ACS-1

MPM INSTRUCTION MANUAL

Part	Title
1	Load and Store Operations
2	Move Operations
3	Floating Point Arithmetic
4	Integer Arithmetic
5	Index Arithmetic
6	Compare Operations
7	Shift Operations
8	Logical Operations
9	Branch at Exit Operations
10	Input/Output Operations
11	Tag and Directory Operations
12	Special Registers

INDEX

LR	Load right half arithmetic	1-16
LX	Load index	1-4
LXA	Load index per alternate key	1-4
LXH	Load index (half word format)	1-4
MAC	Move arithmetic bit to condition bit	2-6
MAX	Move arithmetic to index	2-2
MCX	Move condition bit to index bit	2-6
MDN	Multiply double normalized	3-17
MDR	Multiply double rounded	3-18
MDU	Multiply double unnormalized	3-19
MI	Multiply integer	4-5
MKL	Move constant to left half arithmetic	2-3
MKR	Move constant to right half arithmetic	2-3
MLX	Move location to index	2-4
MMI	Multiply mixed integer	4-5
MMN	Multiply mixed normalized	3-20
MMU	Multiply mixed unnormalized	3-20
MN	Multiply normalized	3-17
MOT	Move one to T register bit	10-5
MR	Multiply rounded	3-18
MSX	Move special to index	2-4
MSXZ	Move special to index and zero	2-5
MTX	Move T register to index	10-4
MU	Multiply unnormalized	3-19
MX	Multiply index	5-4
MXA	Move index to arithmetic	2-2
MXC	Move index bit to condition bit	2-6
MXK	Multiply index by constant	5-10
MXS	Move index to special	2-4
MXSO	Move index to special by oring	2-5

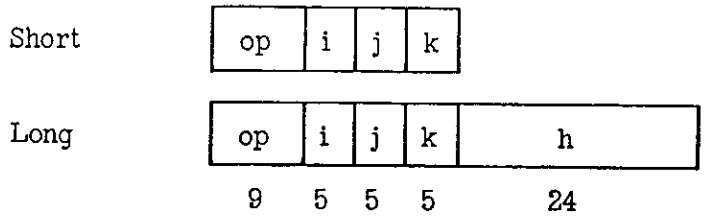
SI	Subtract integer	4-4
SIA	Integer shift arithmetic	7-12
SIAC	Integer shift arithmetic by constant	7-13
SID	Integer shift double	7-15
SIDC	Integer shift double by constant	7-15
SIO	Start I/O	10-2
SIOA	Start I/O per alternate key	10-3a
SIX	Integer shift index	7-12
SIXC	Integer shift index by constant	7-13
SKAND	Skip if and	9-9
SKEQ	Skip if equal	9-9
SKFAF	Skip if false and false	9-9
SKFOF	Skip if false or false	9-9
SKOR	Skip if or	9-9
SKTAF	Skip if true and false	9-9
SKTOF	Skip if true or false	9-9
SKXOR	Skip if exclusive or	9-9
SN	Subtract normalized	3-13
SNF	Set negative, floating	3-26
SNI	Set negative, integer	4-10
SNX	Set negative, index	5-13
SPF	Set positive, floating	3-26
SPI	Set positive, integer	4-10
SPX	Set positive, index	5-13
SR	Subtract rounded	3-14
STA	Store arithmetic	1-10
STAA	Store arithmetic per alternate key	1-11
STAH	Store arithmetic (half word format)	1-10
STAT	Store arithmetic, true indexing	1-14
STATH	Store arithmetic, true indexing (half word format)	1-14
STD	Store double arithmetic	1-13
STDH	Store double arithmetic (half word format)	1-13

