register addresses wraparound from hex F to 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operands must each start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address
- Specification

LM Example

Op 95	B_1 D	I_3 4	B_2 4	D_2 200
0 Bits	8	12	16	20 31

Assembler: LM B_1, I_3, D_2(B_2)

Machine: 95D4 4200

B_2(4): 0000 2A00 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
0000 2A00 0200	0000	1234	5678	0000
	2100	2A30	0000	0A30
	BC30	3A20	FFF3	21A0
	14BC	DEF0	33A0	

Base Register	Before			After		
D	xxxx	xxxx	xxxx	0000	1234	5678
E	xxxx	xxxx	xxxx	0000	2100	2A30
F	xxxx	xxxx	xxxx	0000	0A30	BC30
0	xxxx	xxxx	xxxx	3A20	FFF3	21A0
1	xxxx	xxxx	xxxx	14BC	DEF0	33A0

LOAD MULTIPLE BYTE (LMB)

Instruction Description

A set of registers is loaded from the locations designated by the second-operand address.

Format: RS

75	r_1	I_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The first-operand field identifies the first register to be loaded, and I_3 specifies the number of additional registers to be loaded.

The storage area from which the contents of the registers are obtained starts at the location specified by the second-operand address and continues through as many locations as needed.

The registers are loaded in the ascending order of their addresses, starting with the register specified by the first operand. The register addresses wraparound from hex F to 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

LMB Example

Op 75	r_1 5	I_3 2	B_2 4	D_2 000

0 Bits 8 12 16 20 31

Assembler: LMB r_1, I_3, D_2(B_2)

Machine: 7552 4000

B_2(4): 2A3C F1FA 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
2A3C F1FA 0000	1234	56		

Byte Registers	Before	After
5	xx	12
6	xx	34
7	xx	56

LOAD MULTIPLE HALFWORD (LMH)

Instruction Description

A set of registers is loaded from the locations designated by the second-operand address.

Format: RS

85	R_1	I_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The first-operand field identifies the first register to be loaded, and I_3 specifies the number of additional registers to be loaded.

The storage area from which the contents of the registers are obtained starts at the location specified by the second-operand address and continues through as many locations as needed.

The registers are loaded in the ascending order of their addresses, starting with the register specified by the first operand. The register addresses wraparound from hex F to 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

LMH Example

Op	R_1	I_3	B_2	D_2
85	0	4	0	130

0 Bits 8 12 16 20 31

Assembler: LMH R_1, I_3, D_2(B_2)

Machine: 8504 0130

B_2(0): 0135 A210 0000

Storage – Before and After

	0/8	2/A	4/C	6/E
0135 A210 0130	0102	A1A2	B1B2	C1C2
	D1D2			

Halfword Registers	Before	After
0	0000	0102
1	xxxx	A1A2
2	xxxx	B1B2
3	xxxx	C1C2
4	xxxx	D1D2

LOAD PRIMARY DIRECTORY ENTRY ADDRESS (LPDEA)

Instruction Description

The address of the primary directory entry indexed by the second operand is loaded into the base register specified by the first operand.

Format: RS

83	B_1	1	B_2	D_2

0 Bits 8 12 16 20 31

Operation:

The second operand is used as the primary directory index value. The address of the primary directory entry indexed by the second operand is formed by the shifting index to the left 4 bits, converting it from an index to an offset (bits 12-15 become zeros). This offset is added to the original address of the primary directory and loaded into the base register specified by the first operand. The high-order 4 bits (bits 0-3) of the primary directory index identified by the second operand are ignored and treated as zeros.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand occupies 2 bytes in storage and must begin on a halfword boundary; otherwise, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

LPDEA Example

Op	B_1	E	B_2	D_2
83	6	1	4	076

0 Bits 8 12 16 20 31

Assembler: LPDEA B_1, D_2(B_2)

Machine: 8361 4076

	Before	After
B_1(6):	xxxx xxxx xxxx	0000 0101 0820
B_2(4):	0000 0100 2480	0000 0100 2480

Primary Directory Address: 0000 0101 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
0000 0100 24F6				0082

LOAD PRIMARY DIRECTORY ENTRY ADDRESS REGISTER (LPDEAR)

Instruction Description

The address of the primary directory entry indexed by the second-operand halfword register is loaded into the base register specified by the first operand.

Format: RR

25	B_1	R_2

0 Bits 8 12 15

Operation: The address of the primary directory entry is formed by shifting the second operand to the left 4 bits (bits 12-15 become zeros) to convert the address from an index to an offset. This offset is then added to the origin address of the primary directory; the resulting address is then loaded into the base register specified by the first operand. The high-order 4 bits (bit 0-3) of the primary directory index identified by the second operand are ignored and treated as zeros. No storage reference is made for the primary directory entry.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Not applicable.

Program Exceptions: None.

LPDEAR Example

Op	B_1	R_2
25	6	8

0 Bits 8 12 15

Assembler: LPDEAR B_1, R_2

Machine: 2568

	Before	After
B_1(6):	xxxx xxxx xxxx	0000 0101 0820
R_2(8):	0082	0082

Primary Directory Address: 0000 0101 0000

LOAD REGISTER (LR)

Instruction Description

The second operand is placed in the first-operand register.

Format: RR

15	B_1	B_2

0 Bits 8 12 15

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

LR Example

Op	B_1	B_2
15	1	4

0 Bits 8 12 15

Assembler: LR B_1, B_2

Machine: 1514

	Before			After		
B_1(1):	xxxx	xxxx	xxxx	02A3	1234	5678
B_2(4):	02A3	1234	5678	02A3	1234	5678

LOAD SPACE OFFSET POINTER (LSOP)

Instruction Description

The 4-byte unsigned binary displacement identified by the second operand is added to the 3-byte unsigned binary space locator specified by the third operand. The resultant low-order 3 bytes are concatenated to the right of the high-order 3 bytes of the third-operand address; the result is placed into the first-operand base register.

Format: RS

93	B_1	B_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The address computation is performed in the following manner. The space locator is a 3-byte binary field located at the storage address found by concatenating hex 00 001D to the right of the high-order 3 bytes (segment group identifier) of the third-operand address. That space locator is padded on the left with hex 00 to form a 4-byte binary displacement, and is added to the second operand. If the resultant 4-byte sum exceeds hex 00FF FFFF or causes an overflow, an invalid segment group address exception is recognized and the operation is terminated. The resultant segment group offset (low-order 3 bytes) is concatenated to the right of the segment group identifier (high-order 3 bytes) of the third operand address; the result is placed in the base register specified by the first operand.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid segment group address

LSOP Example

Op 93	B_1 8	B_3 6	B_2 2	D_2 020

0 Bits 8 12 16 20 31

Assembler: LSOP B_1,D_2(B_2),B_3

Machine: 9386 2020

	Before	After
B_1(8):	xxxx xxxx xxxx	00C1 B000 08A0
B_2(2):	00C1 B000 4BC0	00C1 B000 4BC0
B_3(6):	00C1 B000 0920	00C1 B000 0920

Storage – Before and After

	0/8	2/A	4/C	6/E
00C1 B000 4BE0	0000	0800		
00C1 B000 001D			00	00A0

MOVE AND SET TAGS (MVAST)

Instruction Description

The MVAST instruction provides support to build System/38 pointers from a 16-byte value in storage. Both the area for the pointers to be built (the first operand) and that for the source 16-byte value (the second operand) are ensured to be aligned on a 16-byte boundary. The second operand value is moved to the first operand as System/38 pointer (tagged) data.

Format: SS

B4		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The second operand is placed in the first-operand location with the first operand tagged. The second operand may or may not be tagged.

If the two operands are the same quadword, then the effect is to set the tags with no change to the data.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Each operand is a quadword and must start on a quadword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

MVAST Example

Op		E	B_1	D_1	B_2	D_2
B4		0	9	006	4	F00

0 Bits 8 12 16 20 32 36 47

Assembler: MVAST $D_1(B_1)$, $D_2(B_2)$

Machine: B400 9006 4F00

B_1(9): 0001 00FF FF7A

B_2(4): 05AC 0400 0010

Storage — Before

	0/8	2/A	4/C	6/E
0001 00FF FF80	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
05AC 0400 0F10	8000	0000	0000	0000
	0412	1000	3A01	5000

Storage — After

Each word has tag set to 1

	0/8	2/A	4/C	6/E
0001 00FF FF80	8000	0000	0000	0000
	0412	1000	3A01	5000
05AC 0400 0F10	8000	0000	0000	0000
	0412	1000	3A01	5000

MOVE BYTE IMMEDIATE (MVBI)

Instruction Description

The second operand is placed in the first-operand location.

Format: SI

9B	I_2	B_1	D_1
0 Bits	8	16	20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVBI Example

Op 9B	I_2 5C	**B_1** 3	**D_1** 1A0
0 Bits	8	16	20 31

Assembler: MVBI $D_1(B_1)$, I_2

Machine: 9B5C 31A0

B_1(3): 00AC 0003 0000

Storage — Before

	0/8	**2/A**	**4/C**	**6/E**
00AC 0003 01A0	xx			

Storage — After

	0/8	**2/A**	**4/C**	**6/E**
00AC 0003 01A0	5C			

MOVE BYTE IMMEDIATE AND PROPOGATE (MVBIP)

Instruction Description

Each byte of the first operand is filled with the padding character, I_2.

Format: SI

AC	L	B_1	D_1	I_2	

0 Bits 8 16 20 32 40 47

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand must not cross a segment boundary; otherwise an effective address overflow occurs and the operation is suppressed.

Program Exceptions:

- Address Translation
- Addressing
- Effective Address Overflow

MVBIP Example

Op	L_1	B_1	D_1	I_2	
AC	05	3	C20	F9	

0 Bits 8 16 20 32 40 47

Assembler: MVBIP $D_1(L_1, B_1), I_2$

Machine: AC05 3C20 F900

$B_1(3)$: 1234 5678 0000

Storage — Before

	0/8	2/A	4/C	6/E
1234 5678 0C20	xxxx	xxxx	xxxx	

Storage — After

	0/8	2/A	4/C	6/E
1234 5678 0C20	F9F9	F9F9	F9F9	

MOVE CHARACTER REGISTER (MVCR)

Instruction Description

The second operand is placed in the first-operand location. The length is variable and is found as the contents of the third-operand byte register.

Format: SS

DB	r_3	0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Each operand field is processed left to right.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise, the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVCR Example

Op	r_3	E	B_1	D_1	B_2	D_2
DB	6	0	3	2A0	4	BC0

0 Bits 8 12 16 20 32 36 47

Assembler: MVCR $D_1(B_1)$, $D_2(B_2)$, r_3

Machine: DB60 32A0 4BC0

$r_3(6)$: 07

$B_1(3)$: 000C AA1B 0000

$B_2(4)$: 000C AC1B 0000

Storage — Before

	0/8	2/A	4/C	6/E
000C AA1B 02A0	xxxx	xxxx	xxxx	xxxx
000C AC1B 0BC0	1234	5678	9ABC	DEF0

Storage — After

	0/8	2/A	4/C	6/E
000C AA1B 02A0	1234	5678	9ABC	DEF0
000C AC1B 0BC0	1234	5678	9ABC	DEF0

MOVE CHARACTERS (MVC)

Instruction Description

The second operand is placed in the first-operand location.

Format: SS

CB	L	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Operation: Each operand field is processed left to right.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVC Example

Op	L_1	B_1	D_1	B_2	D_2
CB	07	3	2A0	4	BC0

0 Bits 8 16 20 32 36 47

Assembler: MVC $D_1(L_1, B_1), D_2(B_2)$

Machine: CB07 32A0 4BC0

B_1(3): 000C AA1B 0000

B_2(4): 000C AC1B 0000

Storage — Before

	0/8	2/A	4/C	6/E
000C AA1B 02A0	xxxx	xxxx	xxxx	xxxx
000C AC1B 0BC0	1234	5678	9ABC	DEF0

Storage — After

	0/8	2/A	4/C	6/E
000C AA1B 02A0	1234	5678	9ABC	DEF0
000C AC1B 0BC0	1234	5678	9ABC	DEF0

MOVE CHARACTERS AND TAGS (MVCAT)

Instruction Description

The MVCAT instruction moves data in storage while preserving System/38 pointers. Both operands are assumed to be aligned to the same position relative to a 16-byte boundary. System/38 pointers completely contained in the second operand are preserved in the first operand. Partial System/38 pointers are copied from the second operand into the first operand, but they do not retain the pointer attribute (copied untagged). System/38 nonpointer data (untagged data) is copied from the second operand to the first operand.

Format: SS

B5	L	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32	36 47

Operation: The second operand may contain untagged data and/or tagged quadwords. If the first byte does not start on a quadword boundary, then that partial quadword is moved to the first operand untagged. If the last byte does not end on a quadword boundary, then that partial quadword is moved to the first operand untagged. All other quadwords are moved to the first operand with their associated tags.

Each operand is processed left to right. The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable. This unpredictability resulting from operand overlap can destroy valid pointers but cannot create invalid pointers.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Both operands must be comparably aligned within a quadword; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

MVCAT Example

Op	L_1	B_1	D_1	B_2	D_2
B5	16	7	000	2	000

0 Bits 8 16 20 32 36 47

Assembler: MVCAT $D_1(L_1, B_1), D_2(B_2)$

Machine: B516 7000 2000

$B_1(7)$: 001F CB00 CDFD

$B_2(2)$: 0105 07AA 700D

The partial System/38 instruction pointer (tagged) data in the first 3 bytes of the second operand ❶ is moved to the first 3 bytes of the first operand as nonpointer (untagged) data.

The complete System/38 pointer in the next 16 bytes of the second operand ❷ is moved intact as a System/38 pointer (tagged) into the next 16 bytes of the first operand.

The next 4 bytes of nonpointer data in the second operand ❸ is moved intact as a nonpointer data into the next 4 bytes of the first operand.

Storage — After

	0/8	2/A	4/C	6/E	
001F CB00 CDFD			00	7D52	❶ (not tagged)
CE00	8000	0000	0000	0000	❷ (tagged)
	457F	8000	1537	815A	
CE10	F2F3	F0C1 ❸ (not tagged)			
0105 07AA 700D			00	7D52	❶ (tagged)
7010	8000	0000	0000	0000	❷ (tagged)
	457F	8000	1537	815A	
7020	F2F3	F0C1 ❸ (not tagged)			

MOVE CHARACTERS LONG (MVCL)

Instruction Description

The second operand is placed in the first-operand location. The remaining rightmost byte positions, if any, of the first operand location are filled with the padding character, contained in the I_3 field of the instruction.

Format: SS

EB	I_3	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Operation: The leftmost bytes of the first and second operands are located indirectly through addresses contained in storage. The first and second-operand addresses from the instruction identify an 8-byte field in storage. Bytes 0-1 of these 8-byte fields specify 1 less than the number of bytes in the operand location, and bytes 2-7 contain the addresses of the leftmost byte of the operands.

Length	Operand Address SID Offset

0 Bytes 2 8

The operation starts at the leftmost end of both fields, proceeds to the right, and ends when the end of the first-operand field is reached. If the second operand is shorter than the first operand, the remaining rightmost bytes of the first operand are filled with the padding character.

If the 8-byte field associated with the second operand contains all zeros, the second operand is assumed to be of zero length and the first operand is completely filled with the padding character. If the 8-byte field associated with the first operand contains all zeros, the operation is completed with no data moved.

The execution of the instruction is interruptable (the operation can be suspended). When an interruption occurs after a unit of operation other than the last one, the IAR (instruction address register) is not advanced to the next instruction address, the length fields are decremented by the number of bytes moved, and the address fields are incremented by the same number, so that the instruction resumes at the point of interruption. If the operation is interrupted during padding, the length field for the second operand is zero, the address field for the second operand is set to contain all zeros, and the length and address fields for the first operand reflect the extent of the padding operation.

At the completion of the operation, the length and address fields associated with the first operand contain all zeros. The length field of the second operand is decremented by the number of bytes moved and the address field is incremented by the same amount. The length and address fields associated with the second operand contain all zeros if the second operand is completely moved by the operation.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first and second-operand addresses from the instruction identify 8-byte fields in storage that begin on a word boundary and must not cross a page boundary; otherwise a specification exception occurs and the operation is suppressed. Neither data operand may cross a segment boundary; otherwise an effective address overflow exception occurs and the operation is suppressed.

The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

MVCL Example

Op	I_3	B_1	D_1	B_2	D_2
EB	FF	3	100	3	B00
0 Bits	8	16	20	32	36 47

Assembler: MVCL $D_1(B_1)$, $D_2(B_2)$, I_3

Machine: EBFF 3100 3B00

$B_1(3)$ and $B_2(3)$: 0001 1AB2 0000

Storage — Before

	0/8	2/A	4/C	6/E
0001 1AB2 0100	000F	02AC	AC1B	2100
0B00	000D	312A	F215	A0C0
02AC AC1B 2100	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
312A F215 A0C0	C9F3	21A6	ABCD	1234
	EF56	7890	FEDC	

Storage — After

	0/8	2/A	4/C	6/E
0001 1AB2 0100	0000	0000	0000	0000
0B00	0000	0000	0000	0000
02AC AC1B 2100	C9F3	21A6	ABCD	1234
	EF56	7890	FEDC	FFFF
312A F215 A0C0	C9F3	21A6	ABCD	1234
	EF56	7890	FEDC	

MOVE HALFWORD IMMEDIATE (MVHI)

Instruction Description

The second operand is placed in the first-operand location.

Format: SI

AB		0	B_1	D_1	I_2

0 Bits 8 12 16 20 32 47

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

MVHI Example

Op		E	B_1	D_1	I_2
AB		0	3	250	ABCD

0 Bits 8 12 16 20 32 47

Assembler: MVHI $D_1(B_1)$, I_2

Machine: AB00 3250 ABCD

$B_1(3)$: 002A 00A2 0000

Storage – Before

	0/8	2/A	4/C	6/E
002A 00A2 0250	xxxx			

Storage – After

	0/8	2/A	4/C	6/E
002A 00A2 0250	ABCD			

MOVE VIRTUAL PAGE WITH CORRECTED DOUBLE-BIT ERRORS SUPPRESSED (MVMC) (5382 MODELS ONLY)

Instruction Description

The second operand is placed in the first operand location. The IMPI processor must be in machine check mode or a machine check will occur.

Format: RR

1D	B_1	B_2

0 Bits 8 12 15

Operation: Each operand field is processed left to right with tags. If a hard-hard double-bit main storage error occurs on the second operand during this operation and is corrected, the machine check will be suppressed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands must be different main storage frames, page-aligned, and 512 bytes long or a machine check will occur. The second operand address must be a V=V address or a machine check will occur.

Program Exception: None.

MVMC Example

Op	B_1	B_2
1D	7	4

0 Bits 8 12 15

Assembler: MVMC B_1, B_2

Machine: 1D74

B_1(7): 0000 0139 CA00

B_2(4): 000B D065 3600

Storage - Before

	0/8	2/A	4/C	6/E
0000 0139 CA00	xxxx	xxxx	xxxx	xxxx
CBF8	xxxx	xxxx	xxxx	xxxx
000B D065 3600	C1F3	ACBD	0123	ABCD
37F8	FEDB	8765	C2C5	F0F5

Storage - After

	0/8	2/A	4/C	6/E
0000 0139 CA00	C1F3	ACBD	0123	ABCD
CBF8	FEDB	8765	C2C5	F0F5
000B D065 3600	C1F3	ACBD	0123	ABCD
37F8	FEDB	8765	C2C5	F0F5

This page is intentionally left blank.

MOVE NUMERIC TO NUMERIC (MVNN)

Instruction Description

The numeric half of the 1-byte second operand is placed in the numeric half of the 1-byte first operand.

Format: SS

BA		0	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The numeric half of the byte is the rightmost 4 bits.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVNN Example

Op BA		E 0	B_1 3	D_1 100	B_2 3	D_2 400
0 Bits	8	12	16	20	32	36 47

Assembler: MVNN $D_1(B_1)$, $D_2(B_2)$

Machine: BA00 3100 3400

B_1(3) and B_2(3): 1234 5678 0000

Storage—Before

	0/8	2/A	4/C	6/E
1234 5678 0100	xx			
1234 5678 0400	F3			

Storage—After

	0/8	2/A	4/C	6/E
1234 5678 0100	X3			
1234 5678 0400	F3			

MOVE NUMERIC TO ZONE (MVNZ)

Instruction Description

The numeric half of the second operand is placed in the zone half of the first operand.

Format: SS

BB		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The numeric half of the byte is the rightmost 4 bits, and the zone half byte is the leftmost 4 bits.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVNZ Example

Op BB		E 0	B_1 3	D_1 100	B_2 3	D_2 400

0 Bits 8 12 16 20 32 36 47

Assembler: MVNZ $D_1(B_1), D_2(B_2)$

Machine: BB00 3100 3400

B_1(3) and B_2(3): 1234 5678 0000

Storage – Before

	0/8	2/A	4/C	6/E
1234 5678 0100	xx			
1234 5678 0400	F1			

Storage – After

	0/8	2/A	4/C	6/E
1234 5678 0100	1x			
1234 5678 0400	F1			

This page is intentionally left blank.

MOVE PACKED SHIFTED (MVPS)

Instruction Description

The second operand is shifted as specified by the contents of byte register hex F and is placed in the first-operand location.

Format: SS

FB	L_1	L_2	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The result is padded on the right or left with zeros, as required, and the sign is right-adjusted. The second operand is unchanged by the operation except when the operands overlap. The contents of byte register hex F are unchanged by the operation.

Bits 2-7 of byte register hex F indicate the direction of the shift and the number of digit positions to be shifted. The remaining bits in the register are ignored. When bit 2 of byte register hex F is zero, a left shift is specified, and when bit 2 is 1, a right shift is specified. Bits 3-7 of byte register hex F are treated as a true binary number specifying the number of digit positions to be shifted (ranging in value from zero to 31).

The operation is performed as if the entire second operand was shifted prior to modifying any byte of the first operand.

The second operand is treated as a packed decimal format field and is checked for a valid digit code. An improper code causes a data exception and the operation is terminated with the first operand unchanged. Only the digits of the second operand are shifted; its sign is right-aligned in the first-operand field and, if necessary, changed to the preferred sign code. Zeros are supplied to the digits of the first operand that do not receive a digit from the second operand.

During right shifts, the digits shifted out of the rightmost digit position are ignored and lost. For a left shift, all significant digits shifted are placed in the first-operand field except when the first-operand field is too short to include all shifted significant digits. Then the rightmost digits of the shifted second operand are placed in the first-operand location and a decimal overflow occurs.

Overflow: See *Operation*.

Sign Code: The second operand is checked for a valid sign code. An improper code causes a data exception and the operation is terminated with the first operand unchanged.

In the absence of a decimal overflow, the sign of a zero result is made positive. With a decimal overflow, the sign of a zero result is the same as the original sign, but the code is the preferred sign code.

See *Operation* for further information.

Condition Code:

0	Result	= 0
1	Result	< 0
2	Result	> 0
3	--	

A shift of zero can be used to set the condition code based on the value of the field.

Carry: Not applicable.

Boundary Requirements: The first and second operands can overlap. All combinations of overlap and shift are allowed. No data alignment is required for either operand.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Effective address overflow

MVPS Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
FB	F	F	3	CB0	4	F00

0 Bits 8 12 16 20 32 36 47

Assembler: MVPS $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: FBFF 3CB0 4F00

B_1(3): 0000 0A10 0000

B_2(4): 0000 1B20 0000

r(F): 08

Storage — Before

	0/8	2/A	4/C	6/E
0000 0A10 0CB0	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
0000 1B20 0F00	0000	0000	1234	5678
	9012	3456	7890	123F

Storage — After

	0/8	2/A	4/C	6/E
0000 0A10 0CB0	1234	5678	9012	3456
	7890	1230	0000	000F
0000 1B20 0F00	0000	0000	1234	5678
	9012	3456	7890	123F

	Before	After
Condition Code:	x	2

MOVE PACKED SHIFTED ZERO (MVPSZ)

Instruction Description

Byte register hex F is first set to zero and then the second operand is shifted as specified by the contents of byte register hex F and is placed in the first-operand location.

Format: SS

FC	L_1	L_2	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The result is padded on the right or left with zeros, as required and the sign is right-adjusted. The second operand is unchanged by the operation except when the operands overlap. The contents of byte register hex F are set to zero before starting the operation.

Bits 2 through 7 of byte register hex F indicate the direction of the shift and the number of digit positions to be shifted. The remaining bits in the register are ignored. When bit 2 of byte register hex F is zero, a left shift is specified, and when bit 2 is 1, a right shift is specified. Bits 3 through 7 of byte register hex F are treated as a true binary number specifying the number of digit positions to be shifted (ranging in value from zero to 31).

The operation is performed as if the entire second operand was shifted prior to modifying any byte of the first operand.

The second operand is treated as a packed decimal format field and is checked for a valid digit code. An improper code causes a data exception and the operation is terminated with the first operand unchanged. Only the digits of the second operand are shifted; its sign is right-aligned in the first operand field and, if necessary, changed to the preferred sign code. Zeros are supplied to the digits of the first operand that do not receive a digit from the second operand.

During right shifts, the digits shifted out of the rightmost digit position are ignored and lost. For a left shift, all significant digits shifted are placed in the first-operand field except when the first-operand field is too short to include all shifted significant digits. Then the rightmost digits of the shifted second operand are placed in the first-operand location and a decimal overflow occurs.

Overflow: See *Operation.*

Sign Code: The second operand is checked for a valid sign code. An improper code causes a data exception and the operation is terminated with the first operand unchanged.

In the absence of a decimal overflow, the sign of a zero result is made positive. With a decimal overflow, the sign of a zero result is the same as the original sign, but the code is the preferred sign code.

See *Operation* for further information.

Condition Code:

0 Result = 0
1 Result < 0
2 Result > 0
3 --

A shift of zero can be used to set the condition code based on the value of the field.

Carry: Not applicable.

Boundary Requirements: The first and second operands can overlap. All combinations of overlap and shift are allowed. No data alignment is required for either operand.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Effective address overflow

MVPSZ Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
FC	F	F	3	CBO	4	F00

0 Bits 8 12 16 20 32 36 47

Assembler: MVPSZ $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: FCFF 3CB0 4F00

B_1(3): 0000 0A10 0000

B_2(4): 0000 1B20 0000

	Before	After
r(F):	08	00

Storage — Before

	0/8	2/A	4/C	6/E
0000 0A10 0CB0	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
0000 1B20 0F00	0000	0000	1234	5678
	9012	3456	7890	123F

Storage — After

	0/8	2/A	4/C	6/E
0000 0A10 0CB0	0000	0000	1234	5678
	9012	3456	7890	123F
0000 1B20 0F00	0000	0000	1234	5678
	9012	3456	7890	123F

	Before	After
Condition Code:	x	2

MOVE ZONE TO NUMERIC (MVZN)

Instruction Description

The zone half of the second operand is placed in the numeric half of the first operand.

Format: SS

BC		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The zone half of the byte is the leftmost 4 bits, and the numeric half of the byte is the rightmost 4 bits.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVZN Example

Op		E	B_1	D_1	B_2	D_2
BC		0	3	560	3	9A0

0 Bits 8 12 16 20 32 36 47

Assembler: MVZN $D_1(B_1)$, $D_2(B_2)$

Machine: BC00 3560 39A0

B_1(3) and B_2(3): 00DC 0DB1 0000

Storage — Before

	0/8	2/A	4/C	6/E
00DC 0DB1 0560	xx			
00DC 0DB1 09A0	F5			

Storage — After

	0/8	2/A	4/C	6/E
00DC 0DB1 0560	xF			
00DC 0DB1 09A0	F5			

MOVE ZONE TO ZONE (MVZZ)

Instruction Description

The zone half of the second operand is placed in the zone half of the first operand.

Format: SS

BD		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The zone half of the byte is the leftmost 4 bits.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

MVZZ Example

Op		E	B_1	D_1	B_2	D_2
BD		0	3	560	3	9A0

0 Bits 8 12 16 20 32 36 47

Assembler: MVZZ $D_1(B_1)$, $D_2(B_2)$

Machine: BD00 3560 39A0

$B_1(3)$ and $B_2(3)$: 00DC 0DB1 0000

Storage — Before

	0/8	2/A	4/C	6/E
00DC 0DB1 0560	xx			
00DC 0DB1 09A0	F5			

Storage — After

	0/8	2/A	4/C	6/E
00DC 0DB1 0560	Fx			
00DC 0DB1 09A0	F5			

MULTIPLY HALFWORD STORAGE (MHS)

Instruction Description

The product of the multiplier (the second operand) and the multiplicand (the first operand) replaces the multiplicand.

Format: SS

DC		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Both multiplier and multiplicand are 16-bit signed binary integers. The product is always a 32-bit signed binary integer and occupies a word of storage at the first-operand location. The multiplicand is taken from the second halfword of the first operand. The contents of the first halfword are ignored unless it contains the multiplier.

Overflow: An overflow cannot occur.

Sign Code: The sign of the product is determined (by the rules of algebra) from the multiplier and multiplicand signs.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand must start on a word boundary; the halfword storage operand must start on a halfword boundary; and the word storage operand must start on a word boundary. Otherwise a specification exception occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

MHS Example

Op DC		E 0	B_1 3	D_1 150	B_2 3	D_2 160

0 Bits 8 12 16 20 32 36 47

Assembler: MHS $D_1(B_1)$, $D_2(B_2)$

Machine: DC00 3150 3160

B_1(3) and B_2(3): 1A45 BC3D 0000

Storage — Before

	0/8	2/A	4/C	6/E
1A45 BC3D 0150	0000	FFF4	(–12)	
1A45 BC3D 0160	0002		(+2)	

Storage — After

	0/8	2/A	4/C	6/E
1A45 BC3D 0150	FFFF	FFE8	(–24)	
1A45 BC3D 0160	0002		(+2)	

MULTIPLY LONG FLOAT (MLF)

Instruction Description

The second operand is multiplied by the first operand (two-operand format) or the third is multiplied by the second operand (three-operand format), and the product is placed in the first operand location.

Format: SS

CE	B_3	3	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: A two-operand or three-operand format can be specified. A two-operand format is used if base register 0 is specified for the third operand. A three-operand format is used if one of the base registers hex 1 through F is specified for the third operand. Interchanging the two source operands does not affect the value of the product. However, the first operand data is overwritten by the result.

Multiplication of two floating-point numbers uses exponent addition and significand multiplication. The sum of the signed (unbiased) exponents of the source operands is used as the exponent of the intermediate product. This applies for denormalized and normalized numbers.

The multiplication of the significands is performed as if to infinite precision to form the intermediate product significand. This product is normalized, if necessary, before rounding. The rounding is performed according to the mode speciified in the task dispatching element.

When either operand is 0, the product is made 0, and no exceptions occur.

If a masked non-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the product. The source operands are checked for this value in the order of their specification with the masked not-a-number with the larger fraction value being provided as the product.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the product is determined by the rules of algebra. This is the exclusive OR of the signs of the source operands. This applies to the products of 0, infinity, and normal finite numbers.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: All operands must be fullword aligned, otherwise, a specification exception occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

Programming Note: The following is a summary of the products for various combinations of operands.

Product	First Source (Multiplicand)	Second Source (Multiplier)
+0	+Real number ≠ 0	+0
+0	+0	+0
+0	-Real number ≠ 0	-0
+0	-0	-0
+0	+0	+Real number ≠ 0
+0	-0	-Real number ≠ 0
-0	+Real number ≠ 0	-0
-0	+0	-0
-0	-Real number ≠ 0	+0
-0	-0	+0
-0	-0	+Real number ≠ 0
-0	+0	-Real number ≠ 0
+0 (see note)	+Small real number ≠ 0	+Small real number ≠ 0
+0 (see note)	-Small real number ≠ 0	-Small real number ≠ 0
-0 (see note)	+Small real number ≠ 0	-Small real number ≠ 0
-0 (see note)	-Small real number ≠ 0	+Small real number ≠ 0
Invalid operand exception	+Infinity	+0
Invalid operand exception	+Infinity	-0
Invalid operand exception	-Infinity	+0
Invalid operand exception	-Infinity	-0
+Infinity	+Infinity	+Real number ≠ 0
+Infinity	-Infinity	-Real number ≠ 0
+Infinity	+Real number ≠ 0	+Infinity
+Infinity	-Real number ≠ 0	-Infinity
+Infinity	+Infinity	+Infinity
+Infinity	-Infinity	-Infinity
-Infinity	-Infinity	+Real number ≠ 0
-Infinity	+Infinity	-Real number ≠ 0
-Infinity	-Real number ≠ 0	+Infinity

Product	First Source (Multiplicand)	Second Source (Multiplier)
-Infinity	+Real number ≠ 0	-Infinity
-Infinity	+Infinity	-Infinity
-Infinity	-Infinity	+Infinity
Masked not-a-number	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-number	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operand exception	Unmasked not-a-number	Any
Invalid operand exception	Any	Unmasked not-a-number

Legend:

Not not-a-number = Anything but a not-a-number.

Any = Any floating-point field value.

Note: For two small valued real numbers that are not equal to 0, a floating-point underflow that has a zero product rather than a denormalized product can occur.

MLF Example

Op	B_3	E	B_1	D_1	B_2	D_2
CE	3	3	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: MLF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: CE33 4050 4060

$B_3(3)$: 0010 0200 0070

$B_1(4)$ and $B_2(4)$: 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	C080	0230	0000	0000
0010 0200 0070	5080	0230	0000	0000

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	D110	0460	4C90	0000
0010 0200 0060	C080	0230	0000	0000
0010 0200 0070	5080	0230	0000	0000

Condition Code: Not changed.

MULTIPLY PACKED (MP)

Instruction Description

The product of the multiplier (the second operand) and multiplicand (the first operand) replaces the multiplicand.

Format: SS

F3	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The multiplier size is limited to 15 digits and sign and must be less than the multiplicand size. A length code (L_2) larger than seven or larger than or equal to the length code L_1 causes a specification exception.

The multiplicand must have at least as many bytes of left zeros as the multiplier field size, in bytes; otherwise a data exception occurs and the operation is terminated. The maximum product size is 31 digits. At least one leftmost digit of the product field is zero.

All operands and results are treated as signed integers, right-aligned in their field.

Digit codes are checked for validity; invalid codes cause a data exception, and the operation is terminated.

Overflow: The definition of the multiplicand field ensures that no product overflow can occur.

Sign Code: The sign of the product (the product is assigned a preferred sign) is determined by the rules of algebra from the multiplier and multiplicand signs, except that a zero result is always positive. The sign is encoded as 1111 (hex F) for a positive result and 1101 (hex D) for a negative result. The sign codes are checked for validity; an invalid code causes a data exception, and the operation is terminated.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The multiplier and product fields can overlap only if their rightmost bytes coincide. Improperly overlapping fields cause a data exception, and the operation is terminated.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow
- Specification

MP Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
F3	5	3	6	120	7	240

0 Bits 8 12 16 20 32 36 47

Assembler: MP $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: F353 6120 7240

B_1(6): 0A38 2310 0000

B_2(7): 0A38 6405 0000

Storage — Before

	0/8	2/A	4/C	6/E
0A38 2310 0120	0000	0210	261D	
0A38 6405 0240	0000	004D		

Storage — After

	0/8	2/A	4/C	6/E
0A38 2310 0120	0000	0841	044F	
0A38 6405 0240	0000	004D		

MULTIPLY PACKED LONG (MPL)

Instruction Description

The product of the multiplier (the second operand) and the multiplicand (the first operand) replaces the multiplicand.

Format: SS

F9	L_1	L_2	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Note: The Multiply Packed Long instruction is implemented in vertical microcode (VMC) and is treated as an implicit SVL by the IMP processor. The operation code is used as the index into the SVL table, as described in the section on SVLs in Chapter 6.

Operation: The product is placed into the first-operand field and can contain a maximum of 63 digits and sign (L_1+L_2+2=32 bytes). The multiplicand occupies the leftmost L_1+1 bytes of the first operand. The remaining L_2+1 bytes of the first operand are not used as multiplicand data. At least one leftmost digit of the product field is zero.

All operands and results are treated as signed integers, right-aligned in their fields. Digit codes are checked for validity; an invalid code causes a data exception, and the operation is terminated.