STL	Store left half arithmetic	1-16
STMA	Store multiple arithmetic	1-23b
STMAA	Store multiple arithmetic per alternate key	1-23c
STMX	Store multiple index	1-20
STMXA	Store multiple index per alternate key	1-21
STMZ	Store multiple zeros	1-24
STMZA	Store multiple zeros per alternate key	1-24
STR	Store right half arithmetic	1-16
STX	Store index	1-5
STXA	Store index per alternate key	1-6
STXH	Store index (half word format)	1-5
SU	Subtract unnormalized	3-15
SVC	Supervisor call	9-13
SVR	Supervisor return	9-14
SWS	Swap with storage	1-7b
SWSA	Swap with storage per alternate	1-7b
SX	Subtract index	5-4
TAFA	Logical true and false, arithmetic	8-3
TAFC	Logical true and false, condition	8-4
TAFX	Logical true and false, index	8-3
TC	Test channel	10-3b
TOFA	Logical true or false, arithmetic	8-3
TOFC	Logical true or false, condition	8-4
TOFX	Logical true or false, index	8-3
XORA	Logical exclusive or, arithmetic	8-3
XORC	Logical exclusive or, condition	8-4
XORX	Logical exclusive or, index	8-3

LOAD AND STORE OPERATIONS

The load operations replace the contents of index (X) registers, or arithmetic (A) registers, with information from storage. The storage contents remain unchanged. The store operations replace information in storage with information from one or more of the X or A registers. The register contents remain unchanged. The count to storage and swap with storage instructions act as both a load and a store and thus may change both the register and storage.

The load and store instructions have one of the following formats:



For each operation the i field designates the register, or registers, to be loaded or stored. The j and k fields designate two index registers which are added together to calculate the effective address of the storage information. In the long format the h field is also added in forming the effective address.

All addresses generated by the main processor are considered to be virtual addresses by the mapping mechanism. This mechanism transforms (maps) the virtual address into the address of a physical location in storage. The mapping mechanism deals with 36 bit virtual addresses (ea). The low order 24 bits are called the effective address (eal) and the high order 12 bits are called the key (eak).

The eak is specified by one of four key registers: problem normal key PNK, problem alternate key PAK, supervisory normal key SNK, and supervisory alternate key SAK. Which key is used is defined by the MPM mode, which is either problem or supervisory, and the instruction code, which specifies either normal or alternate. The following table describes the key specification:

	Supervisory Mode	Problem Mode
Normal Key	SNK	PNK
Alternate Key	SAK	PAK

The eal may be computed in two ways. In normal indexing the index quantities are aligned so that the low order bits of each are added together. In true indexing the quantity from X^k is doubled by shifting it left one position prior to addition. The eal addition is computed modulo 2^{24} for both types of indexing.

The following table describes normal and true indexing for both long and short formats:

	Short Format	Long Format
Normal Indexing	eal $\leftarrow X^j + X^k$	eal $\leftarrow X^j + X^k + h$
True Indexing	eal $\leftarrow X^j + 2 \times X^k$	eal $\leftarrow X^j + 2 \times X^k + h$

additions are computed modulo 2^{24}

X^j, X^k = contents of the index registers specified by the j and k fields of the instruction

h = literal field of the instruction

Index load and store operands are 24 bits long. The length of the operands for arithmetic loads and stores is specified in the operation code. Three lengths may be specified: half (24 bits), single (48 bits), and double (96 bits). When loading half word quantities the 24 bit number is expanded to 48 bits when placed in the arithmetic register as follows: if the instruction calls for the left half to be loaded, 0's replace the low order 24 bits of the register; if the right half is loaded, the high order bit of the half word is copied into the high order 24 bits of the register. When storing half word quantities, the selected half of the arithmetic register is stored and the register contents are uneffected.

Index register X^0 is specified to be a source of 0's. To specify single indexing or no indexing, either the j or k field or both should be set to zero. When X^0 is stored, 0's replace the 24-bit storage contents located by the effective address. Information loaded into X^0 is not recoverable from X^0.

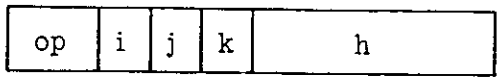
Similarly, A^0 is specified to contain 0's. If A^0 is used as a source in a store operation, the length of the zero quantity stored is determined by the operand length specified in the instruction. Information loaded into A^0 is not recoverable from A^0. If A^0 is specified by the i field of a load arithmetic double instruction, register A^1 is also set to 0's.