Overflow: The definition of the multiplicand field ensures that no product overflow can occur.

Sign Code: The sign of the product (the product is assigned a preferred sign) is determined by the rules of algebra from the multiplier and multiplicand signs, except that a zero result is always positive. A positive sign is encoded as 1111 (hex F); a negative sign is encoded as 1101 (hex D).

The sign codes are checked for validity; an invalid code causes a data exception, and the operation is terminated.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The multiplier and product fields may overlap if their rightmost bytes coincide. Improperly overlapping fields causes a data exception, and the operation is terminated.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow

Notes:

1. L_1 specifies 1 less than the number of bytes by which the length of the first operand exceeds the length of the second operand.
2. L_2 specifies 1 less than the length in bytes of the multiplier. The multiplier can contain a maximum of 31 digits and sign.

MPL Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
F9	3	1	3	550	4	A10

0 Bits 8 12 16 20 32 36 47

Assembler: MPL $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: F931 3550 4A10

B_1(3): 0000 ABCD 0000

B_2(4): 0000 BCDE 0000

Storage — Before

	0/8	2/A	4/C	6/E
0000 ABCD 0550	1234	567F	0000	
0000 BCDE 0A10	123F			

Storage — After

	0/8	2/A	4/C	6/E
0000 ABCD 0550	0015	1851	741F	
0000 BCDE 0A10	123F			

MULTIPLY SHORT FLOAT (MSF)

Instruction Description

The second operand is multiplied by the first operand (two-operand format) or the third is multiplied by the second operand (three-operand format), and the product is placed in the first operand location.

Format: SS

AE	B_3	3	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: A two-operand or three-operand format can be specified. A two-operand format is used if base register 0 is specified for the third operand. A three-operand format is used if one of the base registers hex 1 through hex F is specified for the third operand. Interchanging the two source operands does not affect the value of the product. However, the first operand data is overwritten by the result.

Multiplication of two floating-point numbers uses exponent addition and significand multiplication. The sum of the signed (unbiased) exponents of the source operands is used as the exponent of the intermediate product. This applies for denormalized and normalized numbers.

The multiplication of the significands is performed as if to infinite precision to form the intermediate product significand. This product is normalized, if necessary, before rounding. The rounding is performed according to the mode specified in the task dispatching element.

When either operand is 0, the product is made 0, and no exceptions occur.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the product. The source operands are checked for this value in the order of their specification with the masked not-a-number with the larger fraction value being provided as the product.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the product is determined by the rules of algebra. This is the exclusive OR of the signs of the source operands. This applies to the products of 0, infinity, and normal finite numbers.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: All operands must be fullword aligned, otherwise, a specification occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

Programming Note: The following is a summary of the products for various combinations of operands.

Product	First Source (Multiplicand)	Second Source (Multiplier)
+0	+Real number ≠ 0	+0
+0	+0	+0
+0	-Real number ≠ 0	-0
+0	-0	-0
+0	+0	+Real number ≠ 0
+0	-0	-Real number ≠ 0
-0	+Real number ≠ 0	-0
-0	+0	-0
-0	-Real number ≠ 0	+0
-0	-0	+0
-0	-0	+Real number ≠ 0
-0	+0	-Real number ≠ 0
+0 (see note)	+Small real number ≠ 0	+Small real number ≠ 0
+0 (see note)	-Small real number ≠ 0	-Small real number ≠ 0
-0 (see note)	+Small real number ≠ 0	-Small real number ≠ 0
-0 (see note)	-Small real number ≠ 0	+Small real number ≠ 0
Invalid operand exception	+Infinity	+0
Invalid operand exception	+Infinity	-0
Invalid operand exception	-Infinity	+0
Invalid operand exception	-Infinity	-0
+Infinity	+Infinity	+Real number ≠ 0
+Infinity	-Infinity	-Real number ≠ 0
+Infinity	+Real number ≠ 0	+Infinity
+Infinity	-Real number ≠ 0	-Infinity
+Infinity	+Infinity	+Infinity
+Infinity	-Infinity	-Infinity
-Infinity	-Infinity	+Real number ≠ 0
-Infinity	+Infinity	-Real number ≠ 0

Note: For two small values of real numbers that are not equal to 0, a floating-point underflow which has a zero product rather than a denormalized product can occur.

Product	First Source (Multiplicand)	Second Source (Multiplier)
-Infinity	-Real number ≠ 0	+Infinity
-Infinity	+Real number ≠ 0	-Infinity
-Infinity	+Infinity	-Infinity
-Infinity	-Infinity	+Infinity
Masked not-a-number	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-anumber	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operand exception	Unmasked not-a-number	Any
Invalid operand exception	Any	Unmasked not-a-number

Legend:

Not not-a-number = Anything but not-a-number

Any = Any floating-point field value

MSF Example

Op	B	E	B_1	D_1	B_2	D_2
AE	3	3	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: MSF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: AE33 4050 4060

B_3(3): 0010 0200 0070

B_1(4) and B_2(4): 0010 0200 0000

Storage – Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	4000	0000		
0010 0200 0070	3F80	0000		

Storage – After

	0/8	2/A	4/C	6/E
0010 0200 0050	4000	0000		
0010 0200 0060	4000	0000		
0010 0200 0070	3F80	0000		

Condition Code: Not changed.

MULTIPLY WORD STORAGE (MWS)

Instruction Description

The product of the multiplier (the second operand) and the multiplicand (the first operand) replaces the multiplicand.

Format: SS

EC		0	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Both multiplier and multiplicand are 32-bit signed binary integers. The product is always a 64-bit signed binary integer and occupies 8 bytes of storage at the first-operand location. The mulitiplicand is taken from the second word of the first operand. The contents of the first word are ignored unless it contains the multiplier.

Overflow: An overflow cannot occur.

Sign Code: The sign of the product is determined by the rules of algebra from the multiplier and multiplicand signs.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Both operands must start on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Data
- Effective address overflow

MWS Example

Op EC		E 0	B_1 3	D_1 100	B_2 3	D_2 A00

0 Bits 8 12 16 20 32 36 47

Assembler: MWS $D_1(B_1)$, $D_2(B_2)$

Machine: EC00 3100 3A00

$B_1(3)$ and $B_2(3)$: 0000 1234 0000

Storage — Before

```
                  0/8    2/A    4/C     6/E
0000 1234 0100    0000   0000   0000   1F40
                                  (8000)
0000 1234 0A00    FFFF   FFE4   (-28)
```

Storage — After

```
                  0/8    2/A    4/C     6/E
0000 1234 0100    FFFF   FFFF   FFFC   9500
                                (-224 000)
0000 1234 0A00    FFFF   FFE4   (-28)
```

OR BYTE (OB)

Instruction Description

The inclusive OR of the first and second operands is placed in the first-operand register.

Format: RS

79	r_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

OB Example

Op	r_1	E	B_2	D_2
79	5	0	3	1A0

0 Bits 8 12 16 20 31

Assembler: OB r_1, $D_2(B_2)$

Machine: 7950 31A0

B_2(3): 0001 0A01 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
0001 0A01 01A0	55			

	Before	After
r_1(5):	AA	FF
Condition Code:	x	1

OR BYTE IMMEDIATE (OBI)

Instruction Description

The inclusive OR of the first and second operands is placed in the first-operand location.

Format: SI

99	I_2	B_1	D_1

0 Bits 8 16 20 31

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

OBI Example

Op 99	I_2 A5	B_1 3	D_1 1A0

0 Bits 8 16 20 31

Assembler: $D_1(B_1)$, I_2

Machine: 99A5 31A0

B_1(3): 0A10 B0C5 0000

Storage — Before

	0/8	2/A	4/C	6/E
0A10 B0C5 01A0	AA			

Storage — After

	0/8	2/A	4/C	6/E
0A10 B0C5 01A0	AF			

	Before	After
Condition Code:	x	1

OR BYTE REGISTER (OBR)

Instruction Description

The inclusive OR of the first and second operands is placed in the first-operand register.

Format: RR

19	r_1	r_2

0 Bits 8 12 15

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

OBR Example

Op 19	r_1 3	r_2 4

0 Bits 8 12 15

Assembler: OBR r_1, r_2

Machine: 1934

	Before	After
r_1(3):	59	FD
r_2(4):	A4	A4
Condition Code:	x	1

OR BYTE REGISTER IMMEDIATE (OBRI)

Instruction Description

The inclusive OR of the first and second operands is placed in the first-operand register.

Format: RI

49	r_1	0	I_2	

0 Bits 8 12 16 24 31

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Codes: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

OBRI Example

Op	r_1	E	I_2	
49	3	0	33	

0 Bits 8 12 16 24 31

Assembler: r_1, I_2

Machine: 4930 3300

	Before	After
r_1(3):	55	77
Condition Code:	x	1

OR CHARACTERS (OC)

Instruction Description

The inclusive OR of the first and second operands is placed in the first-operand location.

Format: SS

C9	L	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Each operand field is processed left to right.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result	= 0
1	Result	≠ 0
2	--	
3	--	

Carry: Not applicable.

Boundary Requirements: The operands can overlap if the leftmost byte of the first operand is coincident with or to the left of the leftmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable. Neither operand may cross a segment boundary; otherwise an effective address overflow occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

OC Example

Op	L_1	B_1	D_1	B_2	D_2
C9	03	3	2C0	3	A50

0 Bits 8 16 20 32 36 47

Assembler: OC $D_1(L_1, B_1)$, $D_2(B_2)$

Machine: C903 32C0 3A50

B_1(3) and B_2(3): 001A 5C9E 0000

Storage — Before

	0/8	2/A	4/C	6/E
001A 5C9E 02C0	1234	5678		
001A 5C9E 0A50	5678	9ABC		

Storage — After

	0/8	2/A	4/C	6/E
001A 5C9E 02C0	567C	DEFC		
001A 5C9E 0A50	5678	9ABC		

	Before	After
Condition Code:	x	1

OR HALFWORD (OH)

Instruction Example

The inclusive OR of the first and second operands is placed in the first-operand register.

Format: RS

80	R_1	3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

OH Example

Op	R_1	E	B_2	D_2
80	3	3	4	000

0 Bits 8 12 16 20 31

Assembler: OH R_1, $D_2(B_2)$

Machine: 8033 4000

$B_2(4)$: 0AB1 000A 1000

Storage — Before and After

	0/8	2/A	4/C	6/E
0AB1 000A 1000	A5A5			

	Before	After
$R_1(3)$:	5A5A	FFFF
Condition Code:	x	1

OR HALFWORD REGISTER (OHR)

Instruction Description

The inclusive OR of the first and second operands is placed in the first-operand register.

Format: RR

29	R_1	R_2

0 Bits 8 12 15

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

OHR Example

Op 29	R_1 3	R_2 4

0 Bits 8 12 15

Assembler: OHR R_1, R_2

Machine: 2934

	Before	After
R_1(3):	1234	567C
R_2(4):	5678	5678
Condition Code:	x	1

OR HALFWORD REGISTER IMMEDIATE (OHRI)

Instruction Description

The exclusive OR of the first and second operands is placed in the first-operand register.

Format: RI

59	R_1	0	I_2
0 Bits	8	12	16 31

Operation: Operands are treated as logical quantities, and the inclusive OR is applied bit by bit. A bit position in the result is set if the corresponding bit position in one or both operands is set; otherwise the result bit is reset.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Result = 0
1	Result ≠ 0
2	--
3	--

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

OHRI Example

Op	R_1	E	I_2
59	3	0	1357
0 Bits	8	12	16 31

Assembler: OHRI R_1, I_2

Machine: 5930 1357

	Before	After
R_1:	2468	377F
Condition Code:	x	1

PERFORM PAGING REQUEST (PPR)

Instruction Description

Format: SS

E8	I_3	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Note: The Perform Paging Request instruction is implemented in vertical microcode (VMC) and is treated as an implicit SVL by the IMP processor. The operation code is used as an index into the SVL table, as described in the section on SVLs in Chapter 6.

Operation: The operation specified by the third operand is performed on all pages within a contiguous range of virtual storage addresses.

If the high-order bit of I_3 is zero, the first operand occupies 6 bytes in storage and contains the beginning address of the range of virtual addresses. The second operand occupies 6 bytes in storage and contains the last address of the range of virtual addresses.

If the high-order bit of I_3 is 1, the first and second operand occupy 6 bytes of storage and point to the beginning of the access group table of contents.

The operation specified by the third operand, I_3, is performed only in page size increments. Bits 0-38 of both the first and second operands identify the first and last pages in the range of pages that participate in the operation. Bits 39-47 of both operands are not used.

Byte register hex F, which is used as an operand for bring and clear requests, holds either the storage pool ID or zero. If not zero, the storage pool indicated is used to allocate page frames to satisfy the request. If zero, the storage pool in the TDE (task dispatching element) is used to allocate page frames. If the storage pool cannot satisfy a request that specifies an increment pin count (no unpinned pages available), an invalid pool state exception occurs and the operation is suppressed.

A specification exception occurs and the operation is suppressed for any of the following conditions:

- The first and-second operand addresses are not within the same segment or segment group.

- The second-operand address is less than the first-operand address.

- Either the first or second-operand address is a V=R (virtual equals real) address.

- The third-operand is invalid.

If asynchronous request and increment/decrement pin count are both specified on a bring, purge, or write, a specification exception occurs.

If the segment or segment group specified by the contents of the first operand does not exist, an invalid segment exception occurs and the operation is suppressed. If the page specified by the contents of the second operand does not exist, an invalid page exception occurs and the operation is suppressed. If a permanent I/O error occurs while trying to read a page from auxiliary storage, a page read error exception occurs and the operation is completed abnormally. These last three exceptions can occur for synchronous requests only. If a write to auxiliary storage of a pinned page is attempted, an invalid write request occurs and the operation is completed abnormally.

If bring or purge access group is indicated by I_3, invalid page, invalid segment, or permanent I/O error exceptions are signaled if the access group table of contents indicated by the first and second operands is smaller than expected, is nonexistent, or has an I/O error, respectively.

The third operand is interpreted as follows:

Bits	Code	Operation/Description
0		0 Any function described in bits 3-5 may be performed on an access group. 1 Perform the function described in bits 3-5 on an access group. Only bring and purge are valid and bits 6 and 7 must be zero.
1-2	-	Reserved: must be zero.
3-5	000	Used during page faults.
	001	Bring: copy the specified page(s) from auxiliary to main storage.
	010	Clear: provide zeroed main storage frames for the specified page(s) occurs and the prior contents are lost.
	011	Invalid.
	100	Write: copy the specified page(s), if changed, from main storage to auxiliary storage.
	101	Purge: copy the specified page(s), if changed, from main storage to auxiliary storage. In addition, make all the frames associated with the pages available for reassignment.
	110	Invalid.
	111	Remove: remove the specified page(s) from main storage. Do not copy changed pages back to auxiliary storage. Make the main storage frames immediately available for reassignment; the page(s) are no longer addressable in main storage. The contents of the pages are set to an undefined state by the remove.
6		For Bring, Write, and Purge: 0 Synchronous request (wait). 1 Asynchronous request (do not wait). For Clear and Remove: 0 Must be zero (always synchronous).
7		For Bring and Clear: 0 Leave pin count unchanged. 1 Increment pin count by 1. For Write, Purge, and Remove: 0 Leave pin count unchanged. 1 Decrement pin count by 1.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first and second operands must both begin on a word boundary; otherwise a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Effective address overflow
- Invalid page (synchronous requests only)
- Invalid pin request (synchronous requests only)
- Invalid pool state (synchronous requests only)
- Invalid segment (synchronous requests only)
- Invalid write request (synchronous requests only)
- Page read error (synchronous requests only)
- Specification

PPR Example

Op	I_3	B_1	D_1	B_2	D_2
E8	08	3	600	3	9FF

0 Bits 8 16 20 32 36 47

Assembler: PPR $D_1(B_1)$, $D_2(B_2)$, I_3

Machine: E808 3600 39FF

$B_1(3)$ and $B_2(3)$: 6D00 AC00 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
6D00 AC00 0600	xxxx	xxxx	xxxx	xxxx
09F8	xxxx	xxxx	xxxx	xxxx

0600 through 09FF = xxxx

Storage — Before and After

	0/8	2/A	4/C	6/E
6D00 AC00 0600	0000	0000	0000	0000
09F8	0000	0000	0000	0000

0600 through 09FF = 0000

This page is intentionally left blank.

READ REFERENCE AND CHANGE AND RESET REFERENCE (RRCRR)

Instruction Description

The RRCRR instruction aids the page replacement process. A reference bit is set by the hardware each time a page is referenced. The bit is periodically examined and reset by the page replacement process. The page replacement process is informed in this way whether the page was referenced since it was last examined. A second hardware bit called the change bit is set whenever an instruction alters the contents of a page. This enables the page replacement process to know whether or not to page out the page he wishes tó steal before destroying the addressability.

The RRCRR instruction takes care of these functions as well as accounting for the fact that the current state of the change flag may be in the internal hardware LB (lookaside buffer), rather than the PD (primary directory) entry itself.

Format: SI

83		0	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The reference and change bits of a PD entry are read, and the result determines the setting of the condition code. If a copy of the PD entry resides in the LB, that copy's change bit is ORed into bit 42 of the actual PD entry. The copy is then removed from the LB. Subsequently, the reference bit is reset in the PD entry.

The operand for this instruction is a primary directory index value. This index, when multiplied by 16 (ignoring any bits carried out of the halfword) and added to the base address of the primary directory, addresses the PD entry to be examined.

The primary directory entry is identified by the first operand. Bits 0-15 of this halfword storage operand are used as the primary directory entry index value. If the index value specifies a directory entry that contains V=R addresses, a specification exception occurs and the operation is suppressed. The high-order 4 bits (bits 0-3) of the PD index value are ignored and treated as zeros.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Reference bit zero, change bit zero
1	Reference bit zero, change bit one
2	Reference bit one, change bit zero
3	Reference bit one, change bit one

Carry: Not applicable.

Boundary Requirements: The first operand occupies 2 bytes in storage and must begin on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

RRCRR Example

Op		E_1	B_1	D_1
83		0	1	330

0 Bits 8 12 16 20 31

Assembler: RRCRR $D_1(B_1)$

Machine: 8300 1330

	Before	After
B_1(1):	0000 0407 8300	0000 0407 8300
Condition Code:	x	3

PD (primary directory) base address 0000 0101 0000

The halfword at the first-operand effective address identifies a PD entry index. The processor converts the PD index (hex 00D3) to a PD offset by multiplying the index by 16.

The PD entry address is the sum of the generated offset (hex D300) and the PD base address, which is implicitly supplied by the machine.

Storage – Before and After

	0/8	2/A	4/C	6/E
0000 0407 8630	00D3			

The first 5 bytes of the PD entry **1** (base register contents plus displacement) are verified to be virtual = virtual.

Primary Directory

0000 0101 0000	Segment Identifier 0 Bits 31	PID 32 39	Status 40 47	Index 48 59	Reserved 60 62	EOC 63	PINCNT 64 71	Reserved 72 120
D300	0000 0000 0000 0000 0000 0101 0000 0101	0101 1100	1100 0000	1000 0100 0000	000	0	0000 0000	
	1		**2**					

The values of the reference and change bits in the LB (lookaside buffer) are inclusively ORed into the PD entry **2**. The values are used to set the condition code. The reference bits in both the LB and the PD entry are reset.

RECEIVE COUNT (RECC)

Instruction Description

The current value of the counter designated by the operand is decremented by the limit value in the SRC (send/receive counter) when the current value of the counter is equal to or greater than the limit value of the counter.

Format: SI

67		0	B_1	D_1

0 Bits 8 12 16 24 31

Operation: Normal instruction sequencing proceeds with the updated instruction address.

If the current value of the counter is less than the limit value of the counter, the instruction is nullified, and the TDE (task dispatching element) of the task issuing the instruction is dequeued from the TDQ (task dispatching queue) and is enqueued onto the SRC wait queue in the TDE priority sequence. Bytes hex 16 through 1B of the TDE are updated accordingly. The task dispatcher is invoked; the task issuing the instruction is put into the wait state and another task is dispatched.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid descriptor
- Specification

RECC Example

Op		E	B_1	D_1
67		0	4	000

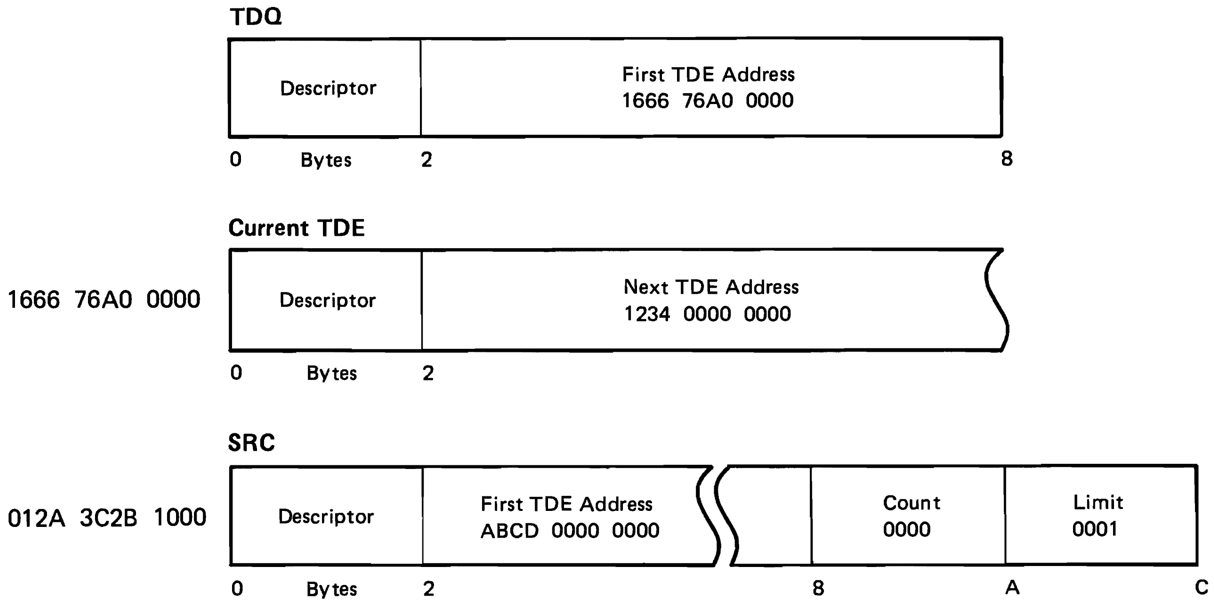
0 Bits 8 12 16 20 31

Assembler: RECC $D_1(B_1)$

Machine: 6700 4000

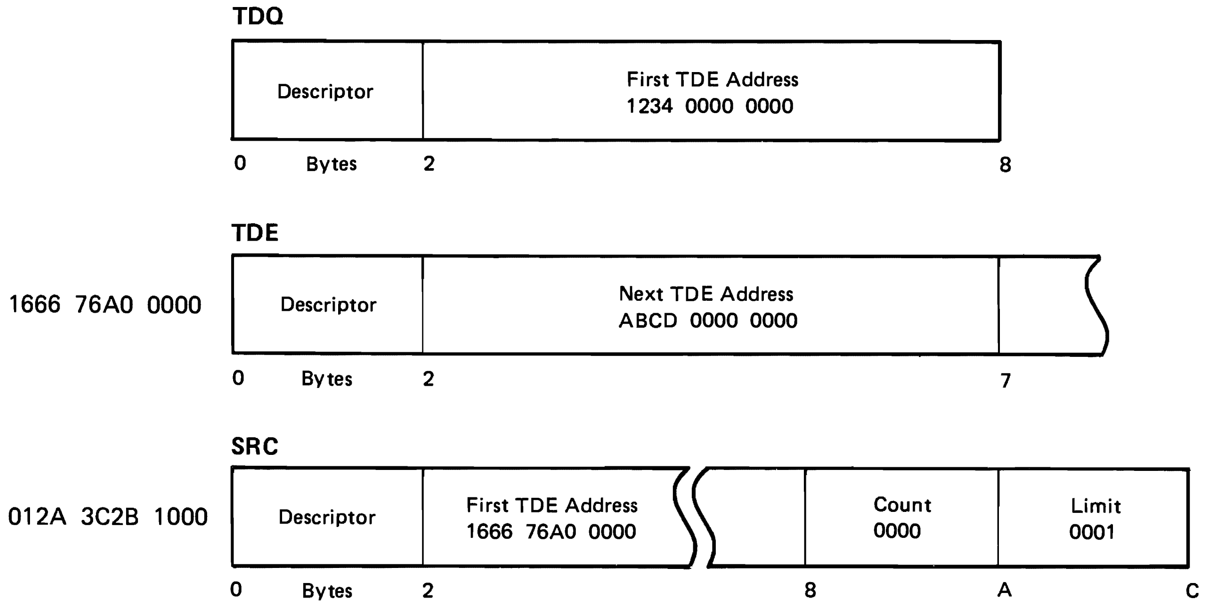
B_1(4): 012A 3C2B 1000

Before



The current TDE is dequeued from the TDQ and enqueued to the SRC.
The task issuing the RECC instruction is put in the wait state.
The task associated with the highest priority TDE on the TDQ is dispatched.

After



TDQ Byte 0, Bit 3	Before	After
Empty = 0	1	0
One or More = 1		

TDE Byte 0, Bit 4	Before	After
Last TDE = 0	0	0
Not Last TDE = 1		

RECEIVE MESSAGE (RECM)

Instruction Description

An SRM (send/receive message) is dequeued from the message list of the SRQ (send/receive queue) designated by the second operand, or the task is put into a wait state. The address of the SRM dequeued is loaded into B_1.

Format: SS

D9	B_1	I	B_2	D_2	B_3	D_3

0 Bits 8 12 16 20 32 36 47

Operation: The search type is specified by the I-field. If no message satisfies the search type, or if the message list is empty, B_1 is not altered. The messages searched are accessed sequentially, starting with the first message. The first message satisfying the search type is dequeued. The key is treated as unsigned binary data.

If no message satisfies the search type, or if the message list is empty, the instruction is nullified. The current TDE (task dispatching element) is dequeued from the TDQ (task dispatching queue) and is enqueued to the SRQ wait list in key (priority) sequence, bytes hex 16-1B of the TDE are updated accordingly, and the task dispatcher is invoked.

I-Field	Search Type
Bit 12	Message Key = Search Key (the third operand)
Bit 13	Message Key < Search Key (the third operand)
Bit 14	Message Key > Search Key (the third operand)
Bit 15	Not used

The search type is the logical OR of the I-bits specified. For a search type of binary 000x, no keys satisfy the search type; therefore a specification exception occurs.

Note: A receive first operation is accomplished by setting the I-field to binary 111x. In this case any search key provides the desired operation. However, because the third operand is accessed and used in the comparison, it is convenient to specify the third operand (the search key) as the header address to eliminate a potential address translation exception. A zero is forced for the length field in the header and the key value is ignored for specifications of I = binary 111x. Also, a check is not made for a page crossing in the key field if enqueue-first or enqueue-last is specified.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The search key specified by the third operand must be fullword aligned, is the length specified in the queue header, and cannot cross a page boundary.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRM descriptor
- Descriptor access: Monitored SRQ descriptor
- Descriptor access: Monitored TDE descriptor
- Effective address overflow
- Invalid descriptor
- Specification

RECM Example

Op	B_1	I	B_2	D_2	B_3	D_3
D9	4	8	5	000	6	000

0 Bits 8 12 16 20 32 36 47

2

Assembler: RECM B_1, $D_2(B_2)$, $B_3(D_3)$, I

Machine: D948 5000 6000

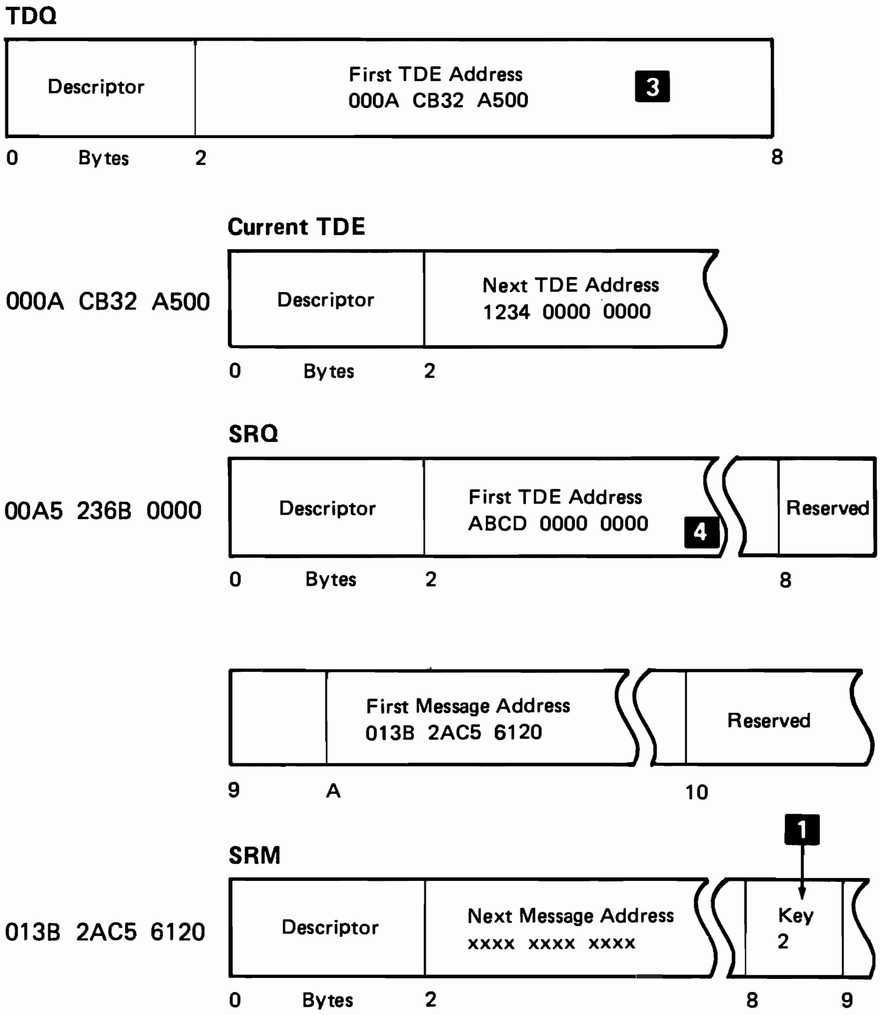
	Before	After
B_1(4):	011A 57CD 0200	011A 57CD 0200
B_2(5):	00A5 236B 0000	00A5 236B 0000
B_3(6):	00A5 23B0 0000	00A5 23B0 0000

Storage — Before and After

	0/8	2/A	4/C	6/E
00A5 23B0 0000	01			

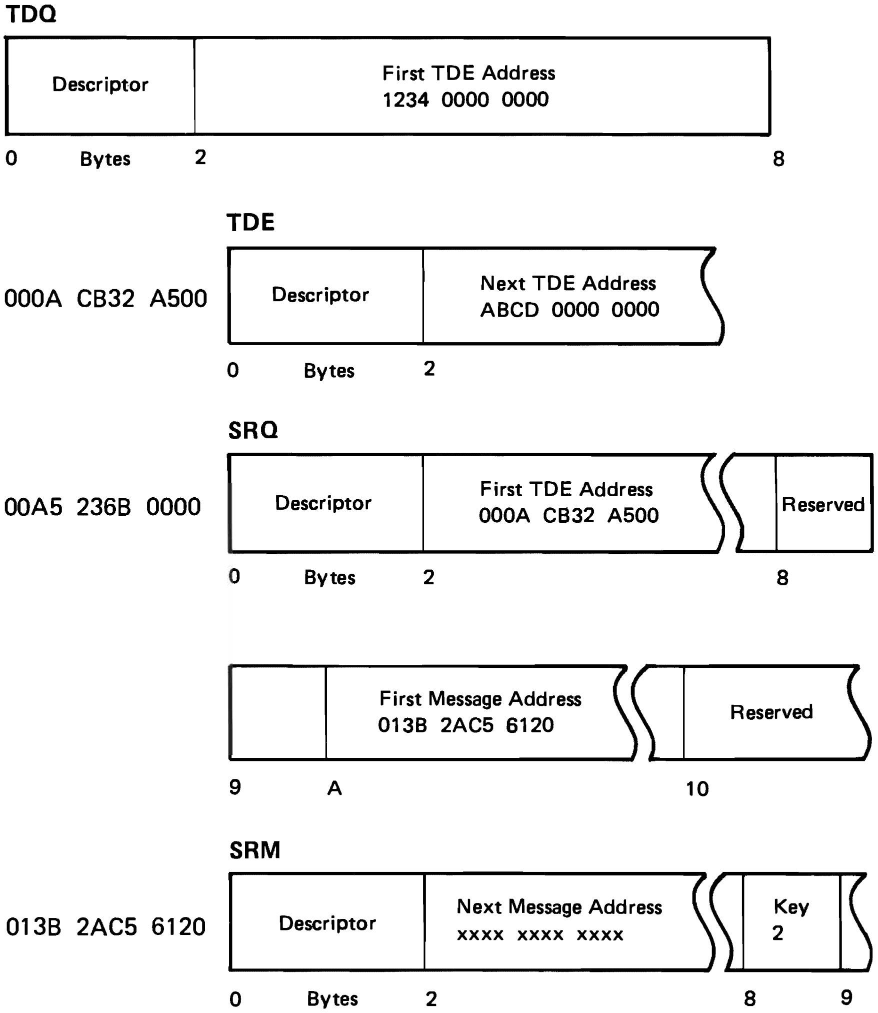
The message chain is searched for 1 a key value of 01. It is found that no message on the SRQ satisfies the equal 2 search type. The instruction is nullified and the TDE is dequeued from the TDQ 3 and enqueued to the SRQ list. The task issuing the RECM is put into the wait state, and the task dispatcher is invoked.

Before



TDQ Byte 0, Bit 3	Before	After
Empty = 0	1	1
One or More = 1		

After


TDQ
Descriptor
First TDE Address
1234 0000 0000
0
Bytes
2
8
TDE
000A CB32 A500
Descriptor
Next TDE Address
ABCD 0000 0000
0
Bytes
2
SRQ
00A5 236B 0000
Descriptor
First TDE Address
000A CB32 A500
Reserved
0
Bytes
2
8
First Message Address
013B 2AC5 6120
Reserved
9
A
10
SRM
013B 2AC5 6120
Descriptor
Next Message Address
xxxx xxxx xxxx
Key
2
0
Bytes
2
8
9

REMOVE PRIMARY DIRECTORY ENTRY (RPDE)

Instruction Description

The RPDE instruction is used to remove an entry from the primary directory. The primary directory entry is identified by the first operand.

Format: SI

83		4	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The first halfword of the first operand contains the hash table index; the high-order bit (bit 0) of the hash table entry index is not used to index the hash table. The second halfword of the first operand contains the PD index, with the high-order 4 bits being ignored and treated as zeros.

If the primary directory (PD) entry identified by the first operand is not on the PD chain, a specification exception is recognized and the operation is terminated. Otherwise, the SID and PID entries (bits 0-39) are updated. Bits 0-23 are forced to hex 00 0001; bits 24-39 are updated by shifting the PD index to the left one position and inserting a 0 into the vacated bit position, and bits 40-63 are forced to zeros. The entry is then removed from the PD chain.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand occupies a word in storage and must begin on a fullword boundary; otherwise, a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: This instruction should be used only to remove an invalid PD entry.

RPDE Example

Op 83		E 4	B_1 4	D_1 074

0 Bits 8 12 16 20 31

Assembler: RPDE $D_1(B_1)$

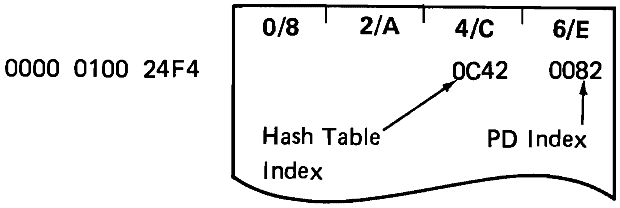
Machine: 8304 4074

B_1(4): 0000 0100 2480

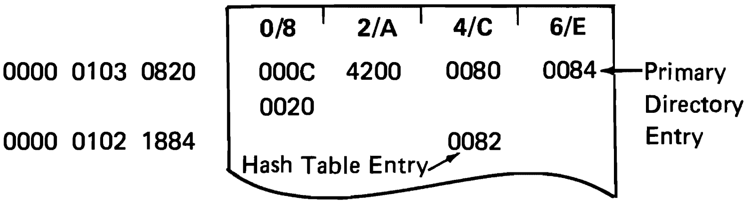
Primary Directory: 0000 0103 0000

Hash Table: 0000 0102 0000

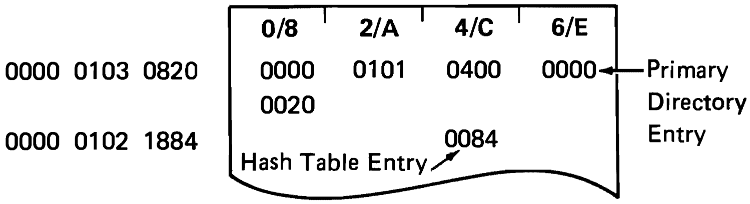
Storage — Before and After



Storage — Before



Storage — After



RESET ADDRESS COMPARE MODE (RACM)

Instruction Description

This instruction resets the soft address compare mode previously set by the Set Address Compare Mode instruction.

Format: RR

0D		3

0 Bits 8 12 15

Operation:

This instruction resets the address compare enable and store-only latches in the virtual address translator, the soft address compare flag (byte hex 24, bit 2 of the LSR [local storage register]), the microprocessor exception flags (byte hex 27, bits 2 and 3 of the LSR), and the virtual address-not-mapped flag (byte hex 22, bit 0 of the LSR).

Overflow and Sign Code: Not applicable.

Condition Code:

0	Address compare mode reset or no previous Set Address Compare Mode instruction executed.
1	Address compare mode not reset.
2	--
3	--

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

Programming Note: This instruction will not reset an address compare that has been entered through the console.

RACM Example

Op 0D		E 3

0 Bits 8 12 15

Assembler: RACM

Machine: 0D03

LSR Byte Hex	Bit	Description	Action
22	0	FLG0—VA Not Mapped	Reset
24	2	FLG2—Soft Address Compare	Reset
27	2, 3	UEX1—Address Compare Status	Reset

RESET CHAIN BUSY (RCB)

Instruction Description

This instruction is used to reset the busy bit of the first hold record on the object hold chain for the first operand.

Format: RR

0D	R_1	6

0 Bits 8 12 15

Operation: A 4096 byte hold hash table, whose address is given in bytes hex 8A-8F of the control address table, is accessed. This hold hash table address is initialized by the IMPL to point to the first byte in the table.

The first operand register contains an object address. This 6-byte effective address is hashed to create a 2-byte index into a hash table. If the chain is empty (contains no hold records), the hash table entry is all zeros, a specification exception is recognized, and the operation is suppressed.

If the hash table entry for the second operand object contains a nonzero value, that value is used as an index to access the first hold record in the chain. The 2-byte hash table entry is multiplied by 16 and concatenated to the right of the high-order 28 bits of the available hold record address; found in the control address table. The available hold record contents point to the start of the hold record area. The busy flag (byte 0, bit 5 of the hold record) is checked, and if it is a 0, a specification exception occurs, and the operation is suppressed. If the busy flag is a 1, it is reset and the operation is completed.

Overflow and Sign Code: Not applicable.

Condition Code: Unchanged.

Carry and Boundary Requirements: Not applicable.

Program Exceptions:

- Address translation
- Addressing
- Specification

RCB Example

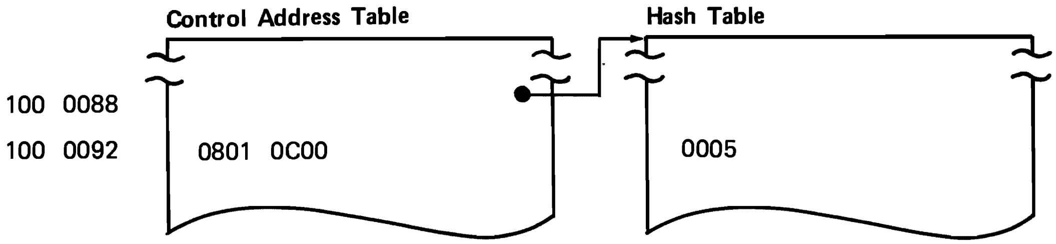
Op	B_1	E
0D	3	6

0 Bits 8 12 15

Assembler: RCB B_1

Machine: 0D36

B_1 (B): 0801 1803 0000



Hold Records — Before

	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	02	02	0801 1802 0000	0000	0000	0000	00	00
40								
50	04	02	0801 1803 0000	0000	0003	0000	00	00

Hold Records — After

	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	02	02	0801 1802 0000	0000	0000	0000	00	00
40								
50	00	02	0801 1803 0000	0000	0003	0000	00	00

RESET MACHINE CHECK MODE (RMCM)

Instruction Description

This instruction is used to reset the processor machine check mode after a machine check has been reported.

Format: RR

0D		2

0 Bits 8 12 15

Operation: This instruction signals the HMC that the reported error has been processed and another machine check can be logged into the MCLB (machine check log buffer).

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

RMCM Example

Op		E
0D		2

0 Bits 8 12 15

Assembler: RMCM

Machine: 0D02

RETURN AVAILABLE HOLD RECORD (RAHR)

Instruction Description

This instruction is used to return the hold record specified to the available hold record chain.

Format: RR

OD	R_1	5

0 Bits 8 12 15

Instruction Description: The first operand is a halfword index that points to a hold record. This hold record is returned to the available hold record chain. The first operand contents are not changed.

The address of the available hold record chain (bytes hex 92-98 of the control address table) is converted to a hold record index (bits 28-43 of the 6-byte virtual address) and the resultant 2-byte value is loaded into the object chain field (bytes A and B) of the hold record to be returned. The flag byte (0), the TDE identifier field (bytes 8 and 9), and the second chain pointer and cumulative hold field (bytes C-F) of the hold record are set to zero. The address of the hold record is loaded into the available hold record chain entry of the control address table and the operation is completed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry and Boundary Requirements: Not applicable.

Program Exceptions:

- Address Translation
- Addressing

RAHR Example

Op	R_1	E
0D	3	5

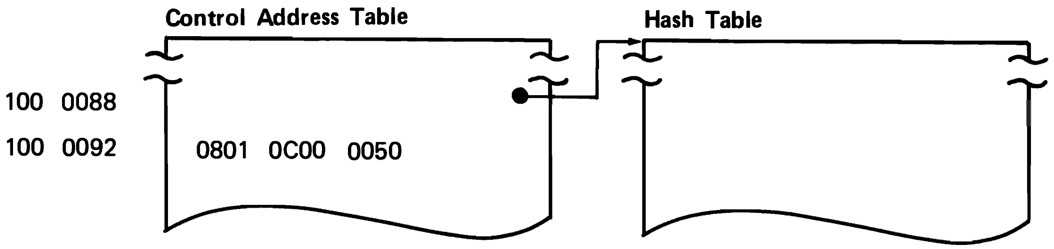
0 Bits 8 12 15

Assembler: RAHR R_1

Machine: 0D35

R_1 (3): 0003

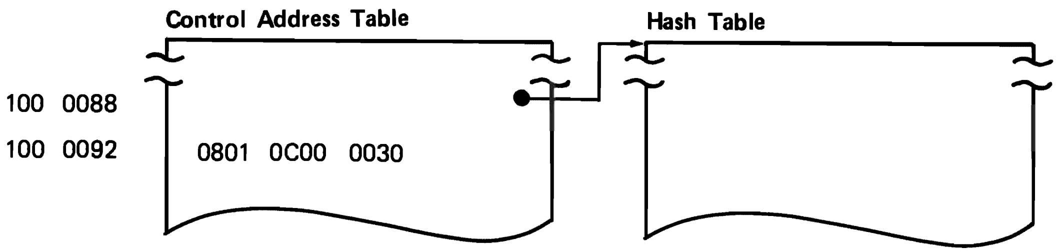
Storage – Before



Hold Records

	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	80	02	0000 0000 0000	0101	0002	0001	02	00
40								
50	00	00	0000 0000 0000	0000	0006	0000	00	00

Storage – After



Hold Records

	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
10								
20								
30	00	02	0000 0000 0000	0000	0005	0000	00	00
40								
50	00	00	0000 0000 0000	0000	0006	0000	00	00

SCAN (SCAN)

Instruction Description

The character string addressed by operand 5 is scanned left to right in order to search for the character value specified in operand 1.

Format: SS

CF	M_3	M_4	B_1	D_1	D_2

0 Bits 8 12 16 20 32 47

Operation: The controls operand (operand 1) specifies the starting address of a doubleword aligned, 8-byte string; a specification exception is signaled for improper alignment. The information contained in the controls operand is used to control the scan operation. The controls operand has the following format:

Bytes	Meaning
0	Mode control
1	Reserved
2, 3	Scan character
4	Reserved
5-7	String end

The *mode control* byte specifies the mode (simple or extended) for the base string character and for the scan character. When a single mode scan is requested in the options mask (operand 3), the base string character and the scan character must be specified as having the same mode. A specification exception occurs if a mixture of modes is specified for a single mode scan. When a mixed mode scan is requested in the options mask, the base string mode bit is used as both input to and output from the instruction. The mode control byte has the following format:

Bits	Value	Meaning
0	0	Base string character is in simple mode.
	1	Base string character is in extended mode.
1	0	Scan character is in a simple mode.
	1	Scan character is in extended mode.
2-7	000000	Reserved (must be 0). A specification exception occurs if any bits are not 0.

The *scan character* is either a 1- or a 2-byte value depending on the mode as specified in bit 1 of byte 0. A scan character in simple mode is specified as a 1-byte value in byte 3 of the controls operand (byte 2 is ignored). A scan character in extended mode is specified as a 2-byte value (bytes 2 and 3).

The string end is a 3-byte value specifying the segment group offset to the last byte of the string to be scanned. This 3-byte value, when concatenated on the right of the 3-byte segment group identifier specified in bytes 0-2 of base register hex D (operand 5), forms the 6-byte address of the rightmost byte of the string. If the base string address specified in operand 5 (base register D) points beyond the address of the rightmost byte of the string, a not found condition code is set, and the instruction is terminated.

Operand 2 is a displacement value and when combined with the value in base register 0 specifies an address for a branch. The branch is taken under control of the options mask if an escape code (any value less than hex 40 with the exceptions listed in the verification step) is encountered in the base string during the scan operation.

Operand 3 is an options mask that provides additional controls over the scan operation. The options mask has the following format:

Bits	Value	Meaning
8	0	Branch on encountering an escape code.
	1	Do not branch on encountering an escape code.
9	0	Mixed mode scan.[1]
	1	Single mode scan.[2]
10-11		Reserved.

[1] The hex 0E and hex 0F shift mode characters are recognized when mixed mode scan is specified.

[2] The hex 0E and hex 0F shift mode characters are not recognized when single mode scan is specified.

Operand 4 (bits 12, 13, and 14) is used as a mask to determine when the scan operation is complete. The bits correspond to condition codes 0, 1, and 2 respectively. Bit 15 of operand 4 is reserved.

The scan operation ends when one of the following conditions occur:

- Bit 12 is on, and the scan character compares equal to the base string character.

- Bit 13 is on, and the scan character compares less than the base string character.

- Bit 14 is on, and the scan character compares greater than the base string character.

- The last byte of the base string has been processed, and one of the previous conditions has not occurred. In this case, a not found condition code is set.

- A compare mask of all zeros results in completion of the instruction with a condition code of *not found*.

Operand 5 (base register hex D) specifies the address of the leftmost byte of the string to be scanned. When the instruction is interrupted or completed, this operand contains the address of the last character in the string that was scanned.

The scan operation consists of three possible steps: verification, comparison, and increment.

Verification Step: The verification step determines whether the base string value is a mode shift character or an escape character.

One of the following actions occurs if the base string charcter has a value less than hex 40.

- For mixed mode scan (bit 9 of operand 3 is off), if the base string byte contains a hex 0E value (shift out of simple mode) and if the base string is in simple character mode (bit 0 of the mode control byte is off), the mode of the base string is changed from simple to extended character mode. The scan operation bypasses the comparison step and continues with the increment step.

- For mixed mode scan (bit 9 of operand 3 is off), if the base string byte contains a hex 0F value (shift into simple mode), if the base string is in exteneded character mode (bit 0 of the mode control byte is on), and if this byte is the first byte of the extended character code, the mode of the base string is changed from extended to simple character mode. The scan operation bypasses the comparison step and continues with the increment step.

- If the base string character value is less than hex 40 and does not result in a mode shift character as previously described, and if the escape option has not been specified (bit 8 of operand 3 is on), the scan operation continues with the comparision step.

- If the base string character value is less than hex 40 and is not a mode shift as previously described, and if the escape option has been specified (bit 8 of the instruction is off), then an escape code has been encountered. The updated instruction address is replaced by the branch address, and a not found condition code is set. The branch address is the sum of the displacement (D2) from the instruction and the offset portion of the instruction stream base address contained in base register 0.

  Escape codes are detected under the following conditions:
  - If bit 9 of the instruction is on (single mode scan), a byte of the character being processed (both bytes are verified in extended mode) contains a value less than hex 40.
  - If bit 9 of the instruciton is off (mixed mode scan), a byte of the character being processed (both bytes are verified in extended mode) contains a value less than hex 40, but it is not a valid mode shift value.

Comparison Step: The scan operation proceeds by performing the appropriate comparison (simple or extended character mode) of the scan character to the base string character. The compare operation is performed with both the scan character and the base string character treated as unsigned quantities.

The mode of the scan character must be the same as the mode of the base string character; otherwise, no compare operation occurs, and the scan operation continues with the increment step.

If the mode of the scan character is the same as the mode of the base string character, a comparison is performed as follows:

- In simple character mode, the scan character (byte 3 of the controls operand) is compared with the string character currently addressed by operand 5.

- In extended character mode, the scan character (bytes 2 and 3 of the controls operand) is compared to the base string character. In extended character mode, the base string character consists of 2 bytes from the base string. If the rightmost byte of the 2-byte base string character requires a storage access beyond the last byte of the string, the Scan instruction is completed with a not found condition code. If this condition occurs, operand 5 addresses the leftmost byte of the 2-byte base string character which is the last byte of the base string.

If the result of the compare operation corresponds to a condition specified by the mask field (operand 4), the condition code is set and the Scan instruction is completed.

If the result of the compare operation does not correspond to a condition specified by the mask field, the scan operation continues with the increment step.

Increment Step: The purpose of the increment step is to alter operand 5 (base register hex D) so that it addresses the next base string character to be scanned. However, depending on certain conditions, operand 5 may or may not be altered.

Operand 5 is not altered and the Scan instruction is completed with a not found condition code when one of the following conditions exist:

- The segment group offset value in bytes 3, 4, and 5 of operand 5 is equal to the base string end value in bytes 5, 6, and 7 of operand 1.

- The segment group offset in bytes 3, 4, and 5 of operand 5 is 1 less than the base string end value in bytes 5, 6, and 7 of operand 1; a mode shift was not encountered; and the base string is being processed in extended mode.

Operand 5 is altered and the Scan instruction continues with the verification step when one of the following conditions exist:

- For a mixed mode scan, operand 5 is altered by 1 if a mode shift was encountered.

- Operand 5 is altered by 1 if the base string is in simple character mode.

- Operand 5 is altered by 2 if the base string is in extended character mode.

The scan operation continues with the verification step. The Scan instruction can be interrupted at this point, except immediately following a shift in or shift out character. In this case, the operation is interruptible at the next character.

The following defines the conditions that can be encountered at the end of the string and the addressability of base register hex D for each case.

Ending Condition	Addressability–Register Hex D Points To	Response
Simple character or mode shift		
• Shift into simple character mode	Character being compared	Mode shift; condition code *Not found*
• Shift out of simple character mode	Character being compared	Mode shift; condition code *Not found*
• Simple character	Character being compared	Condition code: *Not found* (unless compare mask satisfied)
• Escape code in simple character	First byte of character containing escape code	Branch taken
Extended character split across string end • Extended character	First byte of character	Condition code: *Not found*
• Escape code in extended character	First byte of character containing escape code	Branch taken
Extended character at string end • Extended character	First byte of character	Condition code *Not found* (unless compare mask satisfied)
• Escape code in extended character	First byte of character	Branch taken

Overflow and Sign Code: Not applicable.

Condition Code:

0 Scan character = character in string

1 Scan character < character in string

2 Scan character > character in string

3 Not found

Carry: Not applicable.

Boundary Requirements: The controls operand (operand 4) is a doubleword aligned, 8-byte string. A specification exception occurs and the instruction is suppressed if the controls operand is improperly aligned.

Program Exceptions:

– Address translation
– Addressing
– Effective address overflow
– Specification

SCAN Example

Op	M_3	M_4	B_1	D_1	D_2
CF	8	A	0	8F8	0094

0 Bits 8 12 16 20 32 47

Assembler: SCAN $D_1(B_1)$, D_2, M_3, M_4 or SCAN S_1, D_2, M_3, M_4

Machine: CF8A 08F8 0094

M_3(8): 1000 (Binary) Mixed mode, no branch on escape character
M_4(A): 1010 (Binary) Looking for base string character which is less than or equal to SCAN character.
B_1(0): 0102 0101 0000 (Base register for control information)

Before: B(D): 0103 0101 2CDF (Base register for base string start)
After: B(D): 0103 0101 2CF6

Storage — Before

	0/8	2/A	4/C	6/E
0102 0101 08F0	xxxx	xxxx	xxxx	xxxx
0102 0101 08F8	4000	3F3D	0001	2DE3
	xxxx	xxxx	xxxx	xxxx
0103 0101 2CD0	xxxx	xxxx	xxxx	xxxx
0103 0101 2CD8	xxxx	xxxx	xxxx	xx51
0103 0101 2CF0	8CD3	470E	FF38	3F3D
	xxxx	xxxx	xxxx	xxxx

Storage — After

	0/8	2/A	4/C	6/E
0102 0101 08F0	xxxx	xxxx	xxxx	xxxx
0102 0101 08F8	C000	3F3D	0001	2DE3
	xxxx	xxxx	xxxx	xxxx

	Before	After
Condition Code	x	0

SEND COUNT (SENDC)

Instruction Description

The current value of the count field in the SRC (send/receive count) designated by the first operand is incremented by 1.

Format: SI

66	I_2	B_1	D_1

0 Bits 8 16 20 31

Operation: If the new count value is greater than or equal to the limit value of the SRC and the wait list is not empty, if byte 0, bit 7 of the SRC equals:

0 All TDEs (task dispatching elements) are dequeued/enqueued

1 Only the first TDE is dequeued/enqueued

Byte 0, bit 7 determines the TDEs on the SRC wait list that are dequeued and subsequently enqueued in priority sequence to the TDQ (task dispatching queue) specified by the TDE. TDE bytes hex 16-1B are updated accordingly. If a TDE is enqueued at a higher priority than the current task, and bit 15 of the instruction equals zero, a task switch will occur.

Execution of the SENDC instruction may be interrupted by I/O. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point the interrupt was granted.

I-Field Bits	**Description**
8-14	Not used
15	Task Switch Control: 0 Task dispatcher to be invoked after waiting TDEs are moved to the TDQ 1 Task dispatcher not to be invoked

Overflow: A counter overflow causes an SRC overflow exception.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Invalid descriptor
- Specification
- SRC overflow

SENDC Example

Op 66	I 00	B$_1$ 4	D$_1$ 000
0 Bits	8	16	20 31

Assembler: SENDC D$_1$(B$_1$), I

Machine: 6600 4000

B$_1$(4): 000A 1B2C 0000

The count 1 is incremented by one. The TDE on the SRC wait list is dequeued 2 and enqueued to the TDQ.

Before

TDQ

0E1A FA2E 3000	Descriptor	First TDE Address 0E1A F2AE 3100
	0 Bytes	2 8

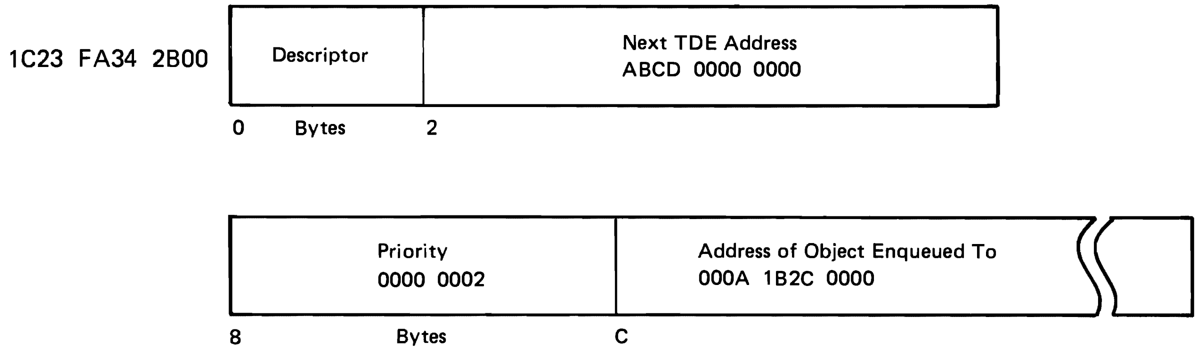
TDE

0E1A F2AE 3100	Descriptor	Next TDE Address 1234 0000 0000		Priority 0000 0001
	0 Bytes	2		8 C

SRC

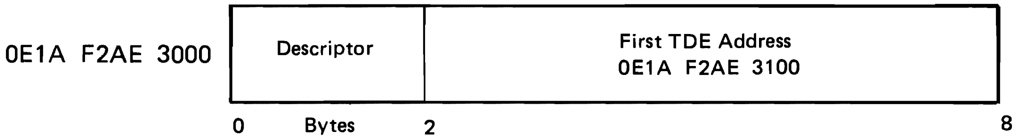
000A 1B2C 0000	Descriptor	First TDE Address 1C23 FA34 2B00		Count 0000	Limit 0001
	0 Bytes	2 2		8 1	A C

TDE

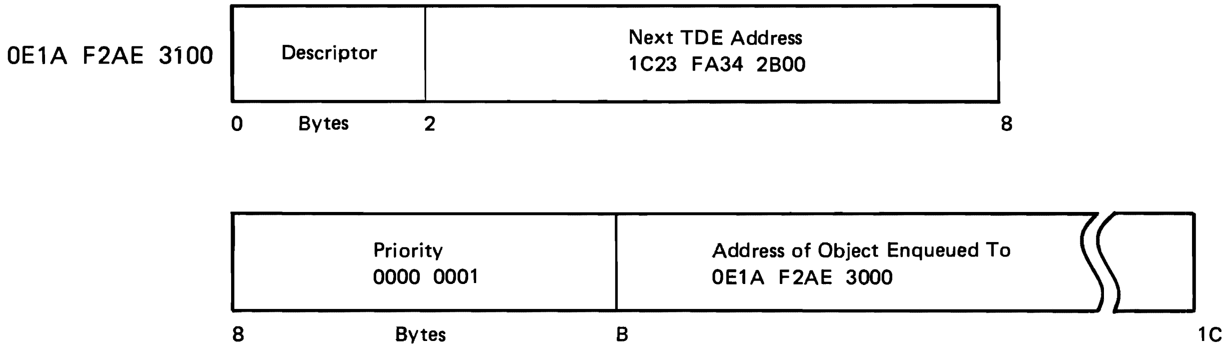


After

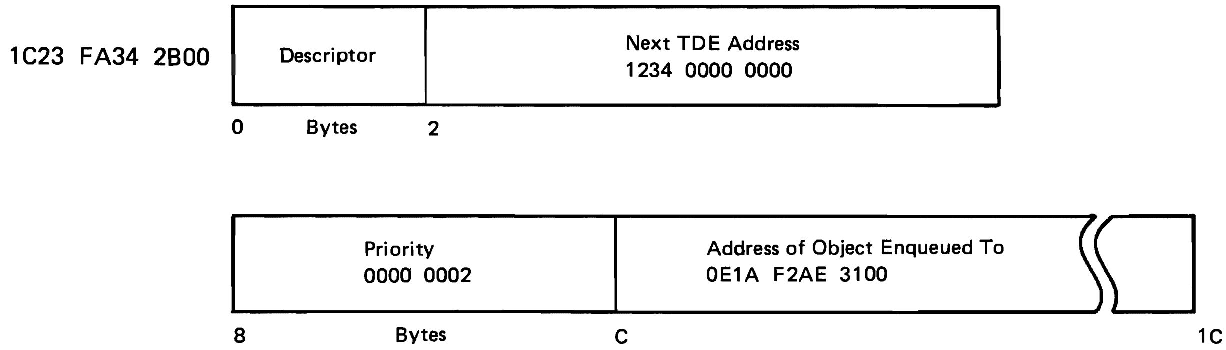
TDQ



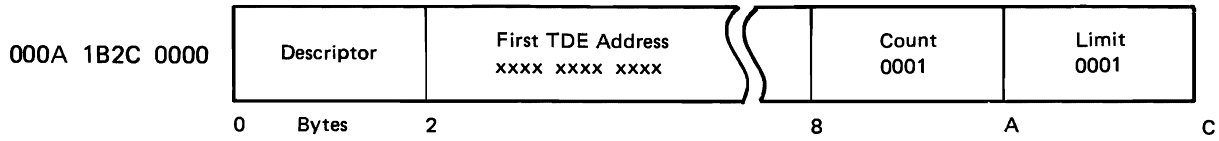
TDE



TDE



SRC



SEND MESSAGE (SENDM)

Instruction Description

The SRM (send/receive message) addressed by B_1 is enqueued to the message list of the SRQ (send/receive queue) designated by the second operand.

Format: RS

68	B_1	I	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The enqueuing method is designated by the I-field. The message list is searched, in sequence, beginning with the first message. The new message (the first operand) is enqueued above the first message that satisfies the search type. If the list is empty, the new message is enqueued first. If the search type is not satisfied, the new message is enqueued last. Search keys begin in byte 8 of the SRM, have a length specified in the queue header, and are treated as unsigned binary data.

The status of byte 0 bit 7 of the SRQ determines the TDEs (task dispatching elements) that are dequeued from the SRQ wait list and subsequently enqueued in priority sequence to the TDQ (task dispatching queue) specified by the TDE. If bit 7 is zero, all TDEs are moved. If bit 7 is one, only the top TDE is moved. TDE hex bytes 16-1B are updated accordingly. If a TDE is enqueued at a higher priority than the current task and if bit 15 of the instruction equals zero, a task switch will occur.

I-Field	**Search Type**
Bit 12	Search message key = the first operand message key
Bit 13	Search message key < the first operand message key
Bit 14	Search message key > the first operand message key
Bit 15	Task switch control: 0 Task dispatcher to be invoked after waiting TDEs are moved to the TDQ 1 Task dispatcher not to be invoked

The search type is the logical OR of the I-bits specified. Therefore, a specification of I = binary 000x results in enqueue last and I = binary 111x results in enqueue first.

Note: The key length specification in the queue header is key length minus 1. Therefore, if enqúeue first (binary 111x) or enqueue last (binary 000x) is specified, the key/text portion of the SRM must be at least 1 byte long. If enqueue-first or enqueue-last is specified, the key field is not checked for a page crossing.

Execution of the SENDM instruction may be interrupted by I/O. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point at which the interrupt was granted.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRM descriptor
- Descriptor access: Monitored SRQ descriptor
- Descriptor access: Monitored TDE descriptor
- Effective address overflow
- Invalid descriptor
- Specification

SENDM Example

Op	B_1	I	B_2	D_2
68	4	E	5	000

0 Bits 8 12 16 20 31

Assembler: SENDM $B_1, D_2(B_2)$, I

Machine: 684E 5000

B_1(4): 002B 3245 A120

B_2(5): 0001 FC30 0000

Before

TDQ

0E1A FA2E 3000

Descriptor	First TDE Address 0E1A F2AE 3100

0 Bytes 2 8

Current TDE

0E1A F2AE 3100

Descriptor	Next TDE Address xxxx xxxx xxxx

0 Bytes 2

Priority 0000 0009		Address of Object Enqueued To 0E1A FA2E 3000

8 Bytes C 16 1C

SRQ

0001 FC30 0000

Descriptor	First TDE Address 12A7 7EAC 0F00	Reserved	Key Length -1 = 0

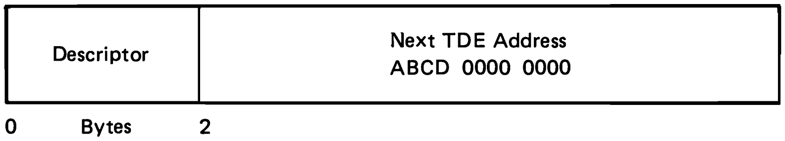
0 Bytes 2 8 9

First Message Address 21AB CB02 DA00	Reserved	

A Bytes 10 20

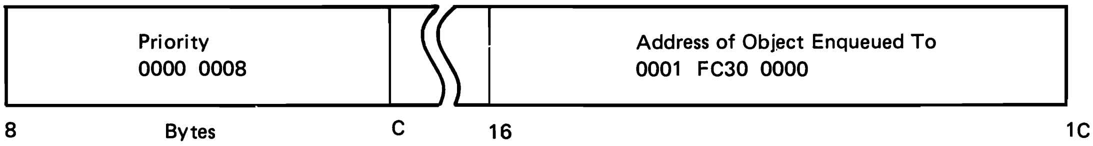
TDE

12A7 7EAC 0F00



Descriptor | Next TDE Address ABCD 0000 0000

0 Bytes 2

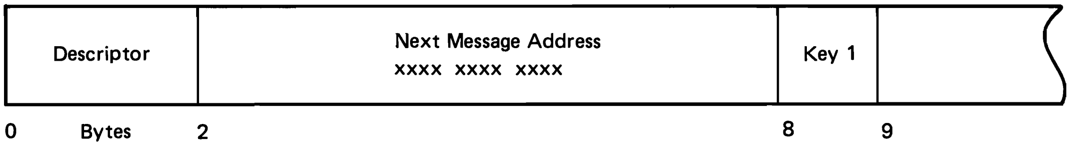


Priority 0000 0008 | Address of Object Enqueued To 0001 FC30 0000

8 Bytes C 16 1C

SRM

002B 3245 A120

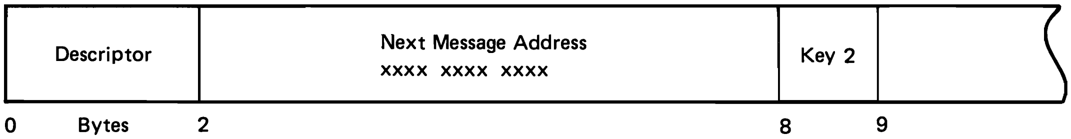


Descriptor | Next Message Address xxxx xxxx xxxx | Key 1

0 Bytes 2 8 9

SRM

21AB CB02 DA00

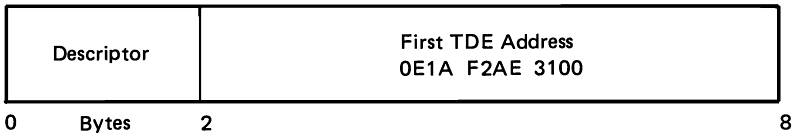


Descriptor | Next Message Address xxxx xxxx xxxx | Key 2

0 Bytes 2 8 9

After

TDQ

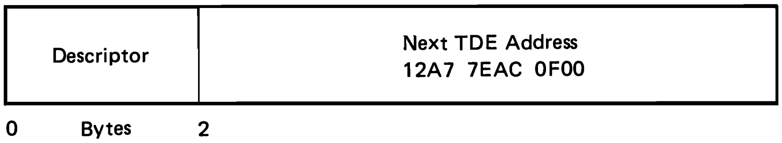


Descriptor | First TDE Address 0E1A F2AE 3100

0 Bytes 2 8

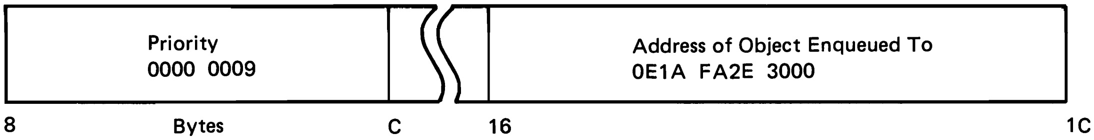
TDE

0E1A F2AE 3100



Descriptor | Next TDE Address 12A7 7EAC 0F00

0 Bytes 2



Priority 0000 0009 | Address of Object Enqueued To 0E1A FA2E 3000

8 Bytes C 16 1C

SRQ

0001 FC30 0000	Descriptor	First TDE Address xxxx xxxx xxxx	Reserved	Key Length-1
	0 Bytes	2	8	9

First Message Address 002B 3245 A120	
A	10 20

TDE

12A7 7EAC 0F00	Descriptor	Next TDE Address xxxx xxxx xxxx
	0 Bytes	2

Priority 0000 0008		Address of Object Enqueued To 0E1A F2AE 3100
8	C	16 20

SRM

002B 3245 A120	Descriptor 6C00	Next Message Address 21AB CB02 DA00	Key 1 0000	
	0 Bytes	2	8	9

SRM

21AB CB02 DA00	Descriptor	Next Message Address 6789 0000 0000	Key 2 0000	
	0 Bytes	2	8	9

Since the task switch control bit (bit 15 of the instruction) was zero, the task dispatcher is now invoked; and since the highest priority TDE on the TDQ is not the current TDE, the task dispatcher will switch out the old TDE and switch in the new TDE.

SEND MESSAGE AND WAIT (SENDMW)

Instruction Description

The SRM (send/receive message) addressed by B_1 is enqueued to the message list of the SRQ (send/receive queue) designated by the second operand.

Format: RS

69	B_1	I	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The enqueuing method is designated by the I-field. The message list is searched, in sequence, beginning with the first message. The new message (the first operand) is enqueued above the first message that satisfies the search type. If the list is empty, the new message is enqueued first. If the search type is not satisfied, the new message is enqueued last. Search keys begin in byte 8 of the SRM, have a length specified in the queue header, and are treated as unsigned binary data.

I-Field	**Search Type**
Bit 12	Search message key = the first operand message key
Bit 13	Search message key < the first operand message key
Bit 14	Search message key > the first operand message key
Bit 15	Not used

The search type is the logical OR of the I-bits specified. Therefore, a specification of I = binary 000x results in enqueue last and I = binary 111x results in enqueue first.

All TDEs (task dispatching elements) on the SRQ wait list are dequeued and subsequently enqueued in priority sequence to the TDQ (task dispatching queue) specified by the TDE. TDE hex bytes 16-1B are updated accordingly.

After all TDEs on the SRQ wait list are processed, the current TDE is dequeued from the prime TDQ and the task dispatcher is invoked. Bit 6 of the TDE descriptor is set on to indicate that the TDE is waiting for an SRM to be processed. Bit 6 of the SRM descriptor is set on to indicate that a TDE should be returned to the TDQ by the next OU task SENDM designating this SRM, instead of enqueuing the SRM to an SRQ.

When an HMC task sends an SRM with bit 6 of its descriptor on, the message is not enqueued to any queue. Instead, the 6-byte TDE address field starting at offset hex 7A from the beginning of the SRM is considered to be the address of a TDE. This TDE is enqueued to the TDQ specified by the TDE and bit 6 of the TDE descriptor and bit 6 of the SRM descriptor are reset. The TDE address field must be within the same page as the SRM descriptor byte or a machine check will occur.

The TDE address at offset hex 7A from the beginning of the SRM is not stored there by HMC, but is assumed to be there prior to the execution of the SENDMW instruction.