Multiple Load and Store

The multiple load operations replace the contents of blocks of successive arithmetic (A) or index (X) registers with information taken from consecutive storage locations. (Register number 0 is considered to be the successor of register number 31.) Storage remains unchanged.

The multiple store operations reverse the process, that is, information from the registers is stored in consecutive storage locations. The registers are unchanged.

The multiple load and store operations have the following format:



For each operation the i-field designates the initial register to be loaded or stored; j gives the number of registers to be loaded or stored; and the modulo 2^{24} sum of index register k and the literal, h, gives the effective address of the first storage location.

Registers X^0 and A^0 are sources of 0's; information loaded into them is not recoverable.

The value of the i-field must be even when X-unit operands are specified. If it is not, the low order bit of the field is forced to 0, exception bit RS is set, and the operation proceeds. The use of the register pair $X^{0,1}$ in multiple load and store instructions results in the loading or storing 24 0's for X^0 and the 24 data bits for X^1.

Exceptions

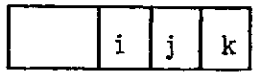
Every virtual address generated for a load or store arithmetic, a load or store arithmetic double, or any multiple load or store, must be divisible by 2. If it is not, the BV (boundary value) exception bit is set to 1, and the operation proceeds using the address minus one as the storage address. Similarly the virtual address for STMZ and STMZA must be divisible by 64. If it is not, the BV bit is set to 1, and the operation proceeds using the address with the seven low order bits forced to 0's as the storage address.

The mapping mechanism checks the validity of all virtual addresses in two ways. First, if the virtual address does not correspond to an actual physical location, a missing address exception occurs and exception bit MA is set to 1. Second, if the virtual address of a store instruction refers to an area to which store access is not permitted, a protected address exception occurs and exception bit PA is set to 1. The setting of the MA or PA exception bit results in a type 2 interruption condition. See the chapter "Interruptions" for further details.

Each storage address generated for multiple load and store operations is individually checked for MA and PA exceptions.

When a double precision A-unit operand is specified by the instruction code, the value of the i-field is assumed to be even. If it is not, the low order bit of the i-field is forced to 0, exception bit RS is set, and the operation proceeds. These fifteen A-unit double precision quantities are specifiable; namely the data in register pairs specified by 2, 4, 6, ... , 30. The double precision quantity specified by A^0 is defined to be 96 0's, so that register A^1 is not the low order half of any double precision quantity.

Load Index (half word format) LXH



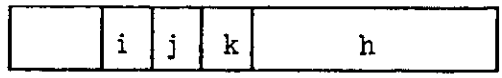
eal ← $X^j + X^k$

eak ← normal key

X^i ← M^{ea}

Exceptions	Exception bit
missing address	MA

Load Index LX



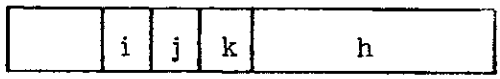
eal ← $X^j + X^k + h$

eak ← normal key

X^i ← M^{ea}

Exceptions	Exception bit
missing address	MA

Load Index per Alternate Key LXA



eal ← $X^j + X^k + h$

eak ← alternate key

X^i ← M^{ea}

This instruction is identical to LX except that in forming the storage address the alternate key is used.

Exceptions	Exception bit
missing address	MA

Store Index (half word format)

STXH

$eal \leftarrow X^j + X^k$

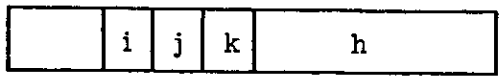
$eak \leftarrow$ normal key

$M^{ea} \leftarrow X^i$

Exceptions	Exception bit
missing address	MA
protected address	PA

Store Index

STX



$eal \leftarrow X^j + X^k + h$

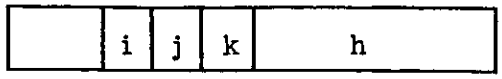
$eak \leftarrow$ normal key

$M^{ea} \leftarrow X^i$

Exceptions	Exception bit
missing address	MA
protected address	PA

Store Index per Alternate Key

STXA



$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ alternate key

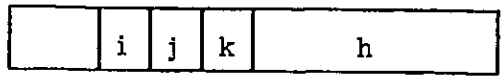
$M^{ea} \leftarrow X^i$

This instruction is identical to STX except that in forming the storage address the alternate key is used.