Note: The key length specification in the queue header is key length minus 1. Therefore, if enqueue first (binary 111x) or enqueue last (binary 000x) is specified, the key/text portion of the SRM must be at least 1 byte long. If enqueue-first or enqueue-last is specified, the key field is not checked for a page crossing.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy
- Descriptor access: Monitored SRM descriptor
- Descriptor access: Monitored SRQ descriptor
- Descriptor access: Monitored TDE descriptor
- Effective address overflow
- Invalid descriptor
- Specification

SENDMW Example

Op	B	I	B_2	D_2
69	4	E	5	000

0 Bits 8 12 16 20 31

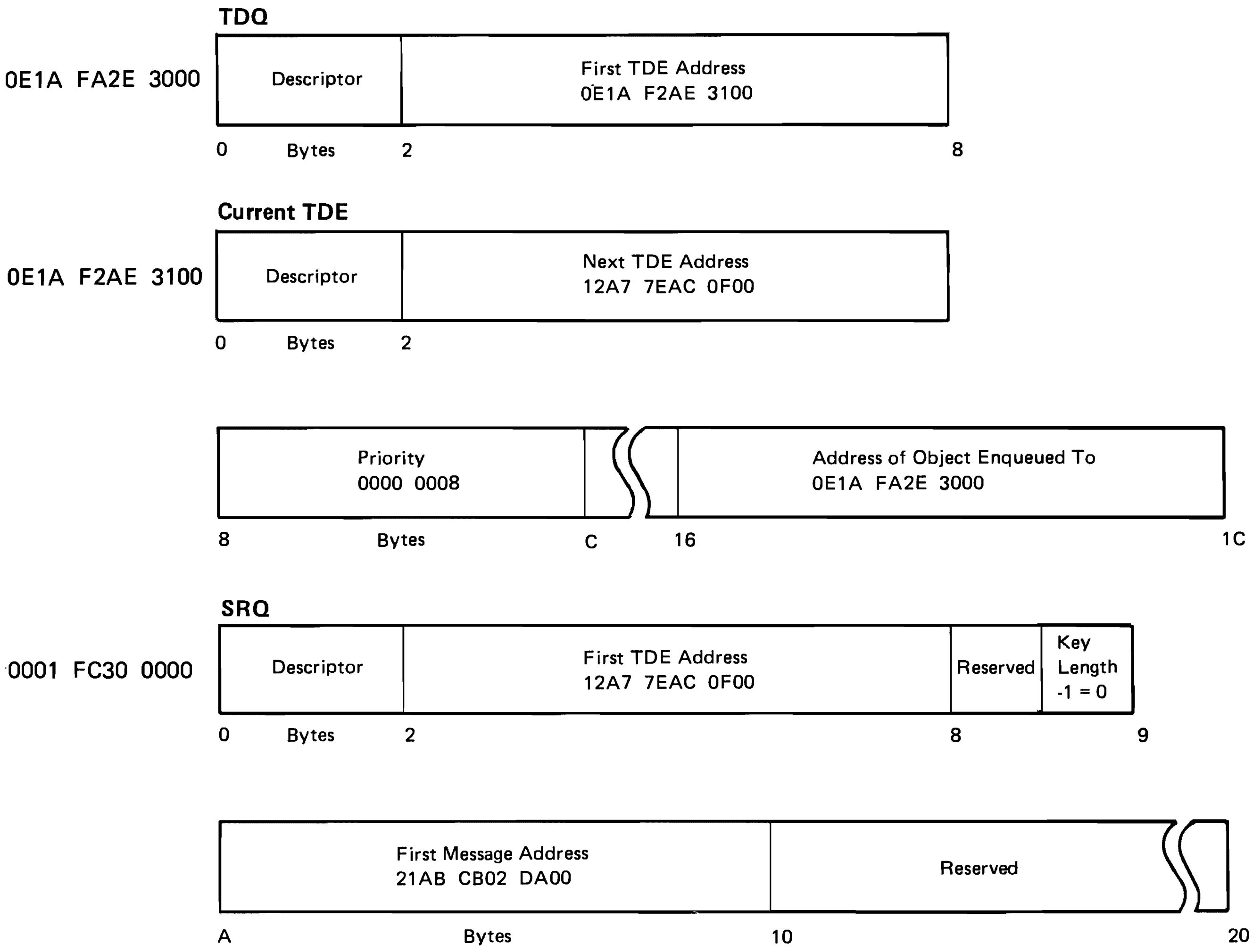
Assembler: SENDMW B_1, $D_2(B_2)$, I

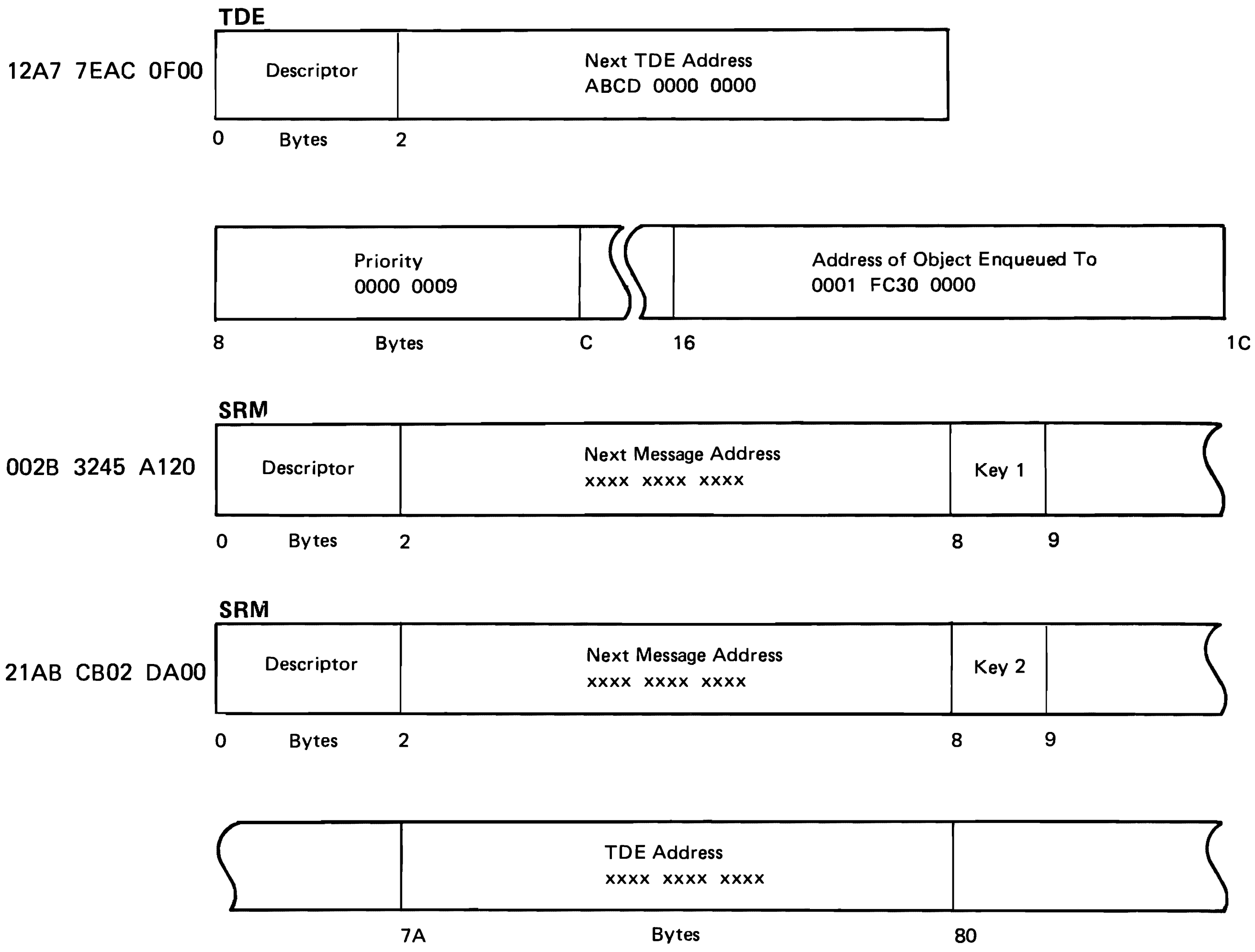
Machine: 694E 5000

B_1(4): 002B 3245 A120

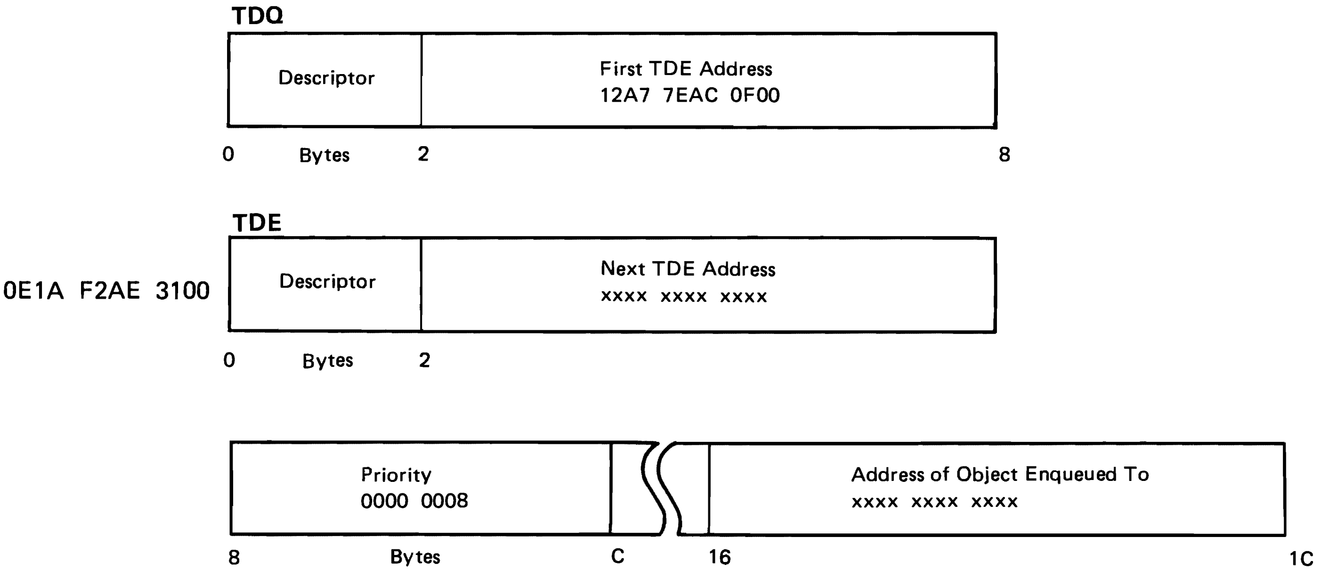
B_2(5): 0001 FC30 0000

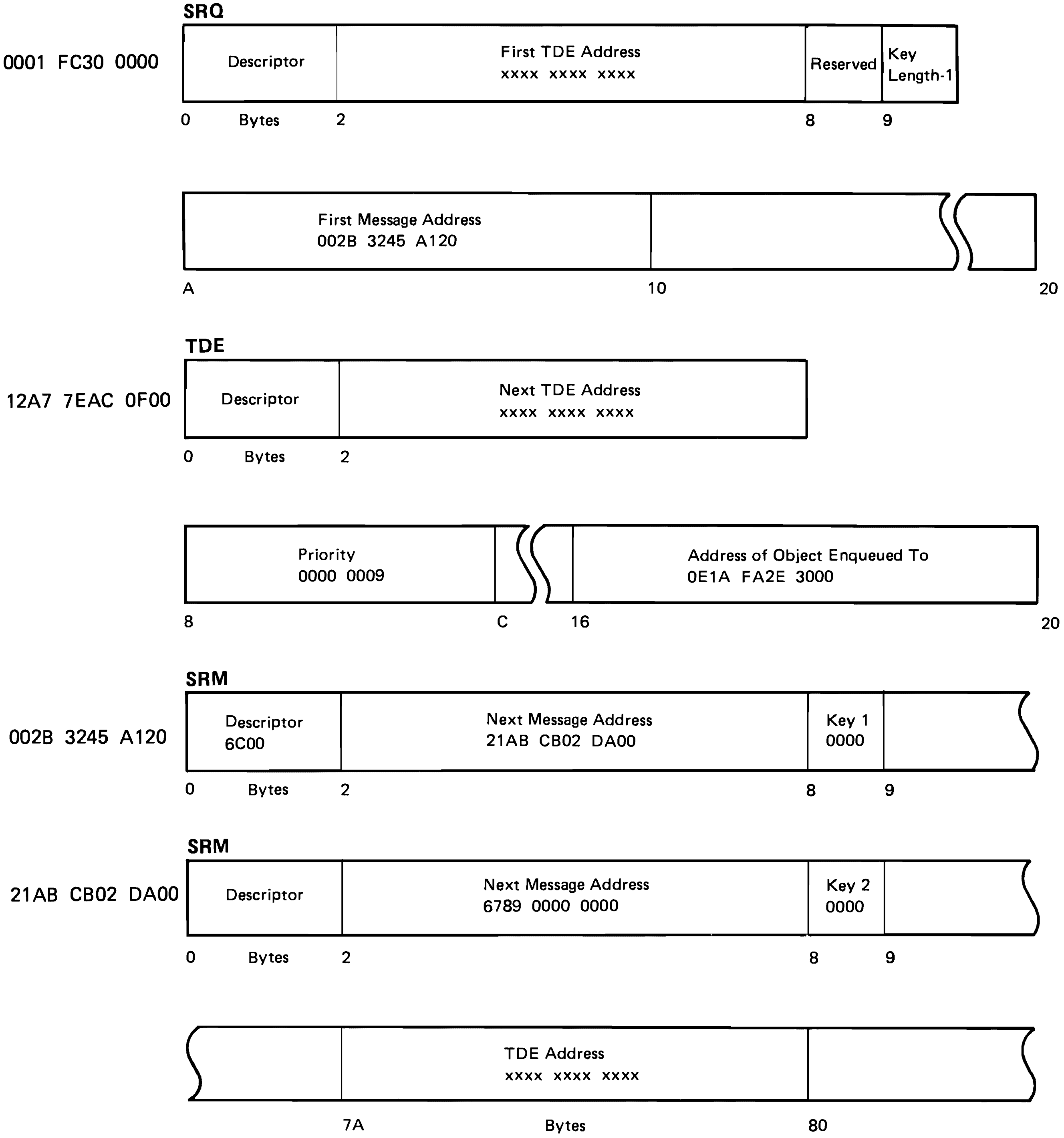
Before


TDQ
0E1A FA2E 3000
Descriptor
First TDE Address
0E1A F2AE 3100
0
Bytes
2
8
Current TDE
0E1A F2AE 3100
Descriptor
Next TDE Address
12A7 7EAC 0F00
0
Bytes
2
Priority
0000 0008
Address of Object Enqueued To
0E1A FA2E 3000
8
Bytes
C
16
1C
SRQ
0001 FC30 0000
Descriptor
First TDE Address
12A7 7EAC 0F00
Reserved
Key
Length
-1 = 0
0
Bytes
2
8
9
First Message Address
21AB CB02 DA00
Reserved
A
Bytes
10
20


TDE
12A7 7EAC 0F00
Descriptor
Next TDE Address
ABCD 0000 0000
0
Bytes
2
Priority
0000 0009
Address of Object Enqueued To
0001 FC30 0000
8
Bytes
C
16
1C
SRM
002B 3245 A120
Descriptor
Next Message Address
xxxx xxxx xxxx
Key 1
0
Bytes
2
8
9
SRM
21AB CB02 DA00
Descriptor
Next Message Address
xxxx xxxx xxxx
Key 2
0
Bytes
2
8
9
TDE Address
xxxx xxxx xxxx
7A
Bytes
80

After


TDQ
Descriptor
First TDE Address
12A7 7EAC 0F00
0
Bytes
2
8
TDE
0E1A F2AE 3100
Descriptor
Next TDE Address
xxxx xxxx xxxx
0
Bytes
2
Priority
0000 0008
Address of Object Enqueued To
xxxx xxxx xxxx
8
Bytes
C
16
1C



Since the task switch control bit (bit 15 of the instruction) was zero, the task dispatcher is now invoked; and since the highest priority TDE on the TDQ is not the current TDE, the task dispatcher will switch out the old TDE and switch in the new TDE.

SET ADDRESS COMPARE MODE (SACM)

Instruction Description

This instruction establishes address compare mode; the first operand contains the compare address and, optionally, a data byte to be used for compare on store. The address compare type is specified by the second operand.

Format: SI

4B	I_2	B_1	D_1
0 Bits	8	16	20 31

Operation: If bit 15 of the instruction (immediate field) equals 1, bits 0-7 of the first operand contain a character that is compared with the data stored into the location at the compare address.

The second operand specifies the address compare type:

Immediate Field Bits	Value	Description
12-13	00	Instruction stream
	01	Processor data
	10	I/O data
	11	Any
14	0	Fetch/store
	1	Store only
15	0	No compare on store
	1	Compare on store

Bits 12 and 13 specify whether the address compare exception is to be presented for one or any of the storage access types. Bit 14 specifies whether the exception is to be recognized for a fetch/store access or a store only access. If bit 14 is a 1, bits 12-13 are ignored and the exception is recognized for any of the store access types. Bit 15 specifies that bits 0-7 of the first operand are to be compared to the value stored at the compare address. If the values are equal, an address compare exception is recognized; otherwise normal operation continues. If bit 15 is a 1, bits 12-14 are ignored.

First Operand Bits	Description
0-7	Character
8-15	Unused
16-63	Compare Address

Overflow and Sign Code: Not applicable.

Condition Code:

0	Address compare mode set
1	Address compare mode not set
2	-
3	--

Carry: Not applicable.

Boundary Requirements: The operand address from the instruction identifies an 8-byte field in storage that must be fullword aligned; otherwise a specification exception is recognized and the operation is suppressed.

Program Exceptions:

- Address compare
- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Notes:

1. If an address stop has previously been set via the console, condition code 1 is set and the operation completes without disturbing the console stop. If a console stop is set after the SACM instruction was processed, the address stop set by the SACM instruction is overridden.
2. This instruction will not override a compare set through the console.
3. The exception produced by an address compare can be masked in a TDE. If this exception is masked and an address compare occurs due to a previous SCAM instruction, the exception mask is reversed by the HMC. Therefore the next time this address compare occurs, an address exception is presented.
4. The IMP exception handler must mask the address compare exception within the TDE whenever an address compare exception occurs. Unless the mask is set, attempting to execute the IMP instruction on which the address compare exception occurred results in another address compare exception.

SACM Example

Op	I_2	B_1	D_1
4B	08	3	024
0 Bits 8		16 20	31

Assembler: SACM $D_1(B_1), I_2$

Machine: 4B08 3024

$B_1(3)$: 0123 4567 8000

Storage – Before and After

	0/8	2/A	4/C	6/E
0123 4567 8024			0000	12C4
	1131	1522		

Address compare on I/O data fetch is set at address 12C4 1131 1522.

	Before	After
Condition Code:	x	0

SET CHAIN BUSY (SCB)

Instruction Description

The SCB instruction locates the chain of hold records for an object address and sets the busy flag in the chain so no other grant or free operations can be done on the chain.

The second-operand register contains an object address. The SCB instruction locates the hash table from the CAT (control address table), hashes the object address, selects the entry from the hash table, and loads the 6-byte address of the first hold record into the first-operand register. The busy flag in the first hold record is also set to 1.

Format: RR

36	B_1	B_2

0 Bits 8 12 15

Operation: The second-operand register contains an object address. A 4096-byte HHT (hold hash table) whose address is given in bytes hex 8A-8F of the CAT is accessed. This HHT address is initialized by an IMPL (initial microprogram load) to point to the first byte in the page.

This 6-byte effective address is hashed to create a 1-byte index into a hash table. The 2-byte hash table entry is used as a record index into the segments containing the hold chains. The selected 2-byte hash table entry (when multiplied by 16 and concatenated to the right of the high-order 28 bits of the AHR [available hold record] address found in bytes 92-95 of the CAT), addresses the first hold record in the chain for the second-operand object address and its hash synonyms. If the chain is empty (contains no hold records), the hash table entry is all zeros. In this case, the first-operand register contents are unchanged, and the condition code is set to 1.

If the hash table entry for the second-operand object contains a nonzero value, that value is used as a record index to access the first hold record in the chain. The 2-byte hash table entry is multipled by 16 and catenated to the right of the high-order 28 bits of the AHR (available hold record) address. These bits of the AHR point to the start of the hold record area. Bit 5 of the first byte of the hold record (the chain busy flag for example) is checked. If it is a 1 (indicating the chain is already busy), a descriptor access busy program exception is recognized and the operation is nullified. If the chain busy flag is a zero, it is set to one, the address of the first hold record in this chain is loaded into the first-operand register, and the condition code is set to zero.

Overflow and Sign Code: Not applicable.

Condition Code:

0 Chain was set busy
1 Chain was empty

Carry: Not applicable.

Boundary Requirements: The hold record must be quadword aligned; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Descriptor access: Busy

SCB Example

Op 36	B_1 D	B_2 D

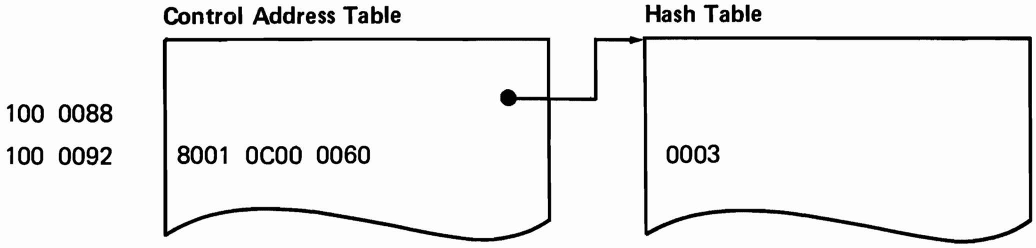
0 Bits 8 12 15

Assembler: SCB B_1, B_2

Machine: 36DD

	Before	After
B_1(D) and B_2(D):	8001 1800 0000	8001 0C00 0030

Base register hex D contains the address of the first hold record on the chain after the instruction has executed.



	Flags	Hold	Object Address	TDE	Primary Chain	Secondary Chain	Cumulative Hold Field	Unused
0801 0C00 0000								
0030	00	02	8001 1801 0000	0001	0004	0000	00	00
0040	00	84	8001 1802 0000	0001	0005	0000	00	00
0050	02	02	8001 1803 0000	0002	0000	0000	00	00
0060	00	00	0000 0000 0000	0000	0007	0000	00	00
0070								

The chain busy flag (hex 04) is set by this instruction.

SET CLOCK COMPARATOR (SETCC)

Instruction Description

The current value of the clock comparator is replaced by the first operand.

Format: SI

6D		2	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The only bits of the operand that are set in the clock comparator are those that correspond to the bit positions to be compared with the time-of-day clock. The remaining rightmost bits are ignored and are not preserved in the clock comparator. If no Set Time-Of-Day Clock instruction has been issued prior to the SETCC, the results of the compare are unpredictable.

The address of the SRC (send/receive counter) that indicates when the value of the time-of-day clock is equal to or greater than the value of the clock comparator is indicated in the control address table (Figure 2-2). No check for a counter limit of zero is made when SETCC is issued.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operand occupies 8 bytes in storage and must begin on a word boundary; otherwise a specification exception occurs, and the operation is suppressed.

The SRC associated with this instruction must begin on a word boundary and be storage resident; otherwise a machine check will occur when the send to the counter takes place.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Programming Note: If the value to be set into the clock comparator is less than the value of the time-of-day clock, the value is loaded in the clock comparator and the send count is issued.

SETCC Example

Op		E	B$_1$	D$_1$
6D		2	3	1A0

0 Bits 8 12 16 20 31

Assembler: SETCC D$_1$(B$_1$)

Machine: 6D02 31A0

B$_1$(3): 0000 A1B2 C000

Storage — Before and After

	0/8	2/A	4/C	6/E
0000 A1B2 C1A0	0000	0008	0400	00xx

Clock Comparator — Before

xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xx	xx	xxxx	xxxx	xxxx	xxxx	xxxx

0 Bits 42 56 64

Clock Comparator — After

0000	0000	0000	0000	0000	0000	0000	1000	0000	0100	00	xx	xxxx	xxxx	xxxx	xxxx	xxxx

0 Bits 42 56 64

SET INDICATOR (SETIND)

Instruction Description

An indicator byte in main storage is set according to the condition code and a mask.

Format: SI

5A	M_2	1	B_1	D_1

0 Bits 8 12 16 20 31

Operation: The mask, M_2, is compared to the condition code and, if a match is found, a hex F1 is stored at the first operand location. If no match is found, a hex F0 is stored at the first operand location.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

SETIND Example

Op	M_1	E	B_1	D_1
5A	9	1	3	035

0 Bits 8 12 16 20 31

Assembler: SETIND $D_1(B_1), M_2$

Machine: 5A91 3035

B_1(3): 029C 1A2C 2000

Storage — Before

	0/8	2/A	4/C	6/E
029C 1A2C 2035			xx	xxxx
	xx			

Storage — After

	0/8	2/A	4/C	6/E
029C 1A2C 2035			F1	xxxx
	xx			

SET INTERVAL TIMER (SETIT)

Instruction Description

The current value in one of the interval timers is replaced by the first operand.

Format: SI

6A	I_2	B_1	D_1
0 Bits	8	16	20 31

Operation: Two interval timers are provided. The first interval timer is called the task interval timer.

Only those bits of the first operand that correspond to the bit positions to be updated are set in the timer. The remaining rightmost bits are ignored and are not preserved in the timer.

When the second interval timer is specified, the SRC (send/receive counter), which is used to indicate when the value in the interval timer has been decremented to zero, is specified in the control address table (Figure 2-2).

When the task interval timer is specified, a dispatcher timer exception is generated to indicate when the value in the task interval timer has been decremented through zero. If an untimed task issues a Set Interval Timer instruction to the task interval timer, a specification exception occurs and the operation is suppressed. Also, if a timed task issued a SETIT instruction to the task interval timer when task dispatching is disabled, the new value is loaded but the task interval timer is not started.

The selection of the particular interval timer to be loaded with the first-operand interval and the technique for handling the interval timer is specified in the I-field. I-field values of hex 3-F cause a specification exception and the operation is suppressed.

I-Field	**Timer Control**
Hex 00	First interval timer (also used as task interval timer)
Hex 01	Second interval timer, single interval
Hex 02	Second interval timer, repeat interval
Hex 03–FF	Invalid

If repetitive timing is specified, the interval timer function is reinitiated using the value in the repetitive interval timer doubleword when the prior interval expires and the SRC specified in the control address table is used. The repetitive interval timer doubleword is not changed by this instruction.

Overflow: No overflow is indicated if the SRC increment causes a carry.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: See *Overflow.*

Boundary Requirements: The first operand occupies 8 bytes in storage and must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

The SRC associated with this instruction must begin on a word boundary and be storage resident; otherwise a machine check will occur when the send to the counter takes place.

A machine check occurs if the repetitive interval timer doubleword does not begin on a doubleword boundary and is not resident.

This page is intentionally left blank.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

SETIT Example

Op	I_2	B_1	D_1
6A	00	3	000

0 Bits 8 16 20 31

Assembler: SETIT $D_1(B_1)$, I_2

Machine: 6A00 3000

$B_1(3)$: 0000 1A2C 2000

Storage — Before and After

	0/8	2/A	4/C	6/E
0000 1A2C 2000	0000	0008	C480	0000

Task Interval Timer — Before

xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xx	xx	xxxx	xxxx	xxxx	xxxx	xxxx

0 Bits 42 56 64

Task Interval Timer — After

0000	0000	0000	0000	0000	0000	0000	1000	1100	0100	10	xx	xxxx	xxxx	xxxx	xxxx	xxxx

0 Bits 42 56 64

SET TIME-OF-DAY CLOCK (SETTOD)

Instruction Description

The current value of the time-of-day clock is replaced by the first operand.

Format: SI

6D		4	B_1	D_1
0 Bits	8	12	16	20 31

Operation: The operand is considered to be an unsigned, 64-bit, binary number. Only bits of the operand that correspond to the bit positions to be updated are set in the time-of-day clock. The remaining rightmost bits are ignored and not saved in the time-of-day clock. If timing functions are still active due to a prior SETCC (set clock comparator) instruction, the SETCC is canceled and no send count is performed.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operand occupies 8 bytes in storage and must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

SETTOD Example

Op		E	B_1	D_1
6D		4	3	000

0 Bits 8 12 16 20 31

Assembler: SETTOD $D_1(B_1)$

Machine: 6D04 3000

B_1(3): 0000 A415 3000

Storage — Before and After

	0/8	2/A	4/C	6/E
0000 A415 3000	00A4	000E	2100	0000

Time of Day Clock — Before

xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xx	xx	xxxx	xxxx	xxxx	xxxx	xxxx

0 Bits 42 56 64

Time of Day Clock — After

0000	0000	1010	0100	0000	0000	0000	1110	0010	0001	00	xx	xxxx	xxxx	xxxx	0000	0000

0 Bits 42 56 64

SHIFT LEFT ARITHMETIC (SLA)

Instruction Description

The integer part of the first operand is shifted left the number of bits specified by the I_2 field.

Format: SS

03	R_1	I_2

0 Bits 8 12 15

Operation: The value in I_2 is 1 less than the number of bits to be shifted. All 15 integer bits of the first operand participate in the left shift, and zeros are supplied to the vacated rightmost register positions.

Overflow: If a bit unlike the sign bit is shifted out of position 1, a binary overflow exception occurs.

Sign Code: The sign of the first operand remains unchanged.

Condition Code:

0	Result = 0
1	Result < 0
2	Result > 0
3	--

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions: Binary overflow

SLA Example

Op	R_1	I_2
03	A	2

0 Bits 8 12 15

Assembler: SLA R_1, I_2

Machine: 03A2

	Before	After
R_1(A):	0A	50
Condition Code:	x	2

SHIFT LEFT HALFWORD AND COUNT (SLHCT)

Instruction Description

The second operand is shifted left until a 1 bit is shifted out of the leftmost bit position. A value equal to the number of bits shifted out is placed in the byte register specified by r_1.

Format: RS

	61	r_1	0	B_2	D_2
0 Bits	8	12	16	20	31

Operation: The second operand occupies a halfword in storage. All 16 bits of the second operand participate in the shift left, and zeros are supplied to the vacated rightmost bit positions.

If the second operand contains no 1 bits, no shift occurs, and the value zero is placed in the byte register specified by r_1.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The halfword storage operand must start on a halfword boundary; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

SLHCT Example

Op	r_1	E	B_2	D_2
61	C	0	3	120

0 Bits 8 12 16 20 31

Assembler: SLHCT r_1, $D_2(B_2)$

Machine: 61C0 3120

B_2(3): 0014 6A3B F000

	Before	After
r_1(C):	xx	09

Storage — Before

	0/8	2/A	4/C	6/E
0014 6A3B F120	00AC			

Storage — After

	0/8	2/A	4/C	6/E
0014 6A3B F120	5800			

SHIFT LEFT LOGICAL (SLL)

Instruction Description

The first operand is shifted left the number of bits specified by I_2.

Format: RR

01	R_1	I_2

0 Bits 8 12 15

Operation: The value contained in the I_2 field is 1 less than the number of bits to be shifted.

All 16 bits of the first operand participate in the shift left, and zeros are supplied to the vacated rightmost register positions. Bits shifted out of the register are lost.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

SLL Example

Op	R_1	I_2
01	6	2

0 Bits 8 12 15

Assembler: SLL R_1, I_2

Machine: 0162

	Before	After
R_1(6):	C4BE	25F0

SHIFT RIGHT ARITHMETIC (SRA)

Instruction Description

The integer part of the first operand is shifted right the number of bits specified by I_2.

Format: RR

04	R_1	I_2

0 Bits 8 12 15

Operation: The value contained in the I_2 field is 1 less than the number of bits to be shifted.

All 15 integer bits of the first operand participate in the shift right, and bits equal to the sign are supplied to the vacated bit positions. Bits shifted out are lost.

Overflow: Not applicable.

Sign Code: The sign of the first operand remains unchanged.

Condition Code:

0	Result	=	0
1	Result	<	0
2	Result	>	0
3	--		

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

SRA Example

Op 04	R_1 5	I_2 3

0 Bits 8 12 15

Assembler: SRA R_1, I_2

Machine: 0453

	Before	After
R_1(5):	C5E6	FC5E
Condition Code:	x	1

SHIFT RIGHT LOGICAL (SRL)

Instruction Description

The first operand is shifted right the number of bits specified by I_2.

Format: RR

02	R_1	I_2

0 Bits 8 12 15

Operation: The value in I_2 is 1 less than the number of bits to be shifted.

All 16 bits of the first operand participate in the shift right, and zeros are supplied to the vacated leftmost register positions. Bits shifted out are lost.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

SRL Example

Op	R_1	I_2
02	4	3

0 Bits 8 12 15

Assembler: SRL R_1, I_2

Machine: 0243

	Before	After
R_1(4):	C5E6	0C5E

This page is intentionally left blank.

STACK (STACK)

Instruction Description

The Stack instruction obtains and loads the address of the next stack entry into the first operand. The stack instruction is used by VMC to control the allocation of storage. The storage area is organized like a stack. Every storage allocation starts on a 16-byte boundary. This allows a variable in storage to have any of the following boundary alignments: byte, halfword, word, doubleword, or quadword.

The maximum size of a storage allocation is limited to 64 K-16 bytes.

Format: RR

1B	B_1	R_2
0 Bits	8	12 15

Operation: The size of the next stack entry is indicated by the second operand.

The first operand initially contains the address of the current stack entry ❶. The first 8 bytes of the current stack entry contain four halfword fields that are used by the Stack and Unstack instructions. The first halfword contains an offset value ❷ which, when concatenated to the SID (segment identifier) portion of the current stack entry address, forms the address of the first byte of the next stack entry. The second halfword in the current stack entry contains an upper limit ❸ for the stack. The third and fourth halfwords are not used by the Stack instruction.

The address of the stack entry following the next entry ❹ is formed by adding the value contained in the first halfword of the current stack entry to the second operand. Both values are considered to be 16-bit unsigned binary integers. If no overflow occurs, the sum is logically compared with the limit value contained in the second halfword of the current stack entry. If the sum is greater than the limit value, a stack exception occurs and the operation is suppressed. If no stack exception is found, the sum is then checked to ensure that it is a multiple of eight (doubleword aligned). If it is not, a specification exception occurs and the operation is suppressed.

The following information is then stored into the first 8 bytes of the next stack entry:

Bytes	Description
0-1	Address of the stack entry following the next entry.
2-3	Stack limit value from current stack entry.
4-5	Address (offset portion) of the current stack entry.
6-7	Flag field that is set to all zeros.

Finally, the offset portion of the first-operand register is loaded with the offset portion of the next stack entry address, thus making this next entry the new current entry.

Overflow: If an overflow occurs as the result of add operation, an effective address overflow exception occurs and the operation is suppressed.

Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Initially, the first operand must start on a doubleword boundary; otherwise a specification exception occurs and the operation is suppressed.

The concatenation of the offset value from the first halfword of the current stack to the SID portion of the current stack entry address must be doubleword aligned; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Addressing
- Address translation
- Effective address overflow
- Specification
- Stack

Programming Note: If the operand specifies a length of zero, the results are unpredictable.

STACK Example

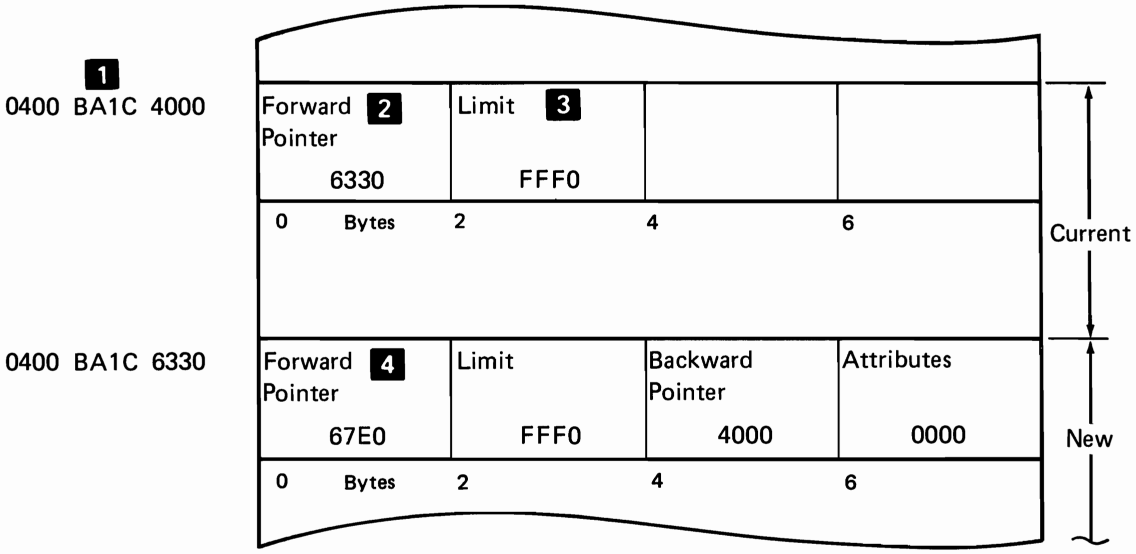
Op	B_1	R_2
1B	3	5

0 Bits 8 12 15

Assembler: STACK B_1, R_2

Machine: 1B35

	Before	After
B_1(3):	0400 BA1C 4000	0400 BA1C 6330
R_2(5):	04B0	



STORE (ST)

Instruction Description

The first operand is stored at the second-operand location.

Format: RS

96	B_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operands must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

The concatenation of the offset value from the first halfword of the current stack to the SID portion of the current stack entry address must be doubleword aligned; otherwise, a specification exceptions occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

ST Example

Op 96	B_1 3	E 0	B_2 4	D_2 250
0 Bits	8	12	16	20 31

Assembler: ST B_1, $D_2(B_2)$

Machine: 9630 4250

$B_1(3)$: 04A5 2330 0000

$B_2(4)$: 04C6 3250 1000

Storage — Before

	0/8	2/A	4/C	6/E
04C6 3250 1250	xxxx	xxxx	xxxx	

Storage — After

	0/8	2/A	4/C	6/E
04C6 3250 1250	04A5	2330	0000	

STORE AND SET COMPUTATIONAL ATTRIBUTES (SSCA)

Instruction Description

The current computational attributes of the task are stored into the receiver (operand 1). The new computational attributes of the task are set from the source (operand 2) as determined by the controls operand (operand 3). In addition, the current computational attributes of the task can optionally be stored in the current invocation control block. The invocation control block is addressed by base register 3, and the attributes are stored under control of indicators in the invocation control block.

Format: SS

BE	B_3	8	B_1	D_1	B_2	D_2
0 Bits	8	12	16	20	32	36 47

Operation: The first operand, the receiver, is always specified and is addressed through the base displacement (bits 16 through 31).

The second operation, the source, is optional if third operand is not specified. The absence of the second operand is indicated by a value of all zeros for bits 32 through 47. A nonzero value for bits 32 through 47 specifies the base displacement to be used to address the second operand. If the second operand is not specified when the third operand is specified, a specification exception results.

The third operand, the controls, is optional. The absence of the third operand is indicated by a value of all zeros for bits 8 through 11. A nonzero value for bits 8 through 11 specifies the base register to be used to address the third operand.

All three operands (receiver, source, and controls) have the same format with the meaning of the values contained within them varying slightly. The common operand format is as follows:

Exception Mask	Exception Occurrence	Comp
0 Bytes	2	4

Exception Mask

Byte(s) **0-1**	**Bits**	**Meaning**
	0-9	Reserved (binary 0)
	10	Floating-point overflow
	11	Floating-point underflow
	12	Floating-point zero divide
	13	Floating-point inexact result
	14	Floating-point invalid operand
	15	Reserved (binary 0)

Exception Occurrence

Byte(s) **2-3**	**Bits**	**Meaning**
	0-9	Reserved (binary 0)
	10	Floating-point overflow
	11	Floating-point underflow
	12	Floating-point zero divide
	13	Floating-point inexact result
	14	Floating-point invalid operand
	15	Reserved (binary 0)

Computational Mode

Byte 4	Bits	Meaning
	0	Reserved (binary 0)
	1-2	Rounding mode 00 = Round towards positive infinity 01 = Round towards negative infinity 10 = Round towards zero 11 = Round to nearest
	3-7	Reserved (binary 0)

The receiver and source operand bit values for the exception mask and occurrence bits have the same meaning.

Exception Mask Bits	Exception Occurrence Bits
0 = Exception is masked	0 = Exception has not occurred
1 = Exception is unmasked	1 = Exception has occurred

The meaning of the receiver and source operand bit values for the computational mode are as defined under the operand format previously described.

The bit values in the controls operand determines which computational attributes of the task are to be set from the bit values in the source operand. A value of 0 for a bit in the controls operand indicates that the corresponding computational attribute of the task is not to be set from the value of that bit of the source operand. A value of 1 for a bit in the controls operand indicates that the corresponding computational attribute of the task is to be set from the value of that bit on the source operand. For an attribute controlled by a multiple bit field, such as the rounding modes, all of the bits in the field must be ones or all of the bits must be zeros. A mixture of ones and zeros in such a field causes a specification exception.

The operation performed by the instruction is dependent on the number of operands specified.

The initial function of storing the computational attributes of the task is always performed. The receiver is set with bit values that reflect the computational attributes in effect at the start of execution of this instruction. Additionally, if the computational attributes of the task are to be altered by the value of the source operand, the computational attributes of the task are optionally stored into the current invocation control block (ICB) addressed by base register hex 3. In the ICB, if bit 7 of the byte at hex offset 06, contains a value of 1 (see Note 1) and bit 0 of the byte at offset hex A9 contains a value of 0 (see Note 2), the computational attributes of the task are stored into the ICB at offset hex E0 (see Note 3) in the format defined for the receiver operand. Also, in this case, bit 0 at byte offset hex A9 is set with a value of 1 to indicate the attributes have been stored.

If the source (operand 2) is specified without the controls (operand 3), the computational attributes of the task are set with the attributes specified in the source operand. If the source is not specified, the instruction does not alter the computational attributes of the task.

If the controls (operand 3) is specified, the source, operand 2, must also be specified. The computational attributes of the task are set with those attributes specified in the source for which the controls contains corresponding bit values of 1. Bit values of 0 in the controls indicate that the corresponding attribute is not to be set from the value in the source operand. If the controls are not specified, the computational attributes of the task are set with all of the values specified in the source operand.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The result obtained from overlapping operands is unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

Notes:
1. This bit, AlITYPE, indicates the ICB is an MI ICB if it has a value of 1.
2. This bit, currently reserved in the ICB, is used to indicate whether the computational attributes have already been stored for this MI invocation. A value of 0 indicates they have not been stored. A value of 1 indicates they have been stored.
3. This area of the ICB, a new extension, is used as the storage area for the 5 bytes of computational attribute information to be stored on an MI invocation basis. This area will only be set with this information by this instruction upon a change to the attributes, thereby avoiding this overhead to the Call External instruction path. It allows for Return External Instruction exception handing, and invocation exit handling, to restore the attributes to those that were in effect prior to an MI invocation which is being destroyed.

SSCA Example

OP	B_3	E	B_1	D_1	B_2	D_2
BE	F	8	2	083	2	088

0 Bits 8 12 16 20 32 36 47

Assembler: SSCA $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: BEF8 2083 2088

$B_1(2)$ and $B_2(2)$: 800D 0C00 0000

$B_3(F)$: 800D 0C00 0301

Storage — Before

	0/8	2/A	4/C	6/E
800D 0C00 0083		xx	xxxx	xxxx
800D 0C00 0088	001C	0000	E0	
800D 0C00 0300	0004	0004	80	

Storage — After

	0/8	2/A	4/C	6/E
800D 0C00 0083		00	3A00	0460
800D 0C00 0088	001C	0000	E0	
800D 0C00 0300	0004	0004	80	

STORE AND SET TAGS (STST)

Instruction Description

The STST instruction provides support for building a Machine Interface pointer from the address value contained in the first operand. The address value contained in the first operand is stored in the pointer along with the segment group extender.

A quadword containing a virtual address is tagged and is stored at the second-operand location.

Format: RS

65	B_1	I_3	B_2	D_2	
0 Bits	8	12	16	20	31

Operation: The quadword is formed as follows:

- Bits 0 and 1 of byte 0 [1] come from the leftmost 2 bits of the I_3 field.

Pointer Bits 0 and 1	Meaning
00	System Pointer
01	Instruction Pointer
10	Space Pointer
11	Data Pointer

- Bits 2 through 7 of byte 0 [2] are reset.

- Bytes 1 through 7 [3] are reset.

- Bytes 8 and 9 come from a halfword in storage [4] whose address is determined by taking bytes 0, 1, and 2 of the first operand and concatenating hex 0 0004 on the right.

- Bytes hex A-F come directly from the first operand, bytes 0-5 [5].

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The second operand must start on a quadword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STST Example

Op	B_1	I_3	B_2	D_2
65	5	8	2	0F0

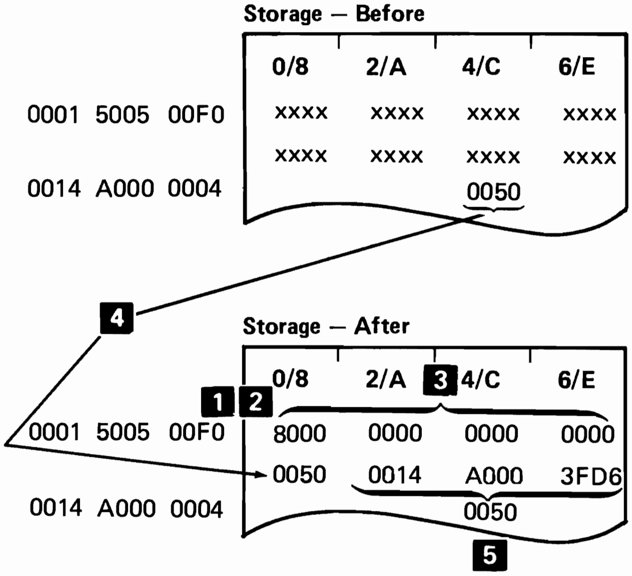
0 Bits 8 12 16 20 31

Assembler: STST B_1, $D_2(B_2)$, I_3

Machine: 6558 20F0

$B_1(5)$: $\underbrace{\text{0014 A000 3FD6}}_{\boxed{5}}$

$B_2(2)$: 0001 5005 0000



Storage — Before

	0/8	2/A	4/C	6/E
0001 5005 00F0	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
0014 A000 0004			0050	

Storage — After

	0/8	2/A	4/C	6/E
0001 5005 00F0	8000	0000	0000	0000
	0050	0014	A000	3FD6
0014 A000 0004			0050	

STORE BYTE (STB)

Instruction Description

The first operand is stored at the second-operand location.

Format: RS

76	r_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

STB Example

Op 76	r_1 8	E 0	B_2 5	D_2 100

0 Bits 8 12 16 20 31

Assembler: STB r_1, $D_2(B_2)$

Machine: 7680 5100

$r_1(8)$: D3

$B_2(5)$: 000C 1234 0000

Storage — Before

	0/8	2/A	4/C	6/E
000C 1234 0100	xx			

Storage — After

	0/8	2/A	4/C	6/E
000C 1234 0100	D3			

This page is intentionally left blank.

STORE CLOCK COMPARATOR (STCC)

Instruction Description

The current value of the clock comparator is stored at the first-operand location.

Format: SI

6D		3	B_1	D_1
0 Bits	8	12	16	20 31

Operation: Zeros are provided for the rightmost bit positions that are not used for comparison with the time-of-day clock. If no Set Clock Comparator instruction has been issued prior to the STCC, an unpredictable value is stored.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand occupies 8 bytes in storage and must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STCC Example

Op 6D		E 3	B_1 5	D_1 2C0
0 Bits	8	12	16	20 31

Assembler: STCC $D_1(B_1)$

Machine: 6D03 52C0

$B_1(5)$: 01A0 CDEF 0000

Clock Comparator — Before and After

0000	0000	0000	0101	1010	0001	0000	1011	0010	0010	10	00	0001	0011	1110	1100	0100
0					Bits						42				56	64

Storage — Before

	0/8	2/A	4/C	6/E
01A0 CDEF 02C0	xxxx	xxxx	xxxx	xxxx

Storage — After

	0/8	2/A	4/C	6/E
01A0 CDEF 02C0	0005	A10B	2280	0000

STORE HALFWORD (STH)

Instruction Description

The first operand is stored at the second-operand location.

Format: RS

86	R_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: See *Instruction Description.*

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operands must each start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STH Example

Op	R_1	E	B_2	D_2
86	4	0	3	AA0

0 Bits 8 12 16 20 31

Assembler: R_1, $D_2(B_2)$

Machine: 8640 3AA0

B_2(3): 215A C158 0000

R_1(4): FFFF

Storage – Before

	0/8	2/A	4/C	6/E
215A C158 0000	xxxx			

Storage – After

	0/8	2/A	4/C	6/E
215A C158 0000	FFFF			

This page is intentionally left blank.

STORE INTERVAL TIMER (STIT)

Instruction Description

The current value in one of the interval timers is stored at the first-operand location.

Format: SI

8C	I_2		B_1	D_1
0 Bits	8	12	16	20 31

Operation: Zeros are provided for the rightmost bit positions that are not updated by the interval timer. If no Set Interval Timer instruction has been issued prior to the STIT instruction, an unpredictable value is stored. Since an untimed task cannot set the task interval timer, any STIT instruction issued by an untimed task to the task interval timer results in an unpredictable value being stored.

The second operand selects the specific timer for the store operation. The second operand values of hex 02-0F causes a specification exception and the operation is suppressed.

I-Field	Timer
Hex 00	First interval timer (also used as task interval timer)
Hex 01	Second interval timer
Hex 02–FF	Invalid

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The first operand occupies 8 bytes in storage and must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Execeptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STIT Example

Op	I_2	B_1	D_1
8C	01	3	220

0 Bits 8 16 20 31

Assembler: STIT $D_1(B_1)$, I_2

Machine: 8C01 3220

B_1(3): 2F1A 3CD1 0000

Interval Timer

0001	1010	1011	0010	0000	0011	1100	0100	0101	1101	01	10	0111	1000	1110	1111	1011

0 Bits 42 56 64

Storage — Before

	0/8	2/A	4/C	6/E
2F1A 3CD1 0220	xxxx	xxxx	xxxx	xxxx

Storage — After

	0/8	2/A	4/C	6/E
2F1A 3CD1 0220	1AB2	03C4	5D40	0000

STORE MULTIPLE (STM)

Instruction Description

A set of registers is stored at the locations designated by the second-operand address.

Format: RS

97	B_1	I_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The first-operand field identifies the first register to be stored, and the I_3 field specifies the number of additional registers to be stored.

The storage area where the contents of the registers are stored starts at the location specified by the second-operand address and continues through as many locations as needed.

The registers are stored in the ascending order of their addresses, starting with the register specified by the first operand. The register addresses wraparound from hex F to 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operands must each start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STM Example

Op	B_1	I_3	B_2	D_2
97	0	4	C	1B0

0 Bits 8 12 16 20 31

Assembler: STM B_1, I_3, $D_2(B_2)$

Machine: 9704 C1B0

B_2(C): 003F CFD5 0000

B(0):	001B	2A30	0000
B(1):	0C3B	1234	4000
B(2):	0001	ABCD	E000
B(3):	213A	C1F4	6000
B(4):	A100	000B	A000

Storage – Before

	0/8	2/A	4/C	6/E
003F CFD5 0000	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	xxxx
	xxxx	xxxx	xxxx	

Storage – After

	0/8	2/A	4/C	6/E
003F CFD5 0000	001B	2A30	0000	0C3B
	1234	4000	0001	ABCD
	E000	213A	C1F4	6000
	A100	000B	A000	

STORE MULTIPLE BYTE (STMB)

Instruction Description

A set of registers is stored at the locations designated by the second-operand address.

Format: RS

77	r_1	I_3	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The first-operand field identifies the first register to be stored, and the I_3 field specifies the number of additional registers to be stored.

The storage area where the contents of the registers are stored starts at the location specified by the second-operand address and continues through as many locations as needed.

The registers are stored in the ascending order of their addresses, starting with the register specified by the first operand. The register addresses wraparound from hex F to 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

STMB Example

Op 77	r_1 D	I_3 3	B_2 3	D_2 000
0 Bits	8	12	16	20 31

Assembler: STMB r_1, I_3, $D_2(B_2)$

Machine: 77D3 3000

B_2(3): 5FC3 0001 C000

r(D): 2A
r(E): CB
r(F): E1
r(0): 4D

Storage — Before

	0/8	2/A	4/C	6/E
5FC3 0001 C00D			xx	xxxx
	xx			

Storage — After

	0/8	2/A	4/C	6/E
5FC3 0001 C00D			2A	CBE1
	4D			

STORE MULTIPLE HALFWORD (STMH)

Instruction Description

A set of registers is stored at the locations designated by the second-operand address.

Format: RS

87	R_1	I_3	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The first-operand field identifies the first register to be stored, and the I-field specifies the number of additional registers to be stored.

The storage area where the contents of the registers are stored starts at the location specified by the second-operand address and continues through as many locations as needed.

The registers are stored in the ascending order of their addresses, starting with the register specified by the first operand. The register addresses wraparound from hex F to 0.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The storage operands must each start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STMH Example

Op	R_1	I_3	B_2	D_2
87	5	2	3	100

0 Bits 8 12 16 20 31

Assembler: STMH R_1, I_3, $D_2(B_2)$

Machine: 8752 3100

B_2(3): 000A 000B 0000

R(5): 1234
R(6): 5678
R(7): 9ABC

Storage – Before

	0/8	2/A	4/C	6/E
000A 000B 0100	xxxx	xxxx	xxxx	

Storage – After

	0/8	2/A	4/C	6/E
000A 000B 0100	1234	5678	9ABC	

STORE SPACE OFFSET POINTER (STSOP)

Instruction Description

The segment group offset (low-order 3 bytes) of the first operand is decremented by the space locator offset referenced by the first operand; the 4-byte result is stored in the second operand.

Format: RS

83	B_1	9	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The space locator is a 3-byte logical binary field located at the storage address found by concatenating hex 00 0001 to the right of the high-order 3 bytes (segment group identifier) of the first-operand address. The space locator is logically subtracted from the low-order 3 bytes (segment group offset) of the address found in the first operand. If a binary underflow results, a specification exception is recognized and the operation is suppressed. Otherwise, the resultant 3-byte difference is padded on the left with 1 byte of zeros, and the result is stored at the location specified by the second operand.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STSOP Example

Op	B_1	E	B_2	D_2
83	8	9	2	020

0 Bits 8 12 16 20 31

Assembler: STSOP B_1,D_2(B_2)

Machine: 8389 2020

	Before	After
B_1(8):	0010 0212 3456	0010 0212 3456
B_2(2):	00C1 B000 4BE0	00C1 B000 4BE0

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 001D	xx		00	3456
00C1 B000 4BE0	xxxx	xxxx		

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 01D0	0034	56xx		
00C1 B000 4BE0	0012	0000		

STORE TIME-OF-DAY CLOCK (STTOD)

Instruction Description

The current value of the time-of-day clock is stored at the operand location.

Format: SI

6D		5	B_1	D_1
0 Bits	8	12	16	20 31

Operation: The value of the clock is expressed as an unsigned 64-bit binary number. Successive STTOD instructions ensure unique values by adding a binary 1 to bits 56-63. Zeros are provided for the bit positions to the left of bit position 56 that are not updated by the time-of-day clock. If a Set Time of Day Clock instruction is not issued prior to the STTOD instruction, an unpredictable result is stored.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: A carry from bit position 56 is ignored.

Boundary Requirements: The first operand occupies 8 bytes in storage and must begin on a word boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

STTOD Example

Op		E	B1	D1
6D		5	3	CA0

0 Bits 8 12 16 20 31

Assembler: STTOD $D_1(B_1)$

Machine: 6D05 3CA0

$B_1(3)$: 0005 35A6 1000

Time-of-Day Clock

0000	0000	0010	1010	0001	0010	1100	1101	0010	0000	00	00	0000	0000	0000	0110	0001

0 Bits 42 56 64

Storage — Before

	0/8	2/A	4/C	6/E
0005 35A6 1CA0	xxxx	xxxx	xxxx	xxxx

Storage — After

	0/8	2/A	4/C	6/E
0005 35A6 1CA0	002A	12CD	2000	0062

SUBTRACT CHARACTERS (SC)

Instruction Description

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

Format: SS

C1	L_1	L_2	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: The operands are treated as signed binary quantities. Subtraction is performed as if the ones complement of the second operand and a rightmost one bit were added to the first operand. If the operands are unequal in length, the shorter operand is considered to be extended to the left with bits equal to the sign bit.

Overflow: If the carry from the sign bit position and the carry from the high-order numeric bit position agree, no overflow occurs; if they disagree, an overflow occurs.

If the first operand is too short to contain all significant bits of the result, an overflow occurs and significant bits are lost.

Sign Code: The sign bit of the difference is not changed after the overflow. A positive result that overflows yields a negative difference and a negative result that overflows yields a positive difference.

Note that the sign of the difference is unpredictable when significant bits are lost (see *Overflow*).

Condition Code: If an overflow occurs, the condition code indicates the sign the difference would have if an overflow had not occurred. When significant bits are lost, the condition code indicates the sign the difference would have if the first operand had been long enough to contain all significant bits of the result.

0	Difference = 0
1	Difference < 0
2	Difference > 0
3	--

Carry: See *Overflow.*

Boundary Requirements: The operands can overlap in storage if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Effective address overflow

SC Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
C1	3	3	3	1A0	4	B20

0 Bits 8 12 16 20 32 36 47

Assembler: SC $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: C133 31A0 4B20

B_1(3): 01C2 6430 0000

B_2(4): 02B0 6320 0000

Storage — Before

	0/8	2/A	4/C	6/E
01C2 6430 01A0	001E	8480		
02B0 6320 0B20	0007	A120		

Storage — After

	0/8	2/A	4/C	6/E
01C2 6430 01A0	0016	E360		
02B0 6320 0B20	0007	A120		

	Before	After
Condition Code:	x	2

SUBTRACT HALFWORD (SH)

Instruction Description

The second operand is subtracted from the first operand and the difference is placed in the first-operand register.

Format: RS

80	R_1	1	B_2	D_2	
0 Bits	8	12	16	20	31

Operation: The operands are treated as signed binary quantities. Subtraction is performed as if the ones complement of the second operand and a rightmost 1 bit were added to the first operand.

Overflow: If the carry from the sign bit position and the carry from the high-order numeric bit position agree, no overflow occurs; if they disagree, an overflow occurs.

Sign Code: The sign bit of the difference is not changed after the overflow. A positive result that overflows yields a negative difference and a negative result that overflows yields a positive difference.

Condition Code: If the overflow occurs the condition code indicates the sign the difference would have if overflow had not occurred.

0	Difference = 0
1	Difference < 0
2	Difference > 0
3	--

Carry: See *Overflow.*

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception is recognized, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Effective address overflow
- Specification

SH Example

Op 80	R_1 6	E 1	B_2 3	D_2 0A0	
0 Bits	8	12	16	20	31

Assembler: SH R_1, $D_2(B_2)$

Machine: 8061 30A0

B_2(3): 001F F100 10A0

	Before	After
R_1(6):	1388	0FA0
Condition Code:	x	2

Storage – Before and After

	0/8	2/A	4/C	6/E
001F F100 1140	03E8			

SUBTRACT HALFWORD REGISTER (SHR)

Instruction Description

The second operand is subtracted from the first operand and the difference is placed in the first-operand register.

Format: RR

21	R_1	R_2

0 Bits 8 12 15

Operation: The operands are treated as unsigned binary quantities. Subtraction is performed as if the ones complement of the second operand and a rightmost 1 bit were added to the first operand.

Overflow: If the carry from the sign bit position and the carry from the high-order numeric bit position agree, the difference is satisfactory; if they disagree, an overflow occurs.

Sign Code: The sign bit of the difference is not changed after the overflow. A positive result that overflows yields a negative difference, and a negative result that overflows yields a positive difference.

Condition Code: If an overflow occurs the condition code indicates the sign the difference would have if overflow had not occurred.

0	Difference = 0
1	Difference < 0
2	Difference > 0
3	--

Carry: See *Overflow.*

Boundary Requirements: None.

Program Exceptions: Binary overflow.

SHR Example

Op 21	R_1 6	R_2 7

0 Bits 8 12 15

Assembler: SHR R_1, R_2

Machine: 2167

	Before	After
R_1:	1388	0FA0
R_2:	03E8	03E8
Condition Code:	x	2

SUBTRACT LOGICAL BYTE (SLB)

Instruction Description

The second operand is subtracted from the first operand and the difference is placed in the first-operand register.

Format: RS

71	r_1	0	B_2	D_2

0 Bits 8 12 16 20 31

Operation: The operands are treated as unsigned binary quantities. Subtraction is performed as if the ones complement of the second operand and a low-order 1 bit were added to the first operand.

Overflow and Sign Code: Not applicable.

Condition Code:

0	--
1	Difference ≠ 0, with no carry
2	Difference = 0, with carry
3	Difference ≠ 0, with carry

Carry: A carry from the high-order bit position is recorded in the condition code.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

SLB Example

Op 71	r_1 5	E 0	B_2 3	D_2 C00

0 Bits 8 12 16 20 31

Assembler: SLB r_1, D_2 (B_2)

Machine: 7150 3C00

B_2 (3): CA20 1254 3000

	Before	After
r_1 (5):	64	32
Condition Code	x	1

Storage — Before and After

	0/8	2/A	4/C	6/E
CA20 1254 3C00	32			

SUBTRACT LOGICAL BYTE REGISTER (SLBR)

Instruction Description

The second operand is subtracted from the first operand and the difference is placed in the first-operand register.

Format: RR

11	r_1	r_2

0 Bits 8 12 15

Operation: The operands are treated as unsigned binary quantities. Subtraction is performed as if the ones complement of the second operand and a low-order 1 bit were added to the first operand.

Overflow and Sign Code: Not applicable.

Condition Code:

0 --
1 Difference ≠ 0, with no carry
2 Difference = 0, with carry
3 Difference ≠ 0, with carry

Carry: A carry from the high-order bit position is recorded in the condition code.

Boundary Requirements and Program Exceptions: None.

SLBR Example

Op 11	r_1 3	r_2 4

0 Bits 8 12 15

Assembler: SLBR r_1, r_2

Machine: 1134

	Before	After
r_1(3):	64	32
r_2(4):	32	32
Condition Code:	x	1

SUBTRACT LOGICAL CHARACTERS (SLC)

Instruction Description

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

Format: SS

C4	L	B_1	D_1	B_2	D_2

0 Bits 8 16 20 32 36 47

Operation: The operands are treated as unsigned binary quantities. Subtraction is performed as if the ones complement of the second operand and a low-order 1 bit were added to the first operand.

Overflow and Sign Code: Not applicable.

Condition code:

0	--
1	Difference ≠ 0, with no carry
2	Difference = 0, with carry
3	Difference ≠ 0, with carry

Carry: A carry from the high-order bit position is recorded in the condition code.

Boundary Requirements: The operands can overlap if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

SLC Example

Op	L_1	B_1	D_1	B_2	D_2
C4	05	3	520	4	6A0

0 Bits 8 16 20 32 36 47

Assembler: SLC $D_1(L_1, B_1), D_2(B_2)$

Machine: C405 3520 46A0

B_1(3): 12C4 1131 1000

B_2(4): 12C4 1133 5000

Storage — Before

	0/8	2/A	4/C	6/E
12C4 1131 1520	1234	5678	9ABC	
12C4 1133 56A0	0000	1234	5678	

Storage — After

	0/8	2/A	4/C	6/E
12C4 1131 1520	1234	4444	4444	
12C4 1133 56A0	0000	1234	5678	

	Before	After
Condition Code:	x	1

SUBTRACT LOGICAL HALFWORD (SLH)

Instruction Description

The second operand is subtracted from the first operand and the difference is placed in the first-operand register.

Format: RS

91	R_1	0	B_2	D_2
0 Bits	8	12	16	20 31

Operation: The operands are treated as unsigned binary quantities. Subtraction is performed as if the ones complement of the second operand and a low-order 1 bit were added to the first operand.

Overflow and Sign Code: Not applicable.

Condition Code:

0 --
1 Difference ≠ 0, with no carry
2 Difference = 0, with carry
3 Difference ≠ 0, with carry

Carry: A carry from the high-order bit position is recorded in the condition code.

Boundary Requirements: The storage operand must start on a halfword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

SLH Example

Op 91	R_1 3	E 0	B_2 7	D_2 150
0 Bits	8	12	16	20 31

Assembler: SLH R_1, $D_2(B_2)$

Machine: 9130 7150

B_2(7): 0ABC 0000 1000

	Before	After
R_1(3):	ABCD	8108
Condition Code:	x	1

Storage — Before and After

	0/8	2/A	4/C	6/E
0ABC 0000 1150	2AC5			

SUBTRACT LOGICAL HALFWORD REGISTER (SLHR)

Instruction Description

The second operand is subtracted from the first operand and the difference is placed in the first-operand register.

Format: RR

31	R_1	R_2

0 Bits 8 12 15

Operation: The operands are treated as unsigned binary quantities. Subtraction is performed as if the ones complement of the second operand and a low-order 1 bit were added to the first operand.

Overflow and Sign Code: Not applicable.

Condition Code:

0	--
1	Difference ≠ 0, with no carry
2	Difference = 0, with carry
3	Difference ≠ 0, with carry

Carry: A carry from the high-order bit position is recorded in the condition code.

Boundary Requirements and Program Exceptions: None.

SLHR Example

Op	R_1	R_2
31	3	4

0 Bits 8 12 15

Assembler: SLHR R_1, R_2

Machine: 3134

	Before	After
R_1(3):	ABCD	8108
R_2(4):	2AC5	2AC5
Condition Code:	x	1

SUBTRACT LONG FLOAT (SLF)

Instruction Description

The second operand is subtracted from the first operand (two-operand format) or the third operand is subtracted from the second operand (three-operand format), and the result is placed in the first operand location.

Format: SS

CE	B_3	2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: A two-operand or three-operand format may be specified. A two-operand format is used, if base register 0 is specified for the third operand. A three-operand format is used, if one of the base registers hex 1 through hex F is specified for the third operand.

The exponents of the two operands are compared. The significand of the smaller exponent is shifted right as its exponent is increased until the exponents are the same. The sign bit of the subtrahend (significand of either the second operand or the third operand for either two-operand or three-operand format respectively) is changed. The significands are then added algebraically to form the intermediate difference.

The intermediate difference is rounded, if necessary, according to the rounding mode specified in the task dispatching element.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the difference. The source operands are checked for this value in order of their specification. The masked not-a-number with the larger fraction value is provided as the difference.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, but its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented exactly (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the difference is determined by the rules of algebra. If the difference of two operands that have the same sign is 0, the sign is made plus for all rounding modes except round toward negative infinity, where the sign is made minus.

Condition Code: The difference is compared to 0. Values of 0 compare equal even if they differ in sign. Not-a-number values and infinite values compare unordered.

0	Difference = 0
1	Difference < 0
2	Difference > 0
3	Difference is unordered

Carry: Not applicable.

Boundary Requirements: All operands must be fullword aligned; otherwise, a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

Programming Note: The following chart shows the condition of the difference for various operands.

Difference	First Source (Minuend)	Second Source (Subtrahend)
+0	+0	-0
-0	-0	+0
+0	-Real number ≠ 0	-Real number ≠ 0
+0	+Real number ≠ 0	+Real number ≠ 0
+Real number ≠ 0	+Real number ≠ 0	+0 or -0
+Real number ≠ 0	+0 or -0	-Real number ≠ 0
-Real number ≠ 0	+0 or -0	+Real number ≠ 0
-Real number ≠ 0	-Real number ≠ 0	+0 or -0
Masked not-a-member	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-member	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operand	Unmasked not-a-number	Any
Invalid operand	Any	Unmasked not-a-number
Invalid operand	+Infinity or -infinity	+Infinity or -infinity
+Infinity	+Real number ≠ 0 or -real number ≠ 0	-Infinity
+Infinity	+Infinity	+Real number ≠ 0, -real number ≠ 0, or 0
-Infinity	+Real number ≠ 0, -real number ≠ 0, or 0	+Infinity
-Infinity	-Infinity	+Real number ≠=, -real number ≠ 0, or 0
+0	+0	+0 Note 1
+0	-0	-0 Note 1
-0	+0	+0 Note 2
-0	-0	-0 Note 2

Legend:

Not not-a-member = Anthing but a not-a-number

Any = Any floating-point field value

Notes:

1. Value is not rounded toward negative infinity
2. Value is rounded negative infinity

SLF Example

Op	B_3	E	B_1	D_1	B_2	D_2
CE	3	2	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: SLF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: CE32 4050 4060

$B_3(3)$: 0010 0200 0070

$B_1(4)$ and $B_2(4)$: 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	4EC0	1234	5678	9ABC
0010 0200 0070	4EB0	1234	5678	9ABC

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	4EB0	1234	5678	9ABC
0010 0200 0060	4EC0	1234	5678	9ABC
0010 0200 0070	4EB0	1234	5678	9ABC

	Before	After
Condition Code:	x	2

SUBTRACT PACKED (SP)

Instruction Description

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

Format: SS

F1	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: Subtraction is algebraic, taking into account the signs and all digits of both operands. All digit codes are checked for validity. Improper codes cause a data exception to be recognized, and the operation is terminated. If necessary, zeros are supplied for the leftmost bytes of either operand.

Overflow: Overflow can occur due to the loss of a carry from the leftmost digit position of the result field, or due to an oversized result, which occurs when the second-operand field is larger than the first-operand field. Significant digits are lost when an overflow occurs.

Sign Code: The sign codes are checked for validity. Improper codes cause a data exception to be recognized, and the operation is terminated. The sign of the second operand, if negative, is treated as positive, and if positive, is treated as negative (this reversal is a normal function of subtraction).

The processor uses the preferred signs for the results as follows: a positive sign is encoded as 1111 (hex F); a negative sign is encoded as 1101 (hex D).

Condition Code: If an overflow occurs, the condition code always indicates the sign the difference would have had if an overflow had not occurred.

0	Difference = 0
1	Difference < 0
2	Difference > 0
3	--

Carry: See *Overflow*

Boundary Requirements: The first and second-operand fields can overlap when their rightmost bytes coincide.

Because digit and sign codes are checked for validity, improperly overlapping fields cause a data exception, and the operation is terminated.

Program Exceptions:

- Address translation
- Addressing
- Data
- Decimal overflow
- Effective address overflow

SP Example

Op	L_1	L_2	B_1	D_1	B_2	D_2
F1	7	3	3	100	3	104

0 Bits 8 12 16 20 32 36 47

Assembler: SP $D_1(L_1, B_1)$, $D_2(L_2, B_2)$

Machine: F173 3100 3104

B_1(3) and B_2(3): 0101 0202 3000

Storage — Before

	0/8	2/A	4/C	6/E
0101 0202 3100	1234	5678	2345	678F

Storage — After

	0/8	2/A	4/C	6/E
0101 0202 3100	1234	5678	0000	000F

	Before	**After**
Condition Code:	x	2

SUBTRACT SHORT FLOAT (SSF)

Instruction Description

The second operand is subtracted from the first operand (two-operand format) or the third operand is subtracted from the second operand (three-operand format), and the result is placed in the first operand location.

Format: SS

AE	B_3	2	B_1	D_1	B_2	D_2	
0 Bits	8	12	16	20	32	36	47

Operation: A two-operand or three-operand format may be specified. A two-operand format is used, if base register 0 is specified for the third operand. A three-operand format is used, if one of the base registers hex 1 through hex F is specified for the third operand.

The exponents of the two operands are compared. The significand of the smaller exponent is shifted right as its exponent is increased until the exponents are the same. The sign bit of the subtrahend (significand of either the second operand or the third operand for either a two-operand or three-operand format respectively) is changed. The significands are then added algebraically to form the intermediate difference.

The intermediate difference is rounded, if necessary, according to the rounding mode specified in the task dispatching element.

If a masked not-a-number value is encountered in one of the source operands, the operation is completed by providing the not-a-number value encountered as the difference. The source operands are checked for this value in order of their specification. The masked not-a-number with the larger fraction value being provided as the difference.

Overflow: A floating-point overflow exception occurs if a rounded result is finite, buts its exponent is too large to be represented in the result format. See *Floating-Point Overflow Exception* in Chapter 6 for further information.