Exceptions	Exception bit
missing address	MA
protected address	PA

Count to Storage

CNTS



$ea1 \leftarrow X^j + X^k + h$

$eak \leftarrow$ normal key

$X^i \leftarrow M^{ea}$

If $M^{ea} \neq 2^{24}-1$: $M^{ea} \leftarrow M^{ea} + 1$

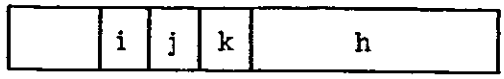
If $M^{ea} = 2^{24}-1$: $M^{ea} \leftarrow M^{ea}$

The contents of the memory location is loaded into X^i. The contents of M^{ea} is treated as an unsigned integer and is incremented by one, modulo 2^{24}. If this would cause M^{ea} to go to zero, the add is suppressed. The fetch from ea and the subsequent storing into it are interlocked so that no intervening accesses are permitted.

Exceptions	Exception bit
missing address	MA
protected address	PA

Swap with Storage

SWS



$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ normal key

If $M^{ea} \neq 0$: $X^i \leftarrow M^{ea}$

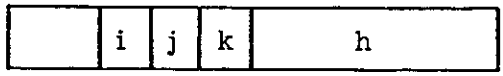
If $M^{ea} = 0$: $X^i \leftarrow M^{ea}$ and $M^{ea} \leftarrow X^i$

The contents of the memory location is loaded into X^i. If the contents of M^{ea} is zero (twenty-four 0's), M^{ea} is replaced by the original contents of X^i. If M^{ea} is different from zero, M^{ea} is not changed. The fetch from M^{ea} and the (potential)subsequent storing into it are interlocked so that no intervening accesses are permitted.

Exceptions	Exception bit
missing address	MA
protected address	PA

Swap with Storage per Alternate Key

SWSA



$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ alternate key

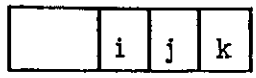
If $M^{ea} \neq 0$: $X^i \leftarrow M^{ea}$

If $M^{ea} = 0$: $X^i \leftarrow M^{ea}$ and $M^{ea} \leftarrow X^i$

This instruction is identical to SWS except that the alternate key is used.

Exceptions	Exception bit
missing address	MA
protected address	PA

Load Arithmetic (half word format) LAH



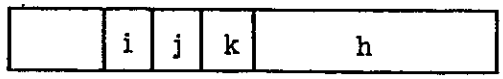
eal ← $X^j + X^k$

eak ← normal key

$A^i \leftarrow M^{ea}$

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Load Arithmetic LA



eal ← $X^j + X^k + h$

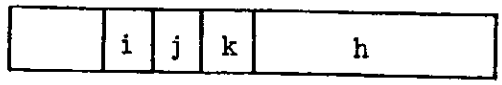
eak ← normal key

$A^i \leftarrow M^{ea}$

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Load Arithmetic per Alternate Key

LAA



$ea1 \leftarrow X^j + X^k + h$

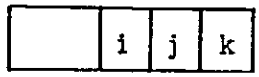
$eak \leftarrow$ alternate key

$A^i \leftarrow M^{ea}$

This instruction is identical to LA except that in forming the storage address the alternate key is used.

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Store Arithmetic (half word format) STAH



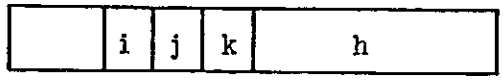
$eal \leftarrow X^j + X^k$

$eak \leftarrow$ normal key

$M^{ea} \leftarrow A^i$

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Store Arithmetic STA



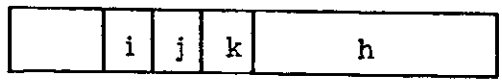
$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ normal key

$M^{ea} \leftarrow A^i$

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Store Arithmetic per Alternate Key STAA



$eal \leftarrow X^j + X^k + h$

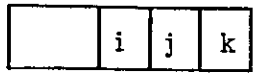
$eak \leftarrow$ alternate key

$M^{ea} \leftarrow A^i$

This instruction is identical to STA except that in forming the storage address the alternate key is used.

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Load Arithmetic Double (half word format) LDH



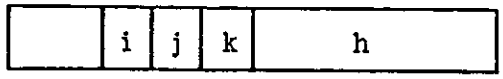
$$eal \leftarrow X^j + X^k$$

$$eak \leftarrow \text{normal key}$$

$$A^{i,i+1} \leftarrow M^{ea,ea+2}$$

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV
i odd	RS

Load Arithmetic Double LD



$$eal \leftarrow X^j + X^k + h$$

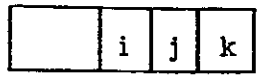
$$eak \leftarrow \text{normal key}$$

$$A^{i,i+1} \leftarrow M^{ea,ea+2}$$

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV
i odd	RS

Store Arithmetic Double
(half word format)

STDH



$eal \leftarrow X^j + X^k$

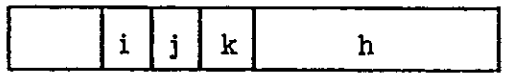
$eak \leftarrow$ normal key

$M^{ea, ea+2} \leftarrow A^{i, i+1}$

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV
i odd	RS

Store Arithmetic Double

STD



$eal \leftarrow X^j + X^k + h$

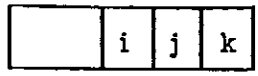
$eak \leftarrow$ normal key

$M^{ea, ea+2} \leftarrow A^{i, i+1}$

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV
i odd	RS

Load Arithmetic, True Indexing
(half word format)

LATH



$eal \leftarrow X^j + 2 \times X^k$

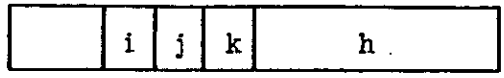
$eak \leftarrow$ normal key

$A^i \leftarrow M^{ea}$

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Load Arithmetic, True Indexing

LAT



$eal \leftarrow X^j + 2 \times X^k + h$

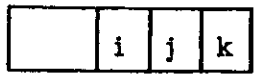
$eak \leftarrow$ normal key

$A^i \leftarrow M^{ea}$

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Store Arithmetic, True Indexing
(half word format)

STATH



$eal \leftarrow X^j + 2 \times X^k$

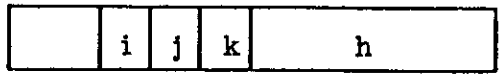
$eak \leftarrow$ normal key

$M^{ea} \leftarrow A^i$

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Store Arithmetic, True Indexing

STAT



$eal \leftarrow X^j + 2 \times X^k + h$

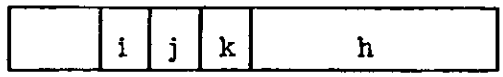
$eak \leftarrow$ normal key

$M^{ea} \leftarrow A^i$

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Load Arithmetic, Left Half

LL



$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ normal key

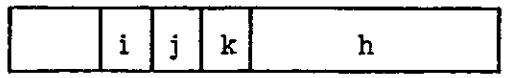
$A^i_{0,1,2,\cdots,23} \leftarrow M^{ea}$

$A^i_{24,25,26,\cdots,47} \leftarrow 0\ [24]$

Exceptions	Exception bit
missing address	MA

Load Arithmetic, Right Half

LR



$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ normal key

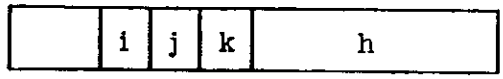
$A^i_{0,1,\cdots,23} \leftarrow M^{ea}_0\ [24]$