A floating-point underflow exception occurs if a result is not a normal 0 and, when examined, is found to have too small an exponent to be represented in the result format without being denormalized and if the number cannot be represented (as a denormalized number) or the underflow mask bit is enabled. See *Floating-Point Underflow Exception* in Chapter 6 for further information.

Sign Code: The sign of the difference is determined by the rules of algebra. If the difference of two operands that have the same sign is 0, the sign is made plus for all rounding modes except round toward negative infinity, where the sign is made minus.

Condition Code: The difference is compared to 0. Values of 0 compare equal even if they differ in sign. Not-a-number values and infinite values compare unordered.

0	Difference = 0
1	Difference < 0
2	Difference > 0
3	Difference is unordered

Carry: Not applicable.

Boundary Requirements: All operands must be fullword aligned; otherwise, a specification exception occurs, and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Floating-point inexact result
- Floating-point invalid operand
- Floating-point overflow
- Floating-point underflow
- Specification

Programming Note: The following chart shows the condition of the difference for various operands.

Difference	First Source (Minuend)	Second Source (Subtrahend)
+0	+0	-0
-0	-0	+0
+0	-Real number ≠ 0	-Real number ≠ 0
+0	+Real number ≠ 0	+Real number ≠ 0
+Real number ≠ 0	+Real number ≠ 0	+0 or -0
+Real number ≠ 0	+0 or -0	-Real number ≠ 0
-Real number ≠ 0	+0 or -0	+Real number ≠ 0
-Real number ≠ 0	-Real number ≠ 0	+0 or -0
Masked not-a-member	Masked not-a-number	Not not-a-number
Masked not-a-number	Not not-a-member	Masked not-a-number
Larger masked not-a-number	Masked not-a-number	Masked not-a-number
Invalid operand	Unmasked not-a-number	Any
Invalid operand	Any	Unmasked not-a-number
Invalid operand	+Infinity or -infinity	+Infinity or -infinity
+Infinity	+Real number ≠ 0 or -real number ≠ 0	-Infinity
+Infinity	+Infinity	+Real number ≠ 0, -real number ≠ 0, or 0
-Infinity	+Real number ≠ 0, -real number ≠ 0, or 0	+Infinity
-Infinity	-Infinity	+Real number ≠=, -real number ≠ 0, or 0
+0	+0	+0 Note 1
+0	-0	-0 Note 1
-0	+0	+0 Note 2
-0	-0	-0 Note 2

Legend:

Not not-a-member = Anthing but a not-a-number

Any = Any floating-point field value

Notes:

1. Value is not rounded toward negative infinity
2. Value is rounded negative infinity

SSF Example

Op	B_3	E	B_1	D_1	B_2	D_2
AE	3	2	4	050	4	060

0 Bits 8 12 16 20 32 36 47

Assembler: SSF $D_1(B_1)$, $D_2(B_2)$, B_3

Machine: AE32 4050 4060

B_3(3): 0010 0200 0070

B_1(4) and B_2(4): 0010 0200 0000

Storage — Before

	0/8	2/A	4/C	6/E
0010 0200 0050	xxxx	xxxx	xxxx	xxxx
0010 0200 0060	BF80	0000		
0010 0200 0070	4000	0000		

Storage — After

	0/8	2/A	4/C	6/E
0010 0200 0050	C040	0000		
0010 0200 0060	BF80	0000		
0010 0200 0070	4000	0000		

	Before	**After**
Condition Code:	x	1

This page is intentionally left blank.

SUPERVISOR EXIT (SVX)

Instruction Description

The routine that invoked the current SVL (supervisor linkage) is returned to by using the contents of the current SVL CRE (call/return element).

Format: RR

3E		0

0 Bits 8 12 15

Operation: The SVX instruction causes the condition code, IAR (instruction address register) or CSAR (control store address register), and the saved base registers to be restored from the first in-use CRE (call/return element) on the current TDE (task dispatching element) CRE chain (current SVL CRE). The exception code and ILC (instruction length count) fields in this CRE are ignored. The status of this CRE is then set to available (byte 8, bit 0 of CRE is reset). If the number of available CREs encountered before this CRE is equal to or greater than the number specified in the control mode field of the TDE, the first CRE is returned to the ACQ (available call/return queue) via an implicit send. This send causes the TDE on the ACQ wait list to be dequeued and subsequently enqueued to the TDQ, possibly causing a task switch to occur.

Execution of the SVX instruction may be interrupted by I/O. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point at which the interrupt was granted.

Overflow and Sign Code: Not applicable.

Condition Code: The code is set to the value saved in the CRE.

Carry: Not applicable.

Boundary Requirements: None.

Program Exception: Specification.

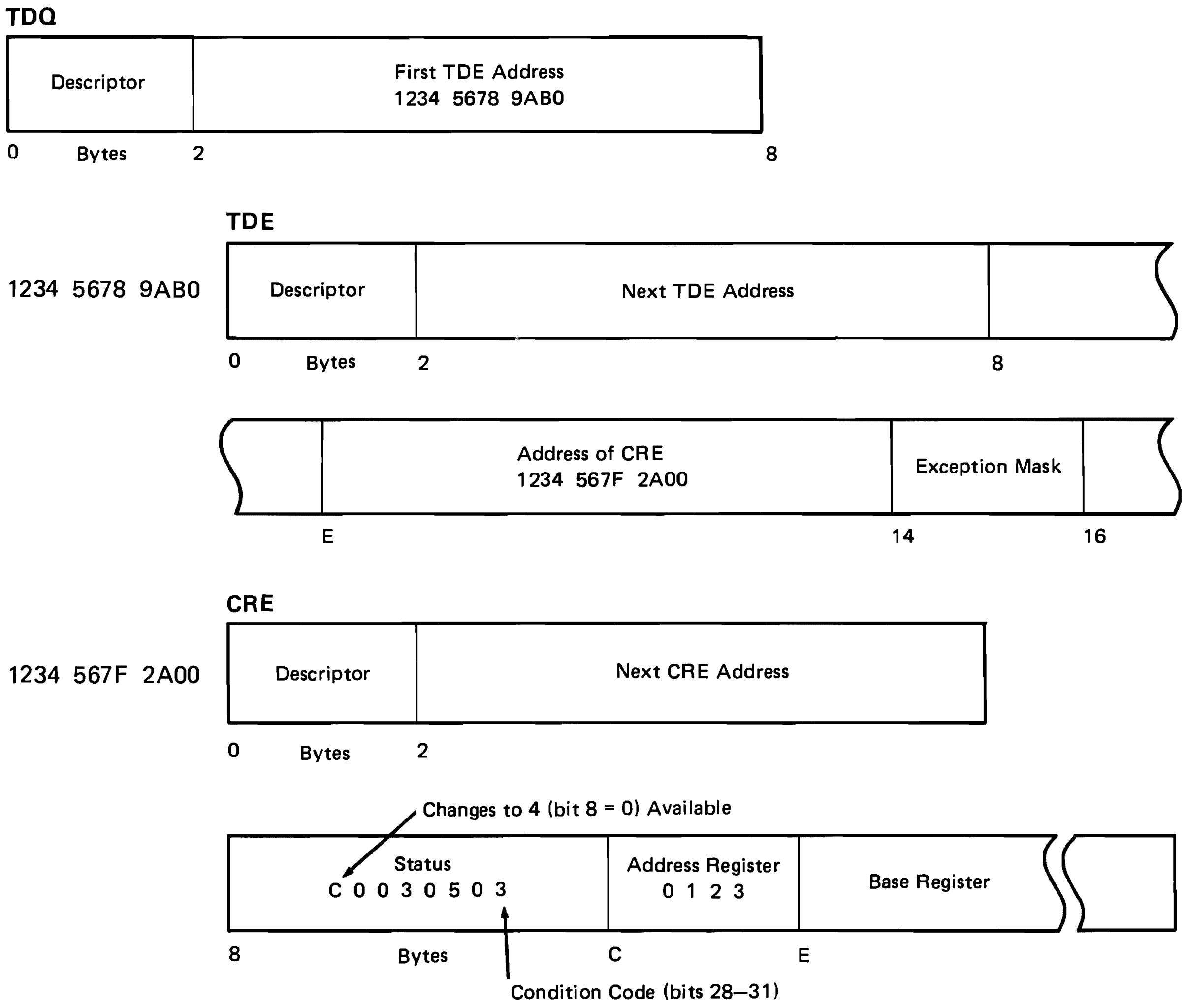
SVX Example

Op 3E		E 0

0 Bits 8 12 15

Assembler: SVX

Machine: 3E00



	Before	After
Condition Code:	x	3
IAR:	xxxx	0 1 2 3
B(0):	xxxx xxxx xxxx	CRE bytes E–13
B(1):	xxxx xxxx xxxx	CRE bytes 14–19
B(2):	xxxx xxxx xxxx	CRE bytes 1A–1F
B(3):	xxxx xxxx xxxx	CRE bytes 20–25

SUPERVISOR LINK DOUBLE (SVL2)

Instruction Description

The SVL (supervisor linkage) routir selected by the index in the I_3 field of the instruct.on is called using the supervisor link mechanism.

Format: SS

DF	I	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32	36 47

Operation: The index is used to access an entry in the SVL table to determine where the SVL routine is located and how many registers are to be saved (into an available CRE) prior to branching to the routine.

The effective addresses of the first and second operands are computed and checked for an effective address overflow exception. No attempt is made to access the first or second operands, and they remain unchanged in storage.

An available CRE (call/return element) is found by searching the CRE list chained to the current TDE (task dispatching element). The current status (IAR [instruction address register], condition code, identification of the first base register stored, and, the number of base registers stored), along with the contents of the specified base registers, are stored in the last available CRE on the list. If there are no available CREs on the list or if the list is empty, a CRE is implicitly received from the ACQ and is enqueued first on the TDE CRE list. The current status and base registers are then stored in that CRE. In either case, the status of the CRE obtained is set to in-use (byte 8, bit 0 = 1).

If it is necessary to obtain a CRE from the ACQ and the ACQ is empty, the implicit receive is unsatisfied and the SVL instruction is nullified. The TDE of the current task is then dequeued from the TDQ and enqueued to the ACQ wait list, and the task dispatcher is invoked.

After the registers specified in the SVL table entry are saved, the effective address of the first operand is placed in base register 1 and the effective address of the second operand is placed in base register 2.

Execution of the SVL2 instruction may be interrupted by I/O. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point at which the interrupt was granted.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Descriptor access: Monitored ACQ descriptor
- Descriptor access: Monitored CRE descriptor
- Descriptor access: Monitored TDE descriptor (SVL receive wait)
- Effective address overflow
- Specification

Programming Notes: If the task dispatcher is disabled and an SVL2 instruction is attempted, a machine check occurs. An index value of zero is valid, but is also used by the processor to signal exceptions.

SVL2 Example

Op	I_3	B_1	D_1	B_2	D_2
DF	nn	3	100	4	200
0 Bits	8	16	20	32	36 47

Assembler: SVL2 $D_1(B_1)$, $D_2(B_2)$, I_3

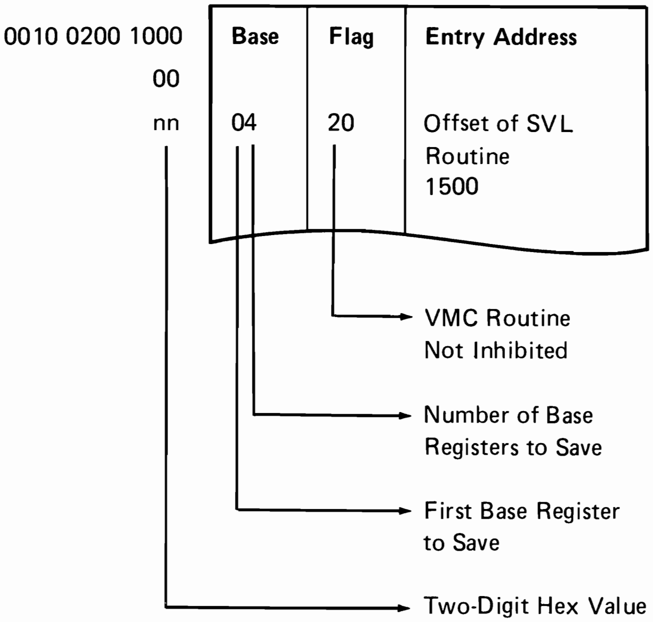
Machine: DFnn 3100 4200

	Before	After
B(0):	0000 2345 0800	0010 0200 1500[1]
B(1):	0101 0101 1110	0123 4567 0100
B(2):	0202 0202 2220	0246 8ACE 1200
B_1(3):	0123 4567 0000	0123 4567 0000
B_2(4):	0246 8ACE 1000	0246 8ACE 1000
IAR:	8800	1500

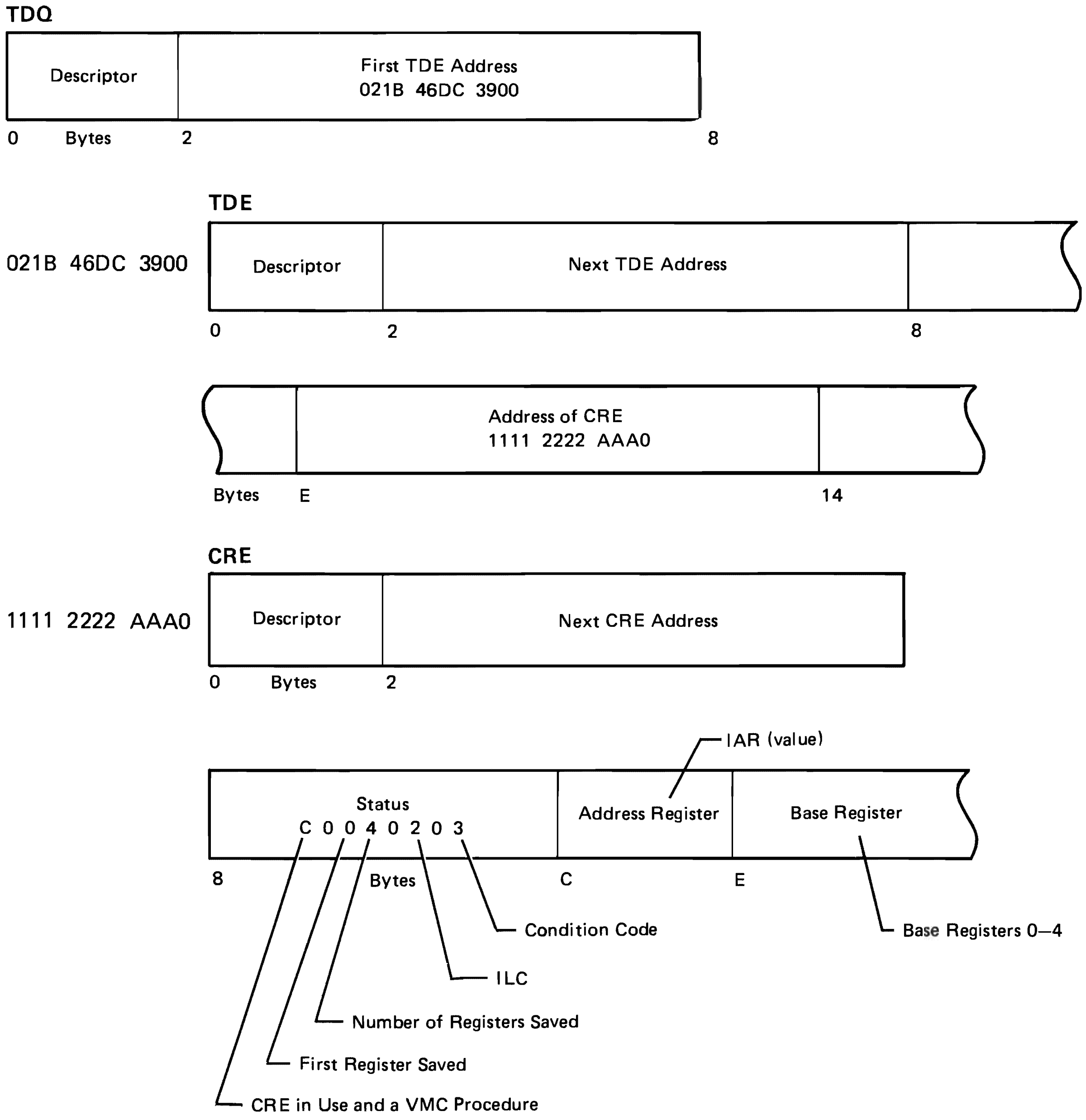
Control Address Table

100 0060	SVL Table Address 0010 0200 1000
68	ACQ Address

SVL Table


0010 0200 1000
00
nn
Base
Flag
Entry Address
04
20
Offset of SVL Routine 1500
VMC Routine Not Inhibited
Number of Base Registers to Save
First Base Register to Save
Two-Digit Hex Value

[1] SID of SVL table address.
Offset from SVL table entry.


TDQ
Descriptor
First TDE Address
021B 46DC 3900
0
Bytes
2
8
TDE
021B 46DC 3900
Descriptor
Next TDE Address
0
2
8
Address of CRE
1111 2222 AAA0
Bytes
E
14
CRE
1111 2222 AAA0
Descriptor
Next CRE Address
0
Bytes
2
IAR (value)
Status
C 0 0 4 0 2 0 3
Address Register
Base Register
8
Bytes
C
E
Condition Code
Base Registers 0—4
ILC
Number of Registers Saved
First Register Saved
CRE in Use and a VMC Procedure

SUPERVISOR LINK MONITORED (SVLM)

Instruction Description

This instruction provides for the conditional execution of an SVL (supervisor linkage) that is maskable for each task.

Format: RR

1F	I

0 Bits 8 15

Operation: When the SVL monitored flag (byte hex C, bit 7) is reset, this instruction acts as a no-operation. When the flag is set, the instruction executes as an SVLO using the second SVLM instruction byte as the SVL table index.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Descriptor access: Monitored ACQ descriptor
- Descriptor access: Monitored CRE descriptor
- Descriptor access: Monitored TDE descriptor (SVL receive wait)
- Specification

Programming Note: The processor recognizes a change in state of the SVL monitored flag bit only after the execution of a Dispatch Task Dispatching Queue instruction or after task switch. Thus, if a task sets or resets the bit and wants immediate action, it should issue a Dispatch Task Dispatching Queue instruction.

SVLM Example

Op	I
1F	xx

0 Bits 8 15

Assembler: SVLM I

Machine: IFxx

If bit 7 of byte hex C (TDE control mode) is set, this instruction executes as an SVLO. See the SVLO instruction for further information.

SUPERVISOR LINK SHORT (SVLO)

Instruction Description

The SVL (supervisor linkage) routine selected by the index in the I-field of the instruction is called using the supervisor link mechanism.

Format: RR

3F	I

0 Bits 8 15

Operation: The index is used to access an entry in the SVL table to determine where the SVL routine is located and how many registers are saved in an available CRE (call/return element) prior to branching to it.

An available CRE is found by searching the CRE list chained to the current TDE (task dispatching element). The current status (instruction address register, condition code, identification of the first base register stored, and the number of base registers stored) and the contents of the specified base registers are stored in the last available CRE on the list. If there are no available CREs on the list or if the list is empty, a CRE is implicitly received from the ACQ (available CRE queue) and is enqueued first on the TDE CRE list. The current status and base registers are then stored in that CRE. In either case, the status of the CRE obtained is set to *in use* (byte 8, bit 0 = 1).

If it is necessary to obtain a CRE from the ACQ and the ACQ is empty, the implicit receive is unsatisfied and the SVL instruction is nullified. The TDE of the current task is then dequeued from the TDQ (task dispatching queue) and enqueued to the ACQ wait list, and the task dispatcher is invoked.

Execution of the SVLO instruction may be halted due to an I/O interrupt. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point at which the interrupt was granted.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Descriptor access: Monitored ACQ descriptor
- Descriptor access: Monitored CRE descriptor
- Descriptor access: Monitored TDE descriptor (SVL receive wait)
- Specification

Programming Notes: An index value of zero is valid, but is also used by the processor to signal exceptions. If the task dispatcher is disabled and an SVLO instruction is attempted, a machine check will occur.

SVLO Example

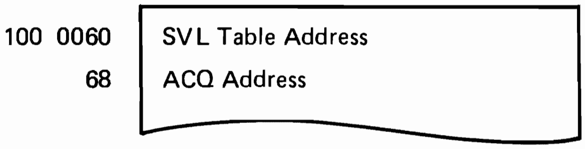
Op 3F	I nn

0 Bits 8 15

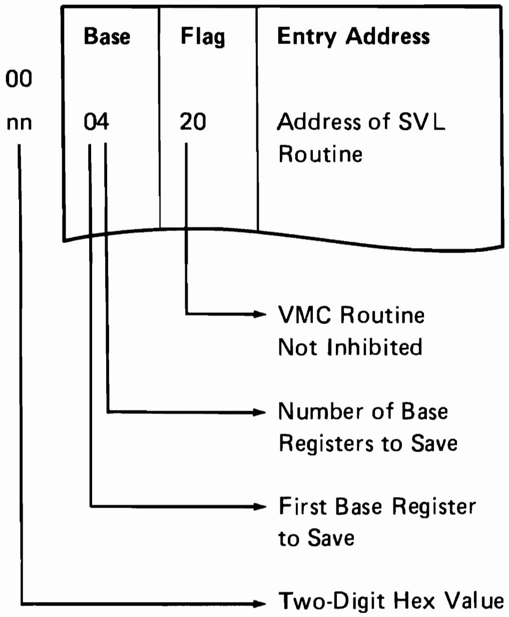
Assembler: SVLO I

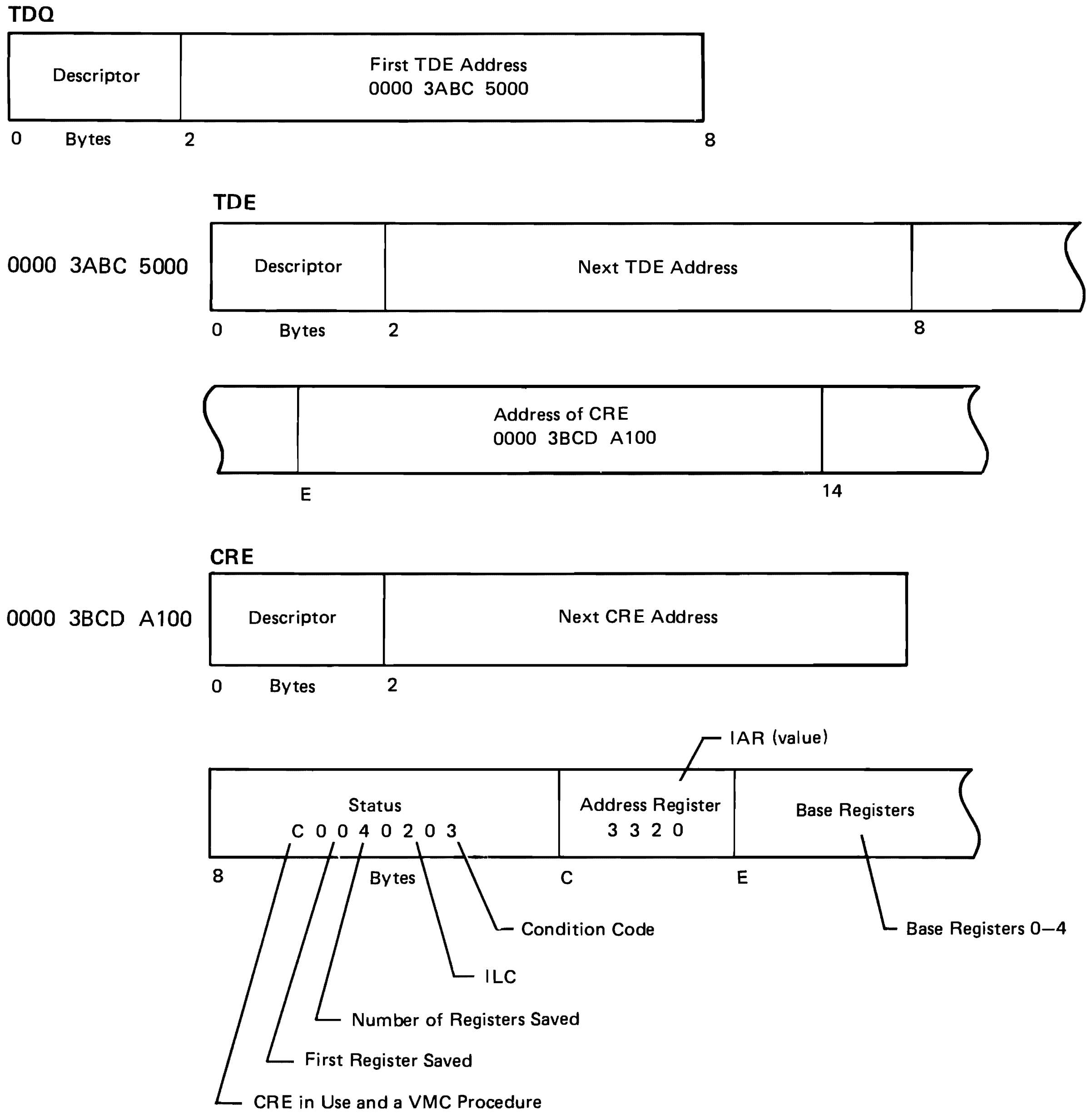
Machine: 3Fnn

Control Address Table



SVL Table


Base
Flag
Entry Address
00
nn
04
20
Address of SVL Routine
VMC Routine Not Inhibited
Number of Base Registers to Save
First Base Register to Save
Two-Digit Hex Value


TDQ
Descriptor
First TDE Address
0000 3ABC 5000
0
Bytes
2
8
TDE
0000 3ABC 5000
Descriptor
Next TDE Address
0
Bytes
2
8
Address of CRE
0000 3BCD A100
E
14
CRE
0000 3BCD A100
Descriptor
Next CRE Address
0
Bytes
2
IAR (value)
Status
C 0 0 4 0 2 0 3
Address Register
3 3 2 0
Base Registers
8
Bytes
C
E
Condition Code
Base Registers 0–4
ILC
Number of Registers Saved
First Register Saved
CRE in Use and a VMC Procedure

SUPERVISOR LINK SINGLE (SVL1)

Instruction Description

The SVL (supervisor linkage) routine selected by the index in the I-field of the instruction is called using the SVL mechanism.

Format: SI

5D	I_2	B_1	D_1
0 Bits	8	16	20 31

Operation: The index is used to access an entry in the SVL table to determine where the SVL routine is located and how many registers are to be saved in an available CRE (call/return element) prior to branching to the routine.

The effective address of the first operand is computed and checked for an effective address overflow exception. No attempt is made to access the first operand, and it remains unchanged in storage.

An available CRE is found by searching the CRE list chained to the current TDE (task dispatching element). The current status (instruction address register, condition code, identification of the first base register stored, and the number of base registers stored) and the contents of the specified base registers are stored in the last available CRE on the list. If there are no available CREs on the list or if the list is empty, a CRE is implicitly received from the ACQ (available call/return element queue) and is enqueued first on the TDE CRE list. The current status and base registers are then stored in that CRE. In either case, the status of the CRE obtained is set to *in use* (byte 8, bit 0 = 1).

If it is necessary to obtain a CRE from the ACQ and the ACQ is empty, the implicit recieve is unsatisfied and the SVL instruction is nullified. The TDE of the current task is then dequeued from the TDQ and enqueued to the ACQ wait list, and the task dispatcher is invoked.

After the registers specified in the SVL table are saved, the effective address of the first operand is placed in base register 1.

Execution of the SVL1 instruction may be interrupted due to I/O. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point at which the interrupt was granted.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Descriptor access: Monitored ACQ descriptor
- Descriptor access: Monitored CRE descriptor
- Descriptor access: Monitored TDE descriptor (SVL receive wait)
- Effective address overflow
- Specification

Programming Notes: An index value of zero is valid, but is also used by the processor to signal exceptions. If the task dispatcher is disabled and an SVL1 instruction is attempted, a machine check occurs.

SVL1 Example

Op	I_2	B_1	D_1
5D	nn	3	100

0 Bits 8 16 20 31

Assembler: SVL1 $D_1(B_1)$, I_2

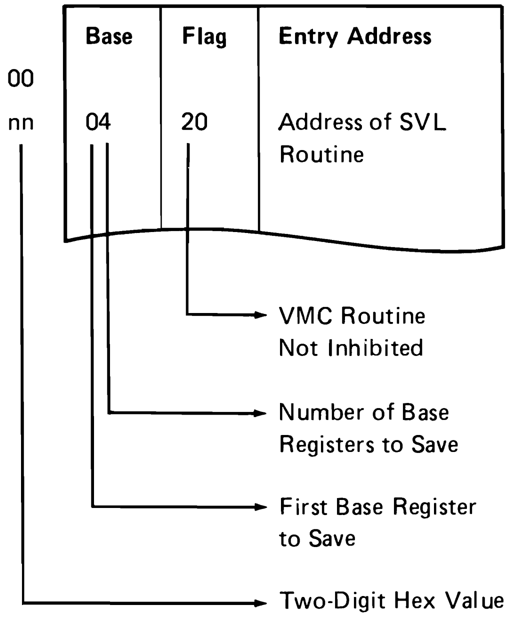
Machine: 5Dnn 3100

	Before			After		
B(0):	xxxx	xxxx	xxxx	1234	0000	1100
$B_1(3)$:	1234	0000	1000	1234	0000	1000
B(1):	xxxx	xxxx	xxxx	1234	0000	1100

Control Address Table

100 0060	SVL Table Address
68	ACQ Address

SVL Table



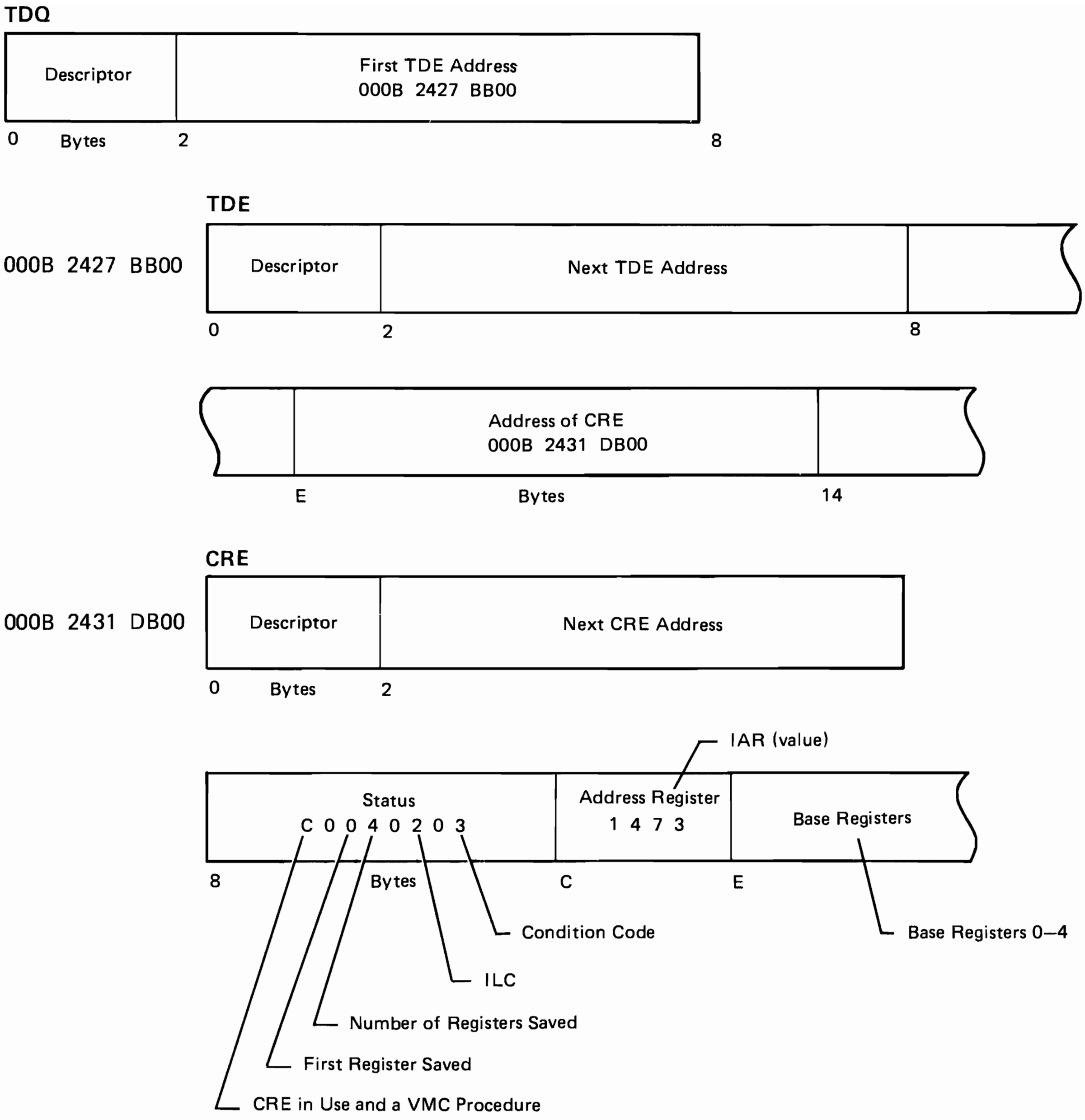
	Base	Flag	Entry Address
00			
nn	04	20	Address of SVL Routine

VMC Routine Not Inhibited

Number of Base Registers to Save

First Base Register to Save

Two-Digit Hex Value


TDQ
Descriptor
First TDE Address
000B 2427 BB00
0
Bytes
2
8
TDE
000B 2427 BB00
Descriptor
Next TDE Address
0
2
8
Address of CRE
000B 2431 DB00
E
Bytes
14
CRE
000B 2431 DB00
Descriptor
Next CRE Address
0
Bytes
2
IAR (value)
Status
C 0 0 4 0 2 0 3
Address Register
1 4 7 3
Base Registers
8
Bytes
C
E
Condition Code
Base Registers 0–4
ILC
Number of Registers Saved
First Register Saved
CRE in Use and a VMC Procedure

SUPERVISOR LINK SINGLE MONITORED (SVLM1)

Instruction Description

The SVL1 function occurs conditionally, depending on the value of a mask bit in the current TDE (task dispatching element).

Format: SI

5B	I_2	B_1	D_1

0 Bits 8 16 20 31

Operation: When the SVL1-monitored flag (byte 12, bit 3 of the TDE) is zero, the instruction acts as a no-operation. When the flag is one, an SVL1 occurs using the second SVLM1 instruction byte as the SVL table index. The index is used to access an entry in the SVL table to determine where the SVL routine is located and how many registers are to be saved in an available CRE (call/return element) prior to branching to the routine.

The effective address of the first operand is computed and checked for an effective address overflow exception. No attempt is made to access the first operand, and it remains unchanged in storage.

An available CRE is found by searching the CRE list chained to the current TDE (task dispatching element). The current status (instruction address register, condition code, identification of the first base register stored, and the number of base registers stored) and the contents of the specified base registers are stored in the last available CRE on the list. If there are no available CREs on the list or if the list is empty, a CRE is implicitly received from the ACQ (available call/return element queue) and is enqueued first on the TDE CRE list. The current status and base registers are then stored in that CRE. In either case, the status of the CRE obtained is set to *in use* (byte 8, bit 0 = 1).

If it is necessary to obtain a CRE from the ACQ and the ACQ is empty, the implicit recieve is unsatisfied and the SVL instruction is nullified. The TDE of the current task is then dequeued from the TDQ and enqueued to the ACQ wait list, and the task dispatcher is invoked.

After the registers specified in the SVL table are saved, the effective address of the first operand is placed in base register 1.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Descriptor access: Monitored ACQ descriptor
- Descriptor access: Monitored CRE descriptor
- Descriptor access: Monitored TDE descriptor (SVL receive wait)
- Effective address overflow
- Specification

Programming Notes: The processor recognizes a change in state of the bit only after execution of a Dispatch Task Dispatching Queue instruction or after a task switch. Thus, if a task sets the bit on or off and wants immediate action, it should execute a Dispatch Task Dispatching Queue instruction.

An index value of zero is valid, but is also used by the processor to signal exceptions.

If the task dispatcher is disabled and an SVLM1 instruction is attempted, a machine check occurs.

SVLM1 Example

Op	I_2	B_1	D_1
5B	nn	3	100

0 Bits 8 16 20 31

Assembler: SVLM1 $D_1(B_1)$, I_2

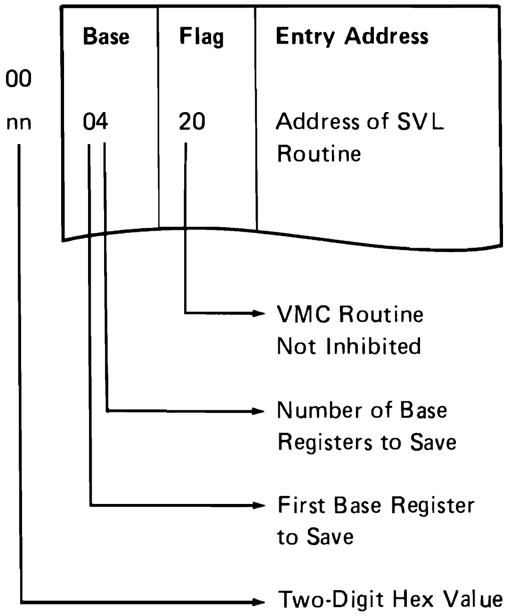
Machine: 5Bnn 3100

	Before			After		
B(0):	xxxx	xxxx	xxxx	1234	0000	1100
B_1(3):	1234	0000	1000	1234	0000	1000

Control Address Table

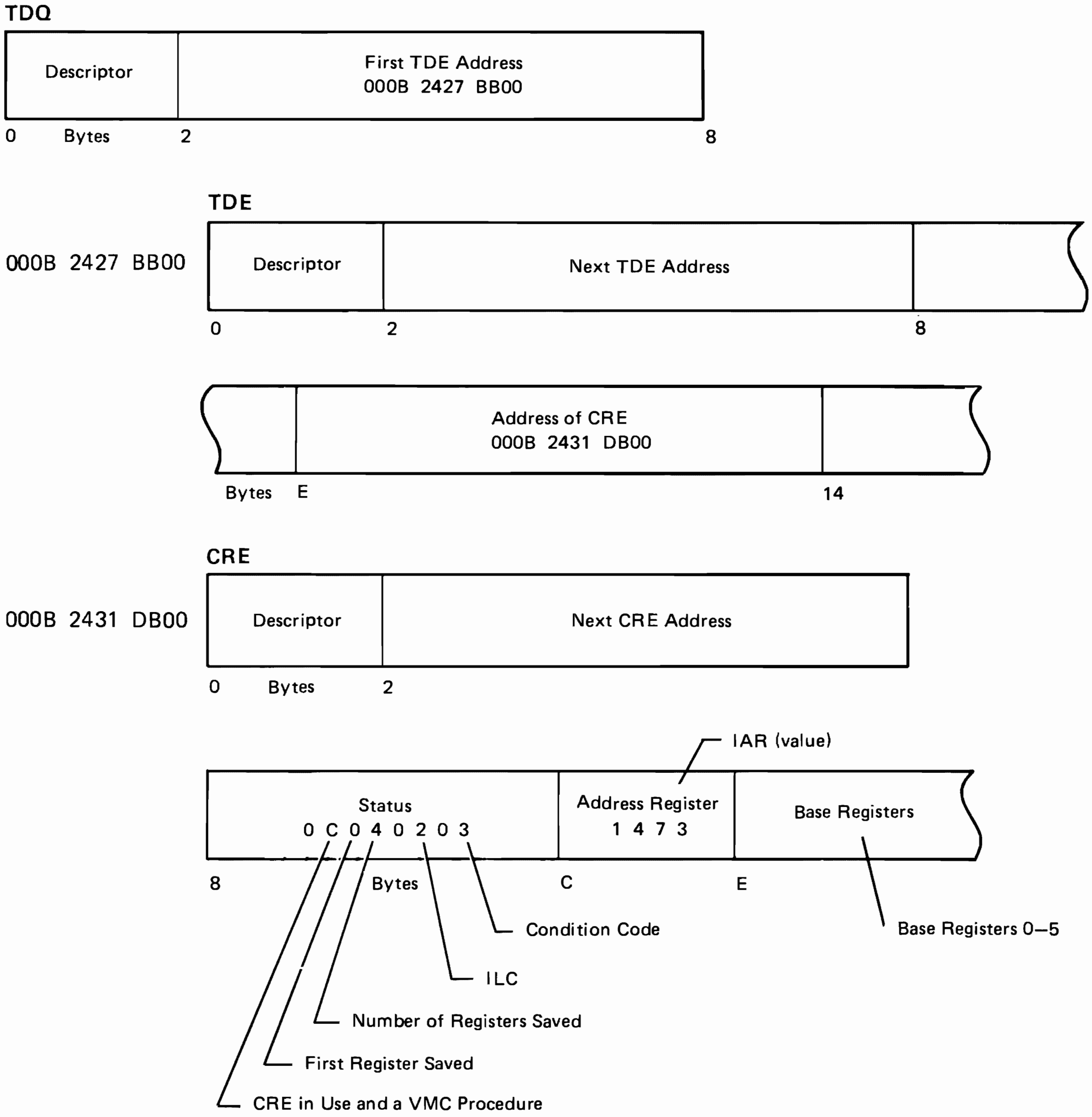
100 0060	SVL Table Address
68	ACQ Address

SVL Table



	Base	Flag	Entry Address
00			
nn	04	20	Address of SVL Routine

VMC Routine Not Inhibited
Number of Base Registers to Save
First Base Register to Save
Two-Digit Hex Value


TDQ
Descriptor
First TDE Address
000B 2427 BB00
0
Bytes
2
8
TDE
000B 2427 BB00
Descriptor
Next TDE Address
0
2
8
Address of CRE
000B 2431 DB00
Bytes
E
14
CRE
000B 2431 DB00
Descriptor
Next CRE Address
0
Bytes
2
IAR (value)
Status
0 C 0 4 0 2 0 3
Address Register
1 4 7 3
Base Registers
8
Bytes
C
E
Condition Code
Base Registers 0–5
ILC
Number of Registers Saved
First Register Saved
CRE in Use and a VMC Procedure

TERMINATE IMMEDIATELY (TI)

Instruction Description

This instruction causes termination of processing.

Format: RR

0D	R_1	4

0 Bits 8 12 15

Operation: The operand register contains the bit patterns used by the SCA to activate the light-emitting diodes on the CE panel. The TI (terminate immediately) instruction causes a machine check when issued if the machine is not already in the machine check mode. If already in machine check mode, the processor enters check stop state.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

TI Example

Op	R_1	E
0D	3	4

0 Bits 8 12 15

The codes displayed in CE panel light-emitting diodes are related to the first-operand halfword as follows:

CE Panel			
First-Operand Halfword Hex Value	**Sequence # 0-7**	**Indicator 8-15**	**Machine Check Light-Emitting Diodes**
00xx-06xx	08	FF	On
07xx	07	xx	On
08xx	08	xx	On
09xx	09	xx	Off
0Axx-FFxx	08	FF	On

TEST UNDER MASK BYTE IMMEDIATE (TMBI)

Instruction Description

The states of the selected first-operand bits are used to set the condition code.

Format: SI

9D	I_1	B_1	D_1

0 Bits 8 16 20 31

Operation: The second operand is used as an 8-bit mask that corresponds one-for-one with the bits of the first operand. A set mask bit indicates that the first-operand bit is to be tested. When the mask bit is reset, the first-operand bit is ignored. When all bits thus selected are reset, condition code zero is set. Condition code zero is also set when the mask bits are zeros. When the selected bits are ones, the code is set to 3; otherwise the code is set to 1.

Overflow and Sign Code: Not applicable.

Condition Code:

0 Selected bits are zeros, or
 the mask bits are zeros
1 Selected bits are mixed zeros and ones
2 --
3 Selected bits are ones

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

TMBI Example

Op	I_2	B_1	D_1
9D	A0	3	000

0 Bits 8 16 20 31

Assembler: TMBI $D_1(B_1)$, I_2

Machine: 9DA0 3000

B_1(3): 0001 2345 1000

Storage — Before and After

	0/8	2/A	4/C	6/E
0001 2345 1000	20			

	Before	After
Condition Code:	x	1

TEST UNDER MASK BYTE IMMEDIATE AND BRANCH IF ONES (TMBIBO)

Instruction Description

The states of the selected first-operand bits are used to determine if a branch will be taken.

Format: SI

E1	I_2	B_1	D_1	D_3

0 Bits 8 16 20 32 47

Operation: The second operand is used as an 8-bit mask that corresponds one-for-one with the bits of the first operand. A set mask bit indicates that the first-operand bit is to be tested. When the mask bit is reset, the first-operand bit is ignored. When all bits thus selected are set, the branch is taken. When the mask is all zeros, the branch is not taken.

When a branch is taken, the updated instruction address is replaced by the sum of the 16-bit displacement (D_3) and the offset portion of the instruction stream base address contained in base register zero.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

TMBIBO Example

Op	I_2	B_1	D_1	D_3
E1	02	3	000	005E

0 Bits 8 16 20 32 47

Assembler: TMBIBO $D_1(B_1), D_3, I_2$

Machine: E102 3000 005E

$B_1(3)$: 0001 2562 1000

Storage — Before and After

	0/8	2/A	4/C	6/E
0001 2562 1000	F2			

	Before	After
B_0	5425 3111 0122	5425 3111 0122
IAR	0150	0180

TEST UNDER MASK BYTE IMMEDIATE AND BRANCH IF ZEROS (TMBIBZ)

Instruction Description

The states of the selected first-operand bits are used to determine if a branch will be taken.

Format: SI

E0	I_2	B_1	D_1	D_3

0 Bits 8 16 20 32 47

Operation: The second operand is used as an 8-bit mask that corresponds one-for-one with the bits, of the first operand. A set mask bit indicates that the first-operand bit is to be tested. When the mask bit is reset, the first-operand bit is ignored. When all bits thus selected are reset, the branch is taken. When the mask is all zeros, the branch is taken.

When a branch is taken, the updated instruction address is replaced by the sum of the 16-bit displacement (D_3) and the offset portion of the instruction stream base address contained in base register zero.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

TMBIBZ Example

Op	I_2	B_1	D_1	D_3
E0	A0	3	000	IFC2

0 Bits 8 16 20 32 47

Assembler: TMBIBZ $D_1(B_1)$, D_3, I_2

Machine: E0A0 3000 1FC2

$B_1(3)$: 0001 2345 1000

Storage — Before and After

	0/8	2/A	4/C	6/E
0001 2345 1000	1A00			

	Before	After
B_0	5432 3210 0020	5432 3210 0020
IAR	0130	1FE2

TEST UNDER MASK BYTE REGISTER (TMBR)

Instruction Description

The states of the selected first-operand bits are used to set the condition code.

Format: RR

13	r_1	r_2

0 Bits 8 12 15

Operation: The second operand is used as an 8-bit mask that corresponds one-for-one with the bits of the first operand.

A set mask bit indicates that the first-operand bit is to be tested. When the mask bit is reset, the first-operand bit is ignored. When all bits thus selected are zero, condition code zero is set. Condition code zero is also set when the mask bits are zeros. When the selected bits are ones, the code is set to 3; otherwise the code is set to 1.

Overflow and Sign Code: Not applicable.

Condition Code:

0	Selected bits are zeros or the mask bits are zeros
1	Selected bits are mixed zeros and ones
2	--
3	Selected bits are ones

Carry: Not applicable.

Boundary Requirements and Program Exceptions: None.

TMBR Example

Op	r_1	r_2
13	3	4

0 Bits 8 12 15

Assembler: TMBR r_1, r_2

Machine: 1334

	Before	After
r_1 (3):	20	20
r_2 (4):	A0	A0
Condition Code:	x	1

TRANSLATE (TR)

Instruction Description

The 8-bit bytes addressed by the first operand are used as arguments to refer to the list of function bytes addressed by the second-operand address. Each 8-bit function byte selected from the list replaces the corresponding argument byte in the first operand.

Format: SS

CC	L	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32	36 47

Operation: The bytes of the first operand are selected one by one for translation, proceeding left to right. Each argument byte is added to the initial second-operand address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an 8-bit unsigned integer and extended to the left with zeros. The sum is used as the address of the function byte, which then replaces the original argument byte. The operation proceeds until the entire first-operand field is translated. The second operand is unchanged by the operation unless the operands overlap in storage.

Execution of the TR instruction may be interrupted due to I/O. If an I/O interrupt does occur, the interrupt will be processed, and instruction processing will resume at the point at which the interrupt was granted.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands may overlap in storage.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

Note: The L-field applies only to the first operand.

TR Example

Op CC	L_1 05	B_1 3	D_1 2C0	B_2 4	D_2 100
0 Bits	8	16	20	32	36 47

Assembler: TR $D_1(L_1, B_1), D_2(B_2)$

Machine: CC05 32C0 4100

B_1(3): 010A B12C 3000

B_2(4): 010A C34D 2000

Storage — Before

	0/8	2/A	4/C	6/E
010A B12C 32C0	F3F1	F4F1	F5F9	
010A C34D 2100				
F0	3031	3233	3435	3637
	3839			

Storage — After

	0/8	2/A	4/C	6/E
010A B12C 32C0	3331	3431	3539	
010A C34D 2100				
F0	3031	3233	3435	3637
	3839			

This page is intentionally left blank.

TRANSLATE AND TEST (TRT)

Instruction Description

The 8-bit bytes addressed by the first-operand are used as arguments to refer to the list of function bytes addressed by the second-operand address.

Format: SS

CD	L	B_1	D_1	B_2	D_2
0 Bits	8	16	20	32	36 47

Operation: Each function byte selected from the list is used to determine the continuation of the operation. When the function byte is a zero (that is, hexadecimal 00), the operation proceeds by fetching and translating the next argument byte. When the function byte is nonzero, the operation is completed by inserting the related argument address into the base register specified by B_1 and then inserting the function byte in byte register hex F.

The bytes of the first operand are selected one by one for translation, proceeding from left to right. The first and second operands remain unchanged in storage. Fetching of the function byte from the list is performed as in the Translate instruction. The function byte retrieved from the list is inspected for an all-zero combination.

When all the first-operand field is translated before a nonzero function byte is encountered, the operation is completed by setting condition code zero. The contents of the base register specified by B_1 and byte register hex F remain unchanged.

Condition code 1 is set when one or more argument bytes have not been translated. Condition code 2 is set if the last selected function byte is nonzero.

Overflow and Sign Code: Not applicable.

Condition Code:

0	All selected function bytes = 0
1	Function byte selected ≠ 0 (before the first-operand field is translated)
2	Last selected function byte ≠ 0
3	--

Carry: Not applicable.

Boundary Requirements: None.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

Note: The L-field applies only to the first operand.

TRT Example

Op	L_1	B_1	D_1	B_2	D_2
CD	05	3	000	4	000

0 Bits 8 16 20 32 36 47

Assembler: TRT $D_1(L_1, B_1), D_2(B_2)$

Machine: CD05 3000 4000

	Before	After
B_1(3):	010A B12C 3000	010A B12C 3000
B_2(4):	010A C34D 2000	010A C34D 2000
r(15):	xx	33
Condition Code:	x	1

Storage — Before and After

	0/8	2/A	4/C	6/E
010A B12C 3000	F3F1	F4F1	F5F9	
010A C34D 2000				
F0	3031	3233	3435	3637
	3839			

TRANSLATE REGISTER (TRR)

Instruction Description

The 8-bit bytes addressed by the first operand are used as arguments to refer to the list of function bytes addressed by the second-operand address. Each 8-bit function byte selected from the list replaces the corresponding argument byte in the first operand.

Format: SS

BE	r_3	0	B	D	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The bytes of the first operand are selected one by one for translation, proceeding left to right. Each argument byte is added to the initial second-operand address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an 8-bit unsigned integer and extended to the left with zeros. The sum is used as the address of the function byte, which then replaces the original argument byte. The operation proceeds until the entire first-operand field is translated. The second operand is unchanged by the operation unless the operands overlap in storage.

This instruction is identical to the TR instruction except the length is specified in a byte register.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: The operands may overlap in storage.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow

Note: The length specified in the byte register applies only to the first operand.

TRR Example

Op BE	r_3 8	E 0	B_1 3	D_1 2C0	B_2 4	D_2 100

0 Bits 8 12 16 20 32 36 47

Assembler: TRR $D_1(B_1)$, $D_2(B_2)$, r_3

Machine: BE80 32C0 4100

B_1(3): 010A B12C 3000

B_2(4): 010A C34D 2000

r_3(8): 05

Storage — Before

	0/8	2/A	4/C	6/E
010A B12C 32C0	F3F1	F4F1	F5F9	
010A C34D 2100				
F0	3031	3233	3435	3637
	3839			

Storage — After

	0/8	2/A	4/C	6/E
010A B12C 32C0	3331	3431	3539	
010A C34D 2100				
F0	3031	3233	3435	3637
	3839			

TRIM (TRIM)

Instruction Description

The trimmed length of the character string located by operand 1 is returned in halfword register 15 (R15), operand 3.

Format: SI

72	I_2	B_1	D_1

0 Bits 8 16 20 31

Operation: Operand 3 initially contains the untrimmed length of the string. A negative value in operand 3 causes a specification exception. The character value to be trimmed is specified by operand 2.

The operation proceeds as follows:

1. If operand 3 is zero, the instruction is complete.
2. The character, located by adding the effective address of operand 1 to operand 3 and then decrementing by one, is compared to I_2. If the compare is not equal, the instruction is complete.
3. Operand 3 is decremented by one. Then go to step 1.

If B_1 is 15, unpredictable results can occur.

The instruction can be interrupted at any time.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Not applicable.

Program Exceptions:

- Address translation
- Addressing
- Effective address overflow
- Specification

TRIM Example

Op	I_2	B_1	D_1
72	40	3	2C0

0 Bits 8 16 20 31

Assembler: TRIM $D_1(B_1)$, I_2

Machine: 7240 32C0

	Before	After
$B_1(3)$:	101A B12C 3000	101A B12C 3000
I_2:	40	40
$r_3(15)$:	08	04

Storage — Before and After

	0/8	2/A	4/C	6/E
101A B12C 32C0	1020	40FE	4040	4040

UNSTACK (UNSTK)

Instruction Description

The current stack entry is released and the address of the previous stack is loaded into the first operand.

The UNSTK instruction is used to control the deallocation of storage. The storage area is organized like a stack. Every storage allocation starts on a 16-byte boundary. The maximum size of a storage allocation is limited to 64 K-16 bytes.

Format: RR

2B	B_1	0
0 Bits	8	12 15

Operation: The first operand initially contains the address of the current stack entry. The first 8 bytes of the current stack entry contain 4 halfword fields that are used by the Stack and Unstack instructions. The first 2 halfwords are not used by the Unstack instruction. The third halfword contains an offset value which, when used with the SID (segment identifier) portion of the current stack entry address, forms the address of the first byte of the previous stack entry. The fourth halfword is the flag field.

Bit position 15 of the fourth halfword in the current stack entry (for example, the flag field) is checked. If bit 15 is set, a stack exception occurs and the operation is suppressed. If bit 15 is reset, the contents of the third halfword of the current stack entry are loaded into the offset portion of the first-operand register, thus making the previous entry the new current entry.

Overflow and Sign Code: Not applicable.

Condition Code: Not changed.

Carry: Not applicable.

Boundary Requirements: Initially the first operand must start on a doubleword boundary; otherwise a specification exception occurs and the operation is suppressed.

Program Exceptions:

- Address translation
- Addressing
- Specification
- Stack

Programming Note: The new current entry is not checked for doubleword alignment.

UNSTK Example

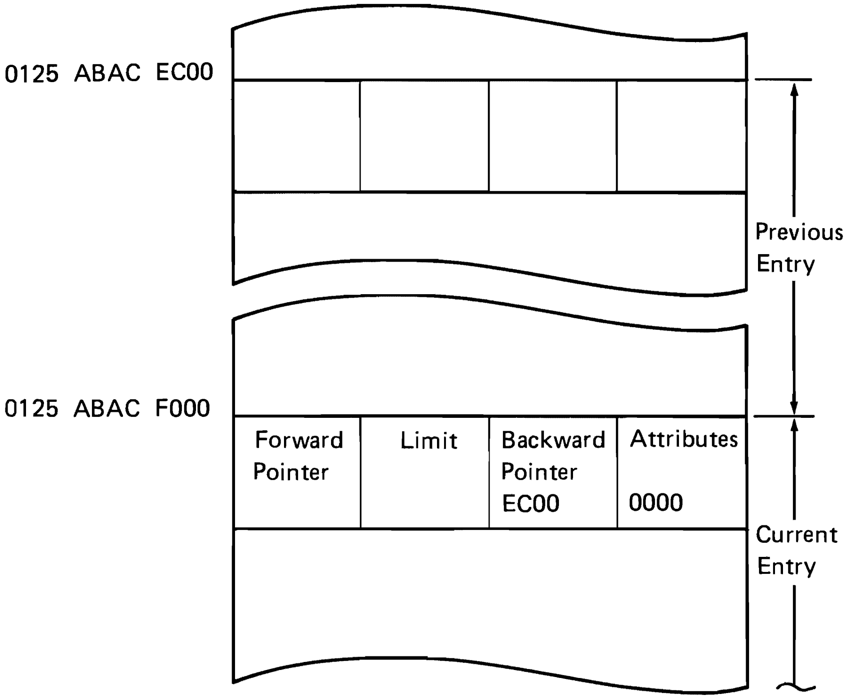
Op 2B	B_1 3	E 0

0 Bits 8 12 15

Assembler: UNSTK B_1

Machine: 2B30

	Before	After
B_1(3):	0125 ABAC F000	0125 ABAC EC00



ZERO AND ADD CHARACTERS (ZAC)

Instruction Description

The second operand is placed in the first-operand location.

Format: SS

C6	L_1	L_2	B_1	D_1	B_2	D_2

0 Bits 8 12 16 20 32 36 47

Operation: The operation is equivalent to an addition to zero with both operands treated as signed binary quantities. If the second operand is shorter than the first operand, the second operand is considered to be extended to the left with bits equal to the sign bit. If the first operand is too short to contain all significant bits of the second operand, an overflow occurs and only the rightmost bits of the second operand are placed in the first-operand location.

Overflow: Not applicable.

Sign Code: The sign of the result is unpredictable when significant bits are lost.

Condition Code: If an overflow occurs, the code indicates the sign the result would have if the first operand was long enough to contain all significant bits of the result.

0	Result	=	0
1	Result	<	0
2	Result	>	0
3	--		

Carry: Not applicable.

Boundary Requirements: The operands can overlap in storage if the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand; otherwise the overlap is destructive and the results are unpredictable.

Program Exceptions:

- Address translation
- Addressing
- Binary overflow
- Effective address overflow

ZAC Example

Op C6	L_1 7	L_2 5	B_1 3	D_1 100	B_2 4	D_2 200

0 Bits 8 12 16 20 32 36 47

Assembler: ZAC $D_1(L_1, B_1), D_2(L_2, B_2)$

Machine: C675 3100 4200

B_1(3): 2102 6100 A000

B_2(4): 5718 9420 B000

Storage — Before

	0/8	2/A	4/C	6/E
2102 6100 A000	xxxx	xxxx	xxxx	xxxx
5718 9420 B000	114A	118B	240E	

Storage — After

	0/8	2/A	4/C	6/E
2102 6100 A000	0000	114A	118B	240E
5718 9420 B000	114A	118B	240E	

	Before	**After**
Condition Code:	x	2

Appendix A. Teleprocessing Instruction Flow Charts

These diagrams have been removed from this document. See the *IBM System/38 Communications Operation Charts,* SY31-0911, for diagrams showing the relationship between segments of the System/38.

Appendix B. Operation Code Assignments

Operation Code (Second Digit)

	2-Byte Instructions				**4-Byte Instructions**						**6-Byte Instructions**					
	Operation Code (First Digit)															
↓	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	Note 2	ALBR	AHR	ALHR	Note 2	AHRI	ALHRI		Note 1	ALH	AHI	ALHI	AC	CBIBE	TMBIBZ	AP
1	SLL	SLBR	SHR	SLHR			SLHCT	Note 1		Note 1			SC	CBIBN	TMBIBO	SP
2	SRL	CLBR	CHR	CLHR	CLBRI	CHRI	CLHRI	TRIM		CLH	CHI	CLHI	CC			CP
3	SLA	TMBR	CLAR		ALBRI	LA	EDPD		Note 1	LSOP			ALC			MP
4	SRA	LBR	LHR		LBRI	LHRI	LVT	LB	LH	L	EXTAG	MVAST	SLC			DP
5		LR	LPDEAR				STST	LMB	LMH	LM	INTAG	MVCAT	CLC		CVPZC	CVPZ
6				SCB	GHRF	GHR	SENDC	STB	STH	ST			ZAC	CVZPB	CVZPC	CVZP
7					FHRF	FHR	RECC	STMB	STMH	STM						CVPB
8		NBR	NHR		NBRI	NHRI	SENDM			NBI	FNC2		NC	HVVA	PPR[3]	CVBP
9		OBR	OHR		OBRI	OHRI	SENDMW	Note 1		OBI			OC	RECM	CLCR	MPL[3]
A		XBR	XHR		XBRI	Note 1	SETIT			XBI		MVNN	XC	DQM	CLCL	DPL[3]
B		STACK	UNSTK		SACM	SVLM1				MVBI	MVHI	MVNZ	MVC	MVCR	MVCL	MVPS
C					CAL	CALHI	EQM		STIT	CLBI	MVBIP	MVZN	TR	MHS	MWS	MVPS Z
D	Note 1	MVMC			CALH	SVL1	Note 1	CSH		TMBI	CSA	MVZZ	TRT	DHS	DWS	
E		BI	BR	SVX	JC	JBN	BC	EX	BCT	BCN	Note 1	Note 1	Note 1			
F		SVLM	BRL	SVLO	BAL	JBF	BU	BCNX	BALL	ALHBL	CSACH	CSAC	SCAN	SVL2	CALLI	Note 2

Notes:

1. Extended operation code instruction; see Figure B-2 for assignments.
2. The operation code is reserved and treated as invalid.

[3] Implicit SVL instructions

Operation Code Extenders

	2-Byte Instructions	4-Byte Instructions							6-Byte Instructions		
	Operation Codes										
	0D	**5A**	**6D**	**71**	**79**	**80**	**83**	**91**	**AE**	**BE**	**CE**
0	ETD	XHRI	EQTDE	SLB	OB	AH	RRCRR	SLH	CSF	AHSPOI	CLF
1	DTD	SETIND	DQTDE	ALB	NB	SH	LPDEA		ASF	AHSPO	ALF
2	RMCM		SETCC	CLB	XB	CH	LHTEA		SSF	AFSPO	SLF
3	RACM		STCC			OH	EPDE		MSF	TRR	MLF
4	TI		SETTOD			NH	RPDE		DSF	CVTCM	DLF
5	RAHR		STTOD			XH	IPDE			CVTMC	
6	RCB		DTDQ							CVTCS	
7			DIAG						CVSLF[1]	CVTSC	CVLSF[1]
8									CVSFB	SSCA[1]	CVLFB
9							STSOP		CVBSF		CVBLF
A									CVSFPD[1]		CVLFPD[1]
B									CVPDSF[1]		CVPDLF[1]
C									CVSFDF[1]		CVLFDF[1]
D									CVDFSF[1]		CVDFLF[1]
E	(see note)								CSFMF1[1]		CLFMF1[1]
F	(see note)								CSFMF2[1]		CLFMF2[1]

Note: These operation codes are reserved for development testing and will yield unpredictable results if executed.

[1] Implicit SVL instructions

Appendix C. Instruction Index

Mnemonic	Instruction	Format	Operation Code	Extender	Page
AC	Add Characters	SS	C0	-	10-2
AFSPO	Add Fullword Space Pointer Offset	SS	BE	2	10-4
AH	Add Halfword	RS	80	0	10-6
AHI	Add Halfword Immediate	SI	A0	-	10-7
AHR	Add Halfword Register	RR	20	-	10-8
AHRI	Add Halfword Register Immediate	RI	50	-	10-9
AHSPO	Add Halfword Space Pointer Offset	SS	BE	1	10-10
AHSPOI	Add Halfword Space Pointer Offset Immediate	SI	BE	0	10-12
ALB	Add Logical Byte	RS	71	1	10-14
ALBR	Add Logical Byte Register	RR	10	-	10-15
ALBRI	Add Logical Byte Register Immediate	RI	43	-	10-16
ALC	Add Logical Characters	SS	C3	-	10-17
ALF	Add Long Float	SS	CE	1	10-25
ALH	Add Logical Halfword	RS	90	-	10-18
ALHBL	Add Logical Halfword and Branch On Limit	RS	9F	-	10-20
ALHI	Add Logical Halfword Immediate	SI	B0	-	10-22
ALHR	Add Logical Halfword Register	RR	30	-	10-23
ALHRI	Add Logical Halfword Register Immediate	RI	60	-	10-24
AP	Add Packed	SS	F0	-	10-28
ASF	Add Short Float	SS	AE	1	10-29
BAL	Branch and Link	RI	4F	-	10-40
BALL	Branch and Link Long	RS	8F	-	10-41
BC	Branch on Condition	RI	6E	-	10-43

Mnemonic	Instruction	Format	Operation Code	Extender	Page
BCN	Branch on Condition Indirect	RS	9E	-	10-44
BCNX	Branch on Condition Indirect Indexed	RS	7F	-	10-45
BCT	Branch on Count	RI	8E	-	10-46
BI	Branch Internal	RR	1E	-	10-42
BR	Branch Register	RR	2E	-	10-47
BRL	Branch Register Long	RR	2F	-	10-48
BU	Branch Unconditional	RI	6F	-	10-49
CAL	Compute Address Long	RS	4C	-	10-80
CALH	Compute Address Long Halfword	RS	4D	-	10-82
CALHI	Compute Address Long Halfword Immediate	RI	5C	-	10-82.1
CALLI	Call Internal	SI	EF	-	10-50
CBIBE	Compare Byte Immediate and Branch Equal	SI	D0	-	10-54
CBIBN	Compare Byte Immediate and Branch Not Equal	SI	D1	-	10-54.1
CC	Compare Characters	SS	C2	-	10-54.2
CH	Compare Halfword	RS	80	2	10-55
CHI	Compare Halfword Immediate	SI	A2	-	10-56
CHR	Compare Halfword Register	RR	22	-	10-57
CHRI	Compare Halfword Register Immediate	RI	52	-	10-58
CLAR	Compare Logical Address Register	RR	23	-	10-59
CLB	Compare Logical Byte	RS	71	2	10-60
CLBI	Compare Logical Byte Immediate	SI	9C	-	10-61
CLBR	Compare Logical Byte Register	RR	12	-	10-62
CLBRI	Compare Logical Byte Register Immediate	RI	42	-	10-63
CLC	Compare Logical Characters	SS	C5	-	10-64

Mnemonic	Instruction	Format	Operation Code	Extender	Page
CLCL	Compare Logical Characters Long	SS	EA	-	10-66
CLCR	Compare Long Character Register	SS	E9	-	10-65
CLF	Compare Long Float	SS	CE	0	10-72
CLFMF1	Compute Long Float Math Function Using One Input Value	SS	CE	E	10-84
CLFMF2	Compute Long Float Math Function Using Two Input Values	SS	CE	F	10-87
CLH	Compare Logical Halfword	RS	92	-	10-68
CLHI	Compare Logical Halfword Immediate	SI	B2	-	10-69
CLHR	Compare Logical Halfword Register	RR	32	-	10-70
CLHRI	Compare Logical Halfword Register Immediate	RI	62	-	10-71
CP	Compare Packed	SS	F2	-	10-75
CSA	Compute Subscript Address	SI	AD	-	10-94
CSAC	Compute Subscript Address Constrained	SS	BF	-	10-96
CSACH	Compute Subscript Address Constrained Halfword	SS	AF	-	10-98
CSF	Compare Short Float	SS	AE	0	10-76
CSFMF1	Compute Short Float Math Function Using One Input Value	SS	AE	E	10-90
CSFMF2	Compute Short Float Math Function Using Two Input Values	SS	AE	F	10-92
CSH	Compare and Swap Halfword	RS	7D	-	10-52
CVBLF	Convert Binary to Long Float	SS	CE	9	10-100
CVBP	Convert Binary to Packed	SS	F8	-	10-101

Mnemonic	Instruction	Format	Operation Code	Extender	Page
CVBSF	Convert Binary to Short Float	SS	AE	9	10-102
CVDFLF	Convert Decimal Form to Long Float	SS	CE	D	10-118
CVDFSF	Convert Decimal Form to Short Float	SS	AE	D	10-120
CVLFB	Convert Long Float to Binary	SS	CE	8	10-122
CVLFDF	Convert Long to Decimal Form	SS	CE	C	10-124
CVLFPD	Convert Long Float to Packed Decimal	SS	CE	A	10-127
CVLSF	Convert Long to Short Float	SS	CE	7	10-130
CVPB	Convert Packed to Binary	SS	F7	-	10-140
CVPDLF	Convert Packed Decimal to Long Float	SS	CE	B	10-136
CVPDSF	Convert Packed Decimal to Short Float	SS	AE	B	10-138
CVPZ	Convert Packed to Zone	SS	FS		10-141
CVPZC	Convert Packed to Zoned with Data Checking	SS	E5	-	10-142
CVSFB	Convert Short Float to Binary	SS	AE	8	10-142
CVSFDF	Convert Short Float to Decimal Form	SS	AE	C	10-144
CVSFPD	Convert Short Float to Packed Decimal	SS	AE	A	10-147
CVSLF	Convert Short to Long Float	SS	AE	7	10-150
CVTCM	Convert Characters to MULTI-LEAVING Remote Job Entry	SS	BE	4 10-112	
CVTCS	Convert Characters to SNA	SS	BE	6	10-103
CVTMC	Convert MULTI-LEAVING Remote Job Entry to Character	SS	5	-	10-132

Mnemonic	Instruction	Format	Operation Code	Extender	Page
CVTSC	Convert SNA to Characters	SS	BE	7	10-152
CVZP	Convert Zoned to Packed	SS	F6	-	10-162
CVZPB	Convert Zoned to Packed with Data Checking and Blank Conversion	SS	D6	-	10-162.1
CVZPC	Convert Zoned to Packed with Data Checking	SS	__	E6	10-162.2
DHS	Divide Halfword Storage	SS	DD	-	10-171
DIAG	Diagnose	SI	6D	7	10-168
DLF	Divide Long Float	SS	CE	4	10-172
DP	Divide Packed	SS	F4	-	10-176
DPL	Divide Packed Long	SS	FA	-	10-178
DQM	Dequeue Message	SS	DA	-	10-164
DQTDE	Dequeue Task Dispatching Element	RS	6D	1	10-166
DSF	Divide Short Float	SS	AE	4	10-180
DTD	Disable Task Dispatching	RR	0D	1	10-169
DTDQ	Dispatch Task Dispatching Queue	SI	6D	6	10-170
DWS	Divide Word Storage	SS	ED	-	10-183
EDPD	Edit Packed Decimal	RS	63	-	10-184
EPDE	Examine Primary Directory Entry	SI	83	3	10-198
EQM	Enqueue Message	RS	6C	-	10-194
EQTDE	Enqueue Task Dispatching Element	RS	6D	0	10-196
ETD	Enable Task Dispatching	RR	0D	0	10-192
EX	Execute	RS	7E	-	10-210
EXTAG	Extract Tags	SS	A4	-	10-212
FHR	Free Hold Record	RS	57	-	10-214
FHRF	Free Hold Record First	RS	47	-	10-216
FNC2	Function Call Double	SS	A8	-	10-223
GHR	Grant Hold Record	RS	56	-	10-228
GHRF	Grant Hold Record First	RS	46	-	10-230