$A^i_{24,25,\cdots,47} \leftarrow M^{ea}$

Exceptions	Exception bit
missing address	MA

Store Arithmetic, Left Half

STL



$eal \leftarrow X^j + X^k + h$

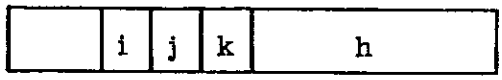
$eak \leftarrow$ normal key

$M^{ea} \leftarrow A^i_{0,1,2,\cdots,23}$

Exceptions	Exception bit
missing address	MA
protected address	PA

Store Arithmetic, Right Half

STR



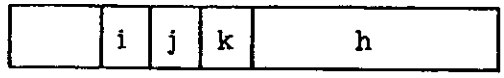
$eal \leftarrow X^j + X^k + h$

$eak \leftarrow$ normal key

$M^{ea} \leftarrow A^i_{24,25,26,\cdots,47}$

Exceptions	Exception bit
missing address	MA
protected address	PA

Load Multiple Index LMX



number of registers loaded ← j

$$eal \leftarrow X^{k} + h$$

eak ← normal key

$$X^{i,i+1} \leftarrow M^{ea,ea+1}$$

$$X^{i+2,i+3} \leftarrow M^{ea+2,ea+3}$$

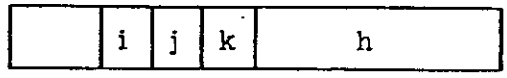
...

$$X^{i+j-2,i+j-1} \leftarrow M^{ea+j-2,ea+j-1}$$

If j is not divisible by 2, the number of registers loaded will be j-1. If j is zero or one, 32 registers will be loaded.

Exceptions	Exceptions
missing address	MA
ea not divisible by 2	BV
i odd	RS

Load Multiple Index per Alternate Key LMXA



number of registers loaded ← j

$$eal \leftarrow X^{k} + h$$

eak ← alternate key

$$X^{i,i+1} \leftarrow M^{ea,ea+1}$$

$$X^{i+2,i+3} \leftarrow M^{ea+2,ea+3}$$

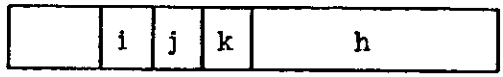
...

$$X^{i+j-2,i+j-1} \leftarrow M^{ea+j-2,ea+j-1}$$

If j is not divisible by 2, the number of registers loaded will be j-1. If j is zero or one, 32 registers will be loaded.

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV
i odd	RS

Store Multiple Index STMX



number of registers stored ← j

$$eal \leftarrow X^{k} + h$$

eak ← normal key

$$M^{ea,\,ea+1} \leftarrow X^{i,\,i+1}$$

$$M^{ea+2,\,ea+3} \leftarrow X^{i+2,\,i+3}$$

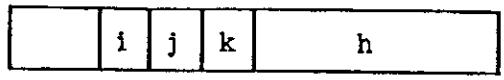
...

$$M^{ea+j-2,\,ea+j-1} \leftarrow X^{i+j-2,\,i+j-1}$$

If j is not divisible by 2, the number of registers stored will be j-1. If j is zero or one, 32 registers will be stored.

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV
i odd	RS

Store Multiple Index per Alternate Key STMXA



number of registers stored ← j

$$eal \leftarrow X^{k} + h$$

eak ← alternate key

$$M^{ea,\,ea+1} \leftarrow X^{i,\,i+1}$$

$$M^{ea+2,\,ea+3} \leftarrow X^{i+2,\,i+3}$$

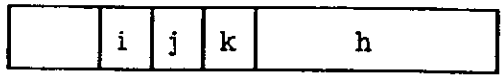
...

$$M^{ea+j-2,\,ea+j-1} \leftarrow X^{i+j-2,\,i+j-1}$$

If j is not divisible by 2, the number of registers stored will be j-1. If j is zero or one, 32 registers will be stored.

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV
i odd	RS

Load Multiple Arithmetic LMA



number of registers loaded ← j

$$eal \leftarrow X^{k} + h$$

eak ← normal key

$$A^{i} \leftarrow M^{ea}$$

$$A^{i+1} \leftarrow M^{ea+2}$$

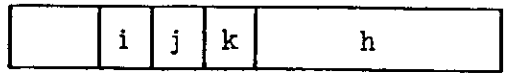
...

$$A^{i+j-1} \leftarrow M^{ea+2j-2}$$

If j is zero, 32 registers will be loaded.