Mnemonic	Instruction	Format	Operation Code	Extender	Page
HVVA	Hash and Verify Virtual Address	SS	D8	-	10-237
INTAG	Insert Tags	SS	A5	-	10-240
IPDE	Invalidate Primary Directory Entry	SI	83	5	10-242
JBF	Jump on Bits Off	RI	5F	-	10-245
JBN	Jump on Bits On	RI	5E	-	10-246
JC	Jump on Condition	RI	4E	-	10-248
L	Load	RS	94	-	10-250
LA	Load Address	RS	53	-	10-251
LB	Load Byte	RS	74	-	10-255
LBR	Load Byte Register	RR	14	-	10-256
LBRI	Load Byte Register Immediate	RI	44	-	10-257
LH	Load Halfword	RS	84	-	10-258
LHR	Load Halfword Register	RR	24	-	10-259
LHRI	Load Halfword Register Immediate	RI	54	-	10-260
LHTEA	Load Hash Table Entry Address	RS	83	2	10-261
LM	Load Multiple	RS	95	-	10-262
LMB	Load Multiple Byte	RS	75	-	10-263
LMH	Load Multiple Halfword	RS	85	-	10-264
LPDEA	Load Primary Directory Entry Address	RS	83	1	10-265
LPDEAR	Load Primary Directory Entry Address Register	RR	25	-	10-266
LR	Load Register	RR	15	-	10-267
LSOP	Load Space Offset Pointer	RS	93	-	10-268
LVT	Load and Verify Tags	RS	64	-	10-252

Mnemonic	Instruction	Format	Operation Code	Extender	Page
MHS	Multiply Halfword Storage	SS	DC	-	10-286
MLF	Multiply Long Float	SS	CE	3	10-287
MP	Multiply Packed	SS	F3	-	10-291
MPL	Multiply Packed Long	SS	F9	-	10-292
MSF	Multiply Short Float	SS	AE	3	10-294
MVAST	Move and Set Tags	SS	B4	-	10-269
MVBI	Move Byte Immediate	SI	9B	-	10-270
MVBIP	Move Byte Immediate and Propagate	SI	AC	-	10-271
MVC	Move Characters	SS	CB	-	10-273
MVCAT	Move Characters and Tags	SS	B5	-	10-274
MVCL	Move Characters Long	SS	EB	-	10-276
MVCR	Move Character Register	SS	DB	-	10-272
MVHI	Move Halfword Immediate	SI	AB	-	10-278
MVMC	Move Virtual Page	RR	1D	-	10-278.1
MVNN	Move Numeric to Numeric	SS	BA	-	10-279
MVNZ	Move Numeric to Zone	SS	BB	-	10-280
MVPS	Move Packed Shifted	SS	FB	-	10-282
MVPSZ	Move Packed Shifted Zero	SS	FC	-	10-284
MVZN	Move Zone to Numeric	SS	BC	-	10-284.1
MVZZ	Move Zone to Zone	SS	BD	-	10-285
MWS	Multiply Word Storage	SS	EC	-	10-297
NB	AND Byte	RS	79	1	10-32
NBI	AND Byte Immediate	SI	98	-	10-33
NBR	AND Byte Register	RR	18	-	10-34
NBRI	AND Byte Register Immediate	RI	48	-	10-35
NC	AND Characters	SS	C8	-	10-36
NH	AND Halfword	RS	80	4	10-37
NHR	AND Halfword Register	RR	28	-	10-38
NHRI	AND Halfword Register Immediate	RI	58	-	10-39

Mnemonic	Instruction	Format	Operation Code	Extender	Page
OB	OR Byte	RS	79	0	10-298
OBI	OR Byte Immediate	SI	99	-	10-299
OBR	OR Byte Register	RR	19	-	10-300
OBRI	OR Byte Register Immediate	RI	49	-	10-301
OC	OR Characters	SS	C9	-	10-302
OH	OR Halfword	RS	80	3	10-303
OHR	OR Halfword Register	RR	29	-	10-304
OHRI	OR Halfword Register Immediate	RI	59	-	10-305
PPR	Perform Paging Request	SS	E8	-	10-306
RACM	Reset Address Compare Mode	RR	0D	3	10-320
RAHR	Return Available Hold Record	RS	0D	5	10-325
RCB	Reset Chain Busy	RR	0D	6	10-321
RECC	Receive Count	SI	67	-	10-312
RECM	Receive Message	SS	D9	-	10-315
RMCM	Reset Machine Check Mode	RR	0D	2	10-324
RPDE	Remove Primary Directory Entry	SI	83	4	10-319
RRCRR	Read Reference and Change and Reset Reference	SI	83	0	10-310
SACM	Set Address Compare Mode	SI	4B	-	10-346
SC	Subtract Characters	SS	C1	-	10-384
SCAN	Scan	SS	CF	-	10-328
SCB	Set Chain Busy	RR	36	-	10-348
SENDC	Send Count	SI	66	-	10-334
SENDM	Send Message	RS	68	-	10-337

Mnemonic	Instruction	Format	Operation Code	Extender	Page
SENDMW	Send Message and Wait	RS	69	-	10-341
SETCC	Set Clock Comparator	SI	6D	2	10-350
SETIND	Set Indicator	SI	5A	-	10-352
SETIT	Set Interval Timer	SI	6A	-	10-352
SETTOD	Set Time-Of-Day Clock	SI	6D	4	10-354
SH	Subtract Halfword	RS	80	1	10-386
SHR	Subtract Halfword Register	RR	21	-	10-387
SLA	Shift Left Arithmetic	RR	03	-	10-356
SLB	Subtract Logical Byte	RS	71	0	10-388
SLBR	Subtract Logical Byte Register	RR	11	-	10-389
SLC	Subtract Logical Characters	SS	C4	-	10-390
SLF	Subtract Long Float	SS	CE	2	10-393
SLH	Subtract Logical Halfword	RS	91	0	10-391
SLHCT	Shift Left Halfword and Count	RS	61	-	10-357
SLHR	Subtract Logical Halfword Register	RR	31	-	10-392
SLL	Shift Left Logical	RR	01	-	10-358
SP	Subtract Packed	SS	F1	-	10-396
SRA	Shift Right Arithmetic	RR	04	-	10-359
SRL	Shift Right Logical	RR	02	-	10-360
SSCA	Store and Set Computational Attributes	SS	BE	8	10-365
SSF	Subtract Short Float	SS	AE	2	10-398
ST	Store	RS	96	-	10-364
STACK	Stack	RR	1B	-	10-362
STB	Store Byte	RS	76	-	10-370
STCC	Store Clock Comparator	SI	6D	3	10-372
STH	Store Halfword	RS	86	-	10-374
STIT	Store Interval Timer	SI	8C	-	10-376
STM	Store Multiple	RS	97	-	10-378
STMB	Store Multiple Byte	RS	77	-	10-379
STMH	Store Multiple Halfword	RS	87	-	10-380
STSOP	Store Space Offset Pointer	RS	83	9	10-381

Mnemonic	Instruction	Format	Operation Code	Extender	Page
STST	Store and Set Tags	RS	65	-	10-368
STTOD	Store Time-Of-Day Clock	SI	6D	5	10-382
SVLM	Supervisor Link Monitored	RR	1F	-	10-407
SVLM1	Supervisor Link Single Monitored	SI	5B	-	10-414
SVL0	Supervisor Link Short	RR	3F	-	10-408
SVL1	Supervisor Link Single	SI	5D	-	10-411
SVL2	Supervisor Link Double	SS	DF	-	10-404
SVX	Supervisor Exit	RR	3E	-	10-402
TI	Terminate Immediately	RR	0D	4	10-417
TMBI	Test Under Mask Byte Immediate	SI	9D	-	10-418
TMBIBO	Test Under Mask Byte Immediate and Branch If Ones	SI	E1	-	10-419
TMBIBZ	Test Under Mask Byte Immediate and Branch If Zeros	SI	E0	-	10-420
TMBR	Test Under Mask Byte Register	RR	13	-	10-421
TR	Translate	SS	CC	-	10-422
TRIM	Trim	SI	72	-	10-417
TRR	Translate Register	SS	8E	3	10-426
TRT	Translate and Test	SS	CD	-	10-424
UNSTK	Unstack	RR	2B	-	10-428
XB	Exclusive OR Byte	RS	79	2	10-201
XBI	Exclusive OR Byte Immediate	SI	9A	-	10-202
XBR	Exclusive OR Byte Register	RR	1A	-	10-203
XBRI	Exclusive OR Byte Register Immediate	RI	4A	-	10-204
XC	Exclusive OR Characters	SS	CA	-	10-205
XH	Exclusive OR Halfword	RS	80	5	10-206
XHR	Exclusive OR Halfword Register	RR	2A	-	10-207
XHRI	Exclusive OR Halfword Register Immediate	RI	5A	0	10-208
ZAC	Zero and Add Characters	SS	C6	-	10-430

Appendix D. Glossary

<: Less than.

>: Greater than.

≥: Greater than or equal to.

=: The value to the left of the symbol is the same as the value to the right of the symbol.

≠: The value to the left of the symbol is not the same as the value to the right of the symbol.

ACQ: Available CRE queue.

active task: The task that is currently executing.

address compare mode: The condition of the machine when an address compare exception can occur if a storage location is either referenced, accessed, or altered.

address event: An I/O event stack entry type that indicates that a page boundary crossing occurred during the modification of a resolved virtual address contained in an I/O address register.

address list element (ALE): An 8-byte IMP object containing a virtual or virtual=real address to be used during page chaining operations.

address operation block (AOB): One of the five forms of the operation block that is used to save, modify, or load the I/O address registers during an operation.

address register (AR): A register in which an address is stored.

AHR: Available hold record.

ALE: Address list element.

ALU: Arithmetic and logic unit.

ANSI: American National Standards Institute.

AOB: Address operation block.

AR: Address register.

argument: (1) (ISO) An independent variable. (2) (ISO) Any value of an independent variable.

arithmetic and logic unit (ALU): The part of a computer that performs arithmetic, logic, and related operations.

asynchronous: (1) Without regular time relationship. (2) Unexpected or unpredictable with respect to the execution of a program's instruction.

available CRE queue (ACQ): The mechanism by which CREs (call/return elements) are made available to the processor and eventually to a TDE (task dispatching element).

available hold record (AHR): An unused hold record.

B_x: A 6-byte base register represented by B is used as operand X.

base: The number system in which an arithmetic value is represented.

base register (B): The register that contains the address of the start of the instruction stream.

basic status (BSTAT): Two bytes of adapter response data reported by the channel to the I/O event stack.

bias: In binary floating-point storage formats, the constant value that, when added to the signed exponent of a binary floating-point number, produces a non-negative biased exponent. The bias for short format is 127 and for long format is 1023.

biased exponent: (1) In binary floating-point storage formats, the non-negative sum of the signed exponent of a binary floating-point number and a constant value (bias). (2) The value between the maximum and minimum field values that is used to represent the signed exponent of a normalized binary floating-point number. The range of biased exponent values is 1 through 254 for the short format and 1 through 2046 for the long format. Contrast with *signed exponent*.

BID: Byte identifier.

binary digits: The numbers 0 and 1 that are used to represent a value in the numbering system that has 2 as its base.

binary floating-point number: A conceptual representation of a numerical value that contains a signed significand and a signed exponent. Its numerical value is the signed product of its significand and 2 raised to the power of its exponent. Contrast with *long format* and *short format* which are used to represent binary floating-point numbers in storage. A binary floating-point number is either a normalized number, a denormalized number, or a signed zero.

binary floating-point value: One of the set of values supported for binary floating-point operations. The set of values supported is composed of binary floating-point numbers, infinity, and not-a-number.

binary point: The point that separates the integer digits from the fraction digits in the numbering system that has 2 as its base, similar to decimal point.

branching instructions: Instructions that may change the sequence of program execution.

BSTAT: Basic status.

built-in function: A well defined HMC (horizontal microcode) operation that is used to enhance the performance of the processor.

byte: A group of 8 adjacent bits that a computer processes as a unit.

byte identifier (BID): Bits 39-47 of a virtual address. The portion of a virtual address that identifies the specific byte of data addressed within a page.

call/return element (CRE): A resident storage area used to save the status of a procedure during an SVL (supervisor linkage).

chain: Two or more objects linked together.

chaining: A system of storing records in which each record belongs to a list or group of records and has a linking field for tracing the chain.

channel interface: The interface between the vertical microcode I/O manager tasks responsible for I/O and operational unit tasks that handle I/O operations.

command end: A function event used by an I/O device to communicate error or exception status to the device I/O manager task.

comparison instructions: Instructions that are designed to test the relationship between items of data.

completed: A term used to describe an action that the system may take when an exception occurs during the execution of an instruction. The instruction is allowed to continue to completion with predictable results and the IAR is advanced to the next instruction address. The ILC indicates the length of the completed instruction.

concatenate: To link together two or more operands.

concurrent: (1) (ISO) Pertaining to the occurrence of two or more activities within a given interval of time. (2) Contrast with consecutive, sequential, simultaneous.

condition code: A 4-bit code that reflects the results of most of the arithmetic, logical, and other manipulative and control instructions.

control address table: The assigned storage location for certain control information that must be known to the processor to execute IMP (internal microprogramming) tasks.

control storage: The storage in which HMC is loaded.

control storage address register (CSAR): The address register used by the HMC (horizontal microcode) to control command sequencing.

CPU: Processing unit.

CPU cluster: The planer board, array board, interposers, main end control storage cards, and the terminators.

CRE: Call/return element.

CSAR: Control storage address register.

D_x: The displacement represented by D is used with operand X.

dequeue: To remove items from a queue.

data field length: The number of bytes of data in the source data field.

denormalized number: In binary floating-point storage formats, the representation of a nonzero number in which the exponent field contains a reserved value (0) at the format's minimum and the fraction field is greater than 0. The significand of the number represented has an integer value of 0, which is implied by the storage representation and a fraction value from the fraction field. The reserved value of 0 in the exponent field indicates the value of the signed exponent (power of 2) is decimal -126 for the short format and decimal -1022 for the long format.

dequeue: To remove items from a queue.

descriptor: That portion of an IMP (internal microprogramming) object that is used as a unique identifier.

destination: See *result field.*

device status (DSTAT): The bytes of information required for proper device maintenance.

displacement: (1) The distance from the beginning of a record, block, or segment to the beginning of a particular field. (2) Synonym for relative address.

DSTAT: Device status.

E: Operation code extender field.

EBCDIC: Extended binary coded decimal interchange code.

enqueue: To place items on a queue.

error event: An I/O event stack entry type that indicates error conditions involving the channel hardware. These errors are handled by IMP (internal microprogramming) channel error microcode.

error recovery procedure: A set of instructions designed to help isolate and, where possible, recover from errors in equipment. These instructions are often used with programs that record the statistics of machine malfunctions.

event handler: A program, specified in an event monitor, that is to receive control when the event occurs.

event stack: A list of 4-byte entries that contain function, address, or error events. The entries are placed on the list by the channel hardware and are removed from the list by the I/O event handler.

exception: The occurrence of a monitorable machine or user-defined condition directly associated with the execution of a particular function within a program. Exceptions generally represent an abnormality detected by the machine or by a program. Exceptions are signaled to a single monitor within the associated process.

explicit designation: Designation by the use of information contained within an operand of an instruction.

explicit invocation: Causing a procedure to wait by the use of an IMP (internal microprogramming) instruction.

explicit length: Length of an instruction as stated within the instruction.

exponent range: In binary floating-point storage formats, the set of integer exponents that can be represented in a particular format. The representable signed exponent range is decimal -126 through +127 for short format and decimal -1022 through +1023 for long format.

extender (E): A 4-bit extension of the IMP operation code.

FIB: Fill instruction buffer.

field replaceable unit: An assembly that is replaced in its entirety when any one of its components fails.

fill instruction buffer (FIB): An HMC (horizontal microcode) status control to fill the instruction stream buffer.

floating-point format: In binary floating-point representation the storage format used to represent a binary floating-point value. See *long format* and *short format.*

FOB: Function operation block.

format's maximum: In binary floating-point storage formats, the value of 255 (short format) or 2037 (long format) in the exponent field. This value indicates that either a signed infinity (fraction equals 0) or a not-a-number (fraction does not equal 0) is represented in the storage format.

format's minimum: In binary floating-point storage formats, the value of zero in the exponent field. This value indicates that either a zero floating-point value (fraction equals 0) or a denormalized number (fraction does not equal 0).

fraction: In binary floating-point representation, the value to the right of the binary point.

FRAT: Function routine address table.

free: To unlock a system or data base object.

FSTAT: Functional status.

function event: An I/O event stack entry type that communicates device or IMP (internal microprogramming) task request to an operational unit task.

function operation block (FOB): One of five forms of the operation block that identify the operational unit and convey the command to be executed by the operational unit.

function routine address table (FRAT): An indexed table of addresses to instruction streams that perform specific tasks.

functional status (FSTAT): One of 4 bytes of operational unit information that can be required by the program for normal device operation.

gap length: The number of bytes of data between fields in the source operand.

gap offset: The number of bytes to the next gap in the source.

greater than (>): The value to the left of the symbol is greater than the value to the right of the symbol.

halfword: 16 bits or 2 bytes on an integral boundary.

hash hold table (HHT): A storage page which contains halfword entries, that are used as an index (after being manipulated) into the hold record area.

hash synonyms: Equal hash values that are obtained by hashing different object addresses.

hashing: The compression of the 39-bit field formed by linking the segment and page identifier fields of a virtual address.

HHT: Hash hold table.

HMC: Horizontal microcode.

hold: A lock on a given system or data base object.

hold record (HR): A record of information describing the constraints that have been imposed on the use of an object.

hold record area: A virtual addressing segment that contains all object chains of HRs.

horizontal microcode (HMC): Microcode that exhibits a high degree of parallelism of execution, controls the detailed state of the hardware, and supports the IMP (internal microprogramming) instruction set.

HR: Hold record.

I_x: The immediate data represented by I is used as operand X in the instruction in which it appears.

I/O manager queue (IOMQ): An IMP send/receive queue used to communicate I/O command response information to an I/O manager task from a device operational unit task.

I/O register table: A table containing pointers to the queue control table. It is accessed by using the operational unit as an index.

IAR: Instruction address register.

ILC: Instruction length count.

immediate data operand: An operand that contains the data attributes and the data in the instruction.

IMP: Internal microprogramming.

IMP objects: A separately addressable unit (or collection or data) that has associated attributes as well as operating characteristics based on these attributes.

IMPL: Initial microprogram load.

implicit designation: Designation by the use of the operation code as an index into a storage table.

implicit invocation: Causing a procedure to wait by the use of an HMC (horizontal microcode) procedure.

implicit leading bit: A bit that does not appear in the storage form of a binary floating-point number. This bit is understood to be to the left of the assumed binary point. See *significand.*

implied length: The length of the instruction as recognized by the specific operation code being used.

inexact result: A result that occurs when bits of the significand are lost in rounding the intermediate result to the precision of the result field or when infinity is stored as the result of a masked overflow.

infinity: In binary floating-point operation, a name for the values beyond the minimum and maximum finite values that can be represented. These finite values are represented in the storage formats when the exponent field contains a reserved value (255 for short format and 2047 for long format) at the formats' maximum and the fraction field is 0. Infinity can be positive or negative.

infinity arithmetic: The adding, subtracting, multiplying, dividing, and comparing of values that are beyond the minimum and maximum values that can be represented as finite values in the binary floating-point format.

initial microprogram load (IMPL): The initiation of processing when the contents of storage are not suitable for processing.

initial program load (IPL): The initialization procedure that causes an operating system to start operations.

input/output: In System/38 the name given the microprocessor used in the attachment of various I/O devices.

input/output controller (IOC): (ISO) A functional unit in a data processing system that controls one or more units of peripheral equipment.

input/output manager (IOM): A VMC (vertical microcode) programming object that controls the flow of information (control and I/O data) to and from an I/O unit.

instruction address register (IAR): (ISO) A register from whose contents the address of the next instruction is derived.

instruction length count (ILC): A 3-bit code that provides the length of the last instruction executed.

integral boundary: A location in main storage at which a fixed-length field, such as a halfword or doubleword, must be positioned. The address of an integral boundary is a multiple of the length of the field, in bytes.

intermediate denormalized floating-point number: In binary floating-point operation, an intermediate unrounded form of the result in which a value that is too small to be represented in the floating-point format of the result has had the significand digits shifted right (zeros are supplied on the left) and the exponent incremented until the exponent attains the format's assumed value for denormalized numbers (-126 for a short format and -1022 for long format).

intermediate result: In floating-point operations, the normalized result produced prior to the adjustments required to store it in the result field.

interval timer: A means of measuring elapsed time and determining when a prespecified amount of time has elapsed.

invocation: An invocation is the execution of a program. It represents the status of the process after the program is invoked. When one programs calls another program, the two programs are said to be in different invocations. The invocation of a program that is called a second time by the same calling invocation is also considered to be a different invocation. Automatic storage is allocated for a program at every invocation.

IOC: Input/output controller.

IOM: Input/output manager.

IOMQ: I/O manager queue.

ISO: International Organization for Standardization.

$J_{x:}$ Jump displacement.

jump displacement (J_x): The number of bytes (address increments) added to the instruction address after a jump instruction is executed.

L_x: The length of the operand represented by L is used as operand X in the instruction in which it appears.

LB: Lookaside buffer.

LOB: Loop operation block.

local storage register (LSR): A register that is assigned to hold processor information.

long format: In binary floating-point operations, the storage representation of a binary floating-point number, a not-a-number, or infinity. The long format is a 64-bit string in which bit 0 is the sign field, bits 1 through •• are the 11-bit exponent field, and bits 12 through 63 are the 52-bit fraction field. Contrast with *binary floating-point number* which is the conceptual view of the number.

lookaside buffer (LB): A separate hardware storage array used to store recently translated virtual addresses along with their corresponding real addresses.

loop operation block (LOB): One of the five forms of the operation block that allows an operation program to contain a loop for efficient operation.

LSR: Local storage register.

M_x: The mask represented by M is specified for operand X.

machine check (MCHK): A detected machine malfunction that can occur in hardware or HMC.

machine check log buffer (MCLB): A data area in virtual storage used to store the processor status and the task status when a machine check occurs.

machine communications area (MCA): The assigned storage locations, which contain control information required for VMC objects to communicate with each other.

machine interface (MI): The instruction set interface to the machine. The instruction set is called the System/38 instruction set.

main storage: See *real storage.*

MCA: Machine communications area.

MCHK: Machine check.

MCLB: Machine check log buffer.

message operation block (MOB): One of the five forms of the operation block that either sends a message to a queue or increments a counter.

MI: Machine interface.

microcode: The instructions providing the basic machine functions and supporting the machine interface.

MOB: Message operation block.

monitor: A process that checks for the occurrence of an event or exception and takes action based on that event or exception.

MSAR: Main storage address register.

NaN: See *not-a-number.*

negative infinity: See *infinity.*

normalized number: In binary floating-point storage formats, the representation of a nonzero floating-point number whose exponent field contains a biased exponent. The range of biased exponent values is 1 through 254 for the short format and 1 through 2046 for the long format. The significand of the number represented has an integer value of 1 and a fraction value from the fraction field. Note that the exponent field values of 0 and 255 for the short format and 0 and 2047 for the long format are used to indicate the representations of infinity, not-a-number, denormalized number, and signed 0.

no-operation: No operation is performed. The IAR is updated to the next sequential instruction.

not-a-number: In binary floating-point storage formats, the name for a value that is not interpreted as a number. A not-a-number (NaN) is represented by an exponent field that contains a reserved value at the format's maximum (255 for short format and 2047 for long format) and a fraction field that does not contain 0. A not-a-number may represent the results of incorrect combinations of operands in floating-point operations.

nullified: A term used to describe the action the system may take when an exception occurs during the execution of an instruction. The instruction is stopped with the IAR not advanced to the next instruction address. The ILC is set to zero.

OB: Operation block.

object: A separately addressable unit that has associated with it certain attributes as well as operational characteristics based on these attributes.

offset: The distance from the beginning of a register or record to the beginning of a particular field.

op code: Operation code.

operation block (OB): The portion of the ORE (operation request element) that contains operation unit information. The five types of operation blocks are: address operation block, function operation block, loop operation block, message operation block, and program operation block.

operation code: An 8-bit code which specifies the operation to be performed by the IMP instruction to which the code is unique.

operation program (OP): A set of operation blocks placed in storage and executed together prior to any response.

operation request element (ORE): An IMP (internal microprogramming) message, placed on an operational unit queue, to cause an I/O operation. It consists of a standard IMP queue element header, a status field, and an operation block.

operational unit (OU): An I/O device or source of asynchronous events together with an OU task that controls the device or the event.

operational unit number: A 1-byte number that uniquely defines an operational unit and is used as an index into the operational unit table to locate the queue control table.

operational unit queue (OUQ): The queue upon which OREs (operation request elements) are placed by the source/sink component below MI (machine interface). There is one operational unit queue for each operational unit.

operational unit task: A microcode task that exists for each operational unit that performs operations such as operation block execution and command completion functions. The operational unit task services the operational unit queue and I/O events.

ORE: Operation request element.

OU: Operational unit.

OUQ: Operational unit queue.

page: (1) (ISO) In a virtual storage system, a fixed-length block that has a virtual address and that can be transferred between real storage and auxiliary storage. (2) *A block of instruction, or data, or both, that can be located in main storage or in auxiliary storage. Segmentation and loading of these blocks is automatically controlled by a computer. (3) To transfer instructions, or data, or both between real storage and external page storage. (4) In System/38 a page contains 512 bytes.

page fault: In a virtual storage system, a program exception that occurs when a page that is not in main storage is referred to by an active task.

page frame: In a virtual storage system, a 512-byte block of main storage that can contain a page.

page identifier (PID): Bits 32 through 38 of a virtual address.

PD: Primary directory.

PEM: Program event monitor.

permanent storage assignments: The assignments of storage locations contained within the control address table.

PID: Page identifier.

pin count (PINCNT): A counter that records the number of times a page of storage is pinned while the page is in storage. The pin count is used for holding pages in storage.

PINCNT: Pin count.

pinning: A mechanism used to hold pages in storage.

placeholder: A symbol that may be replaced by some other value.

PMCH: Processor machine check handler.

POB: Program operation block.

positive infinity: See *infinity.*

preempt wait: A task switch that occurs if a TDE is enqueued to the TDQ at a higher priority than the current TDE.

primary directory (PD): A list of entries in which each entry contains the virtual address and the status of a page frame in main storage.

procedure: (1) *(ISO) The course of action taken for the solution of a problem. (2) *The description of the course of action taken for the solution of a problem.

processing unit (CPU): The unit of the computer that includes circuits controlling the interpretation and execution of instructions.

processor machine check handler (PMCH): An HMC (horizontal microcode) routine that attempts to recover from apparent machine malfunctions.

program event monitor (PEM): The processor comparing the initial byte of the instructions to determine if they fall within the range of the PEM start and PEM stop addresses.

program operation block (POB): One of five forms of the operation block that is used in an operation request element when an operation program is to be executed.

QCT: Queue control table.

quadword: A group of 4 consecutive words located at an integral boundary.

queue: (1) A line or list formed by items in a system waiting for service, for example, tasks to be performed or messages to be transmitted in a message switching system. (2) A system object to which a list or line of items are related while waiting for service.

queue control table (QCT): A table, accessed by microcode and machine product code, that controls I/O operations. There is one table for each operational unit.

r_x: A one-byte register represented by r is used as operand X.

R_x: A halfword register represented by R is used as operand X.

RAR: Resolved address register.

real storage: (1) (ISO) The main storage in a virtual storage system. Physically, real storage and main storage are identical. Conceptually however, real storage represents only part of the range of addresses available to the user of a virtual storage system. Traditionally, the total range of addresses available to the user was that provided by main storage. (2) Same as processor storage.

reserved values: In binary floating-point representation, the exponent field values of 0 and 255 for the short format and 0 and 2047 for the long format that are used to indicate representations of infinity, not-anumber, denormalized number, and 0.

resolved address: A translated virtual address.

result offset: The number of bytes of the result field that are to be processed (upon entry to the instruction).

RI: An instruction type that uses register and immediate operand parameters.

rotary switches: Those console switches that are used to control the basic machine functions.

rounding: In binary floating-point operations, a modification of a value, if necessary, so that it is representable in the format of the result field. An inexact result exception condition may occur due to rounding.

rounding to nearest: In floating-point operations, to modify a value to the nearest representable value. However, if the value of those digits being dropped is exactly half of the least significant digit of the retained value, the nearest representable value in which the least significant digit is even is chosen.

round toward negative infinity: In floating-point operations, a modification of a value to the representable value that is closest to but no greater than the unmodified value. The result may be negative infinity.

round toward positive infinity: In floating-point operations, a modification of a value to the representable value that is closest to but not less than the unmodified value. The result may be positive infinity.

round toward zero: In floating-point operations, a modification of a value to the representable value that is closest to and no greater in absolute value than the unmodified value. The result may be 0.

RR: An instruction type that uses only register operands.

RS: An instruction type that uses register and storage operand parameters.

S: A 4-byte register represented by S is used as operand.

SCA: System control adapter.

scalar: *(1) (ISO) A quantity characterized by a single number. (2) Contrast with vector.

SDR: Statistical data recording.

segment: A unique, continuous area of virtual storage. Segments are nonoverlapping and noncontinuous with each other.

segment identifier (SID): Bits 0 through 31 of a virtual address.

send/receive counter (SRC): The IMP (internal microprogramming) instruction object used to exchange intertask information and to synchronize the flow of control between tasks; a count field used for control but no messages are enqueued.

send/receive message (SRM): An IMP (internal microprogramming) instruction object that contains a message and may be enqueued to an SRQ (send/receive queue).

send/receive queue (SRQ): An IMP (internal microprogramming) instruction object that is used to exchange intertask information to synchronize the flow of control between tasks.

set: (1) (ISO) To put all or part of a data processing device into a specified state. (2) Contrast with reset.

short format: In binary floating-point operations, the storage representation of a binary floating-point number, not-a-number, or infinity. The short format is a 32-bit string in which bit 0 is the sign field, bits 1 through 8 are the 8-bit exponent field, and bits 9 through 31 are the 23-bit fraction field. See also *binary floating-point number.*

SI: An instruction type that uses storage and immediate operand parameters.

SID: Segment identifier.

signed exponent: In floating-point operations, the arithmetic representation of the exponent value of the floating-point number.

signed zero: In binary floating-point formats, the representation of the number 0 whose exponent field contains a reserved value at the format's minimum and a fraction field that is equal to 0. Zero can be positive or negative; however, positive 0 for denormalized numbers and 1 for normalized numbers.

significand: In binary floating-point operations, the part of a binary floating-point number that is composed of binary digits which contain integers to the left of a binary point and one or more fraction digits to the right. The value of the integer is implied by the storage representation of a binary floating-point number. The value of the integer digit is 0 for denormalized numbers and 1 for normalized numbers.

source/sink: Pertaining to devices capable of originating or accepting data signals to or from a transmission device (such as a central processor) and pertaining to the data management components supporting such devices. Source/sink devices include locally and remotely attached, batch and work station devices, but not the internal storage of the system.

source offset: The number of bytes of the source field that are to be processed (upon entry to the instruction).

source operand: The operand that contains the source as provided by the user of a data processing system.

source record length: The number of bytes of data in a source record as provided by the user.

SRC: Send/receive counter.

SRM: Send/receive message.

SRQ: Send/receive queue.

SS: An instruction type that uses only storage operands for parameters.

stack: (1) (ISO) A list that is constructed and maintained so that the next item to be retrieved and removed is the most recently stored item still in the list, that is, last in-first out. Synonymous with pushdown list.

statistical data recording (SDR): Statistical information for each I/O device on the system stored in auxiliary storage by VMC.

storage capacity: The number of bytes provided without regard to the storage width.

storage width: The number of bytes that can be fetched or stored in one storage cycle.

string: *(1) (ISO) A linear sequence of entities such as characters or physical elements.

supervisor linkage (SVL): The method by which IMP (internal microprogramming) procedure switching is accomplished within a task and the method by which IMP exceptions are reported.

suppressed: A term used to describe the action the system may take when an exception occurs during the execution of an instruction. The instruction is not allowed to continue and the IAR is advanced to the next instruction address. The result fields are not changed. The ILC indicates the length of the suppressed instruction.

suspended: A term used to describe the action the system may take when an exception occurs during the execution of an instruction. The instruction is stopped at the point of the exception and checkpoint data is stored in a reserved area. The IAR is not advanced to the next instruction address so that the operation can be resumed at the point of the exception. The ILC is set to zero.

SVL: Supervisor linkage.

SVL table: A table in storage that is used to contain the number of registers to be stored, the address of the procedure to which control is passed, and other descriptive control information.

synchronous: Pertains to arising, existing, or happening precisely at the same time.

system control adapter (SCA): An interface used in conjunction with the CE/operator panel for initiating and monitoring the system during system initiation.

system specialization: The tailoring of the system (programming and devices) for installation and the redefinition of the system when the user adds a device or feature or changes some part of the programming.

system unit: The main unit of the system, which contains the processing unit, the system console keyboard/display, the operator/service panel, the diskette magazine drive, main sotrage, auxiliary storage, the work station controller, and the communications subsystem.

tag: One or more characters, attached to a set of data, that contains information about the set, including its identification.

task dispatching element (TDE): An IMP (internal microprogramming) object used to identify a task and the attributes associated with that task.

task dispatching queue (TDQ): An IMP (internal microprogramming) object used by the task dispatcher to allocate processor time to the dispatchable tasks in the system.

tasking: The process of controlling the execution of IMP (internal microprogramming) tasks.

tasks: (1) A semi-independent unit of work that can be performed concurrently with other tasks and requires coordination with other tasks only at certain points within the execution. (2) Units of work activated by the task dispatcher.

TDE: Task dispatching element.

TDQ: Task dispatching queue.

terminated: A term used to describe the action the system takes when an exception occurs during the execution of an instruction. The instruction is terminated at the point of the exception with unpredictable results and the IAR is advanced to the next instruction address. The ILC indicates the length of the terminated instruction.

time quantum: The time span remaining for a task to execute.

time-of-day clock: The object used by the system to accumulate time within the system.

TOD: Time of day.

trap: (ISO) An unprogrammed conditional jump to a specified address that is automatically activated by hardware. (2) A recording being made of the location from which the dump occurred.

trapped instructions: See *trap*.

V=R: Virtual address equals real address.

V=V: Virtual address equals virtual address.

VAT: Virtual address translator.

vertical microcode (VMC): Microcode that defines logical operations on data, is primarily sequential in execution and supports the System/38 machine instruction set.

virtual = real: The planned occurrence in addressing when a virtual address addresses the same part of memory as the real address.

virtual address: The address of a storage location in virtual storage.

virtual address translator (VAT): Hardware which converts a virtual storage address to a real storage address.

virtual storage: The combination of main storage and auxiliary storage, treated as a single addressable unit.

VMC: Vertical microcode.

word: 32 bits or 4 bytes on an integral boundary.

zone: The leftmost 4 bits of a byte in a decimal field are called zone, except for the rightmost byte of the field, where these bits may be treated either as a zone or as a sign code.

Index

IBM System/38
Internal Microprogramming
Instructions, Formats, and Functions
Reference Manual

SC21-9037-3

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

□ If your comment does not need a reply (for example, pointing out a typing error) check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

□ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): **Comment(s):**

Please contact your nearest IBM branch office to request additional publications.

Name ____________________

Company or
Organization ____________________

Address ____________________

____

City State Zip Code

Phone No. ( ) ____________________

Area Code

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

Cut Along Line

Fold and tape. **Please do not staple.**

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 40 / ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
Rochester, Minnesota, U.S.A. 55901

Fold and tape. **Please do not staple.**

Cut Along Line

IBM

IBM System/38
Internal Microprogramming
Instructions, Formats, and Functions
Reference Manual

SC21-9037-3

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

□ If your comment does not need a reply (for example, pointing out a typing error) check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

□ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): **Comment(s):**

Please contact your nearest IBM branch office to request additional publications.

Name ______________________

Company or
Organization ______________________

Address ______________________

____
City State Zip Code

Phone No. ( ) ______________________
Area Code

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

Fold and tape. **Please do not staple.**

Cut Along Line

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 40 / ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
Rochester, Minnesota, U.S.A. 55901

Fold and tape. **Please do not staple.**

Cut Along Line

IBM®

IBM®

IBM System/38 Internal Microprogramming Instructions, Formats, and Functions Reference Manual   Printed in U.S.A.   SC21-9037-3

SC21-9037-3

SC21-9037-3