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Load Multiple Arithmetic per Alternate Key LMAA



number of registers loaded ← j

$eal \leftarrow X^{k} + h$

eak ← alternate key

$A^{i} \leftarrow M^{ea}$

$A^{i+1} \leftarrow M^{ea+2}$

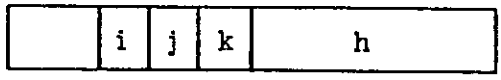
...

$A^{i+j-1} \leftarrow M^{ea+2j-2}$

If j is zero, 32 registers will be loaded.

Exceptions	Exception bit
missing address	MA
ea not divisible by 2	BV

Store Multiple Arithmetic STMA



number of registers stored ← j

$$eal \leftarrow X^{k} + h$$

eak ← normal key

$$M^{ea} \leftarrow A^{i}$$

$$M^{ea+2} \leftarrow A^{i+1}$$

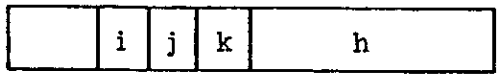
...

$$M^{ea+2j-2} \leftarrow A^{i+j-1}$$

If j is zero, 32 registers will be stored.

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Store Multiple Arithmetic per Alternate Key STMAA



number of registers stored ← j

$$eal \leftarrow X^{k} + h$$

eak ← alternate key

$$M^{ea} \leftarrow A^{i}$$

$$M^{ea+2} \leftarrow A^{i+1}$$

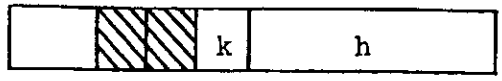
...

$$M^{ea+2j-2} \leftarrow A^{i+j-1}$$

If j is zero, 32 registers will be stored.

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 2	BV

Store Multiple Zeros STMZ



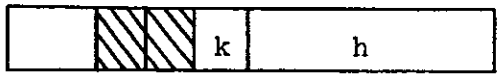
$eal \leftarrow X^{k} + h$

$eak \leftarrow$ normal key

$M^{ea, ea+1, \ldots, ea+63} \leftarrow 0$ [1536]

Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 64	BV

Store Multiple Zeros per Alternate Key STMZA



$eal \leftarrow X^{k} + h$

$eak \leftarrow$ alternate key

$M^{ea, ea+1, \ldots, ea+63} \leftarrow 0$ [1536]

This instruction is identical to STMZ except that the alternate key is used.

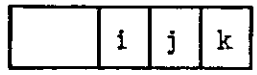
Exceptions	Exception bit
missing address	MA
protected address	PA
ea not divisible by 64	BV

MOVE OPERATIONS

The move operations are for transfering data between registers of two different types. Examples are moves from a special register to an index register or from an index register to an arithmetic register. Most of the instructions involve movement of entire registers or register pairs. However there is a class of move instructions which move single bits to or from the condition register.

Movement of information to or from special registers involve certain interlock considerations which are treated in the section, "Interlocking".

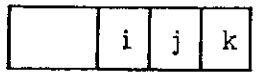
Move Index to Arithmetic MXA



$A^i \leftarrow X^{j,k}$

Exceptions: none

Move Arithmetic to Index MAX



$X^{i,j} \leftarrow A^k$

If i = j, X^i will be set to $A^k_{24,\cdots,47}$.

Exceptions: none

Move Constant to Left Half Arithmetic MKL

$$A^{i}_{0,1,2,\cdots,23} \leftarrow h$$

$$A^{i}_{24,25,26,\cdots,47} \leftarrow 0\ [24]$$

Exceptions: none

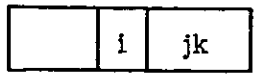
Move Constant to Right Half Arithmetic MKR

$$A^{i}_{24,25,26,\ldots,47} \leftarrow h$$

Note that bits $A^{i}_{0,1,2,\cdots,23}$ are unchanged.

Exceptions: none

Move Location to Index MLX

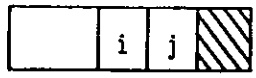


$X^i \leftarrow ia + jk$

The value of ia is the 24-bit storage location of the MLX instruction. The 10-bit literal jk-field is extended to a 24-bit quantity before the addition by appending 14 high-order bits equal in value to the high order bit of the jk-field. The addition is performed modulo 2^{24}.

Exceptions: none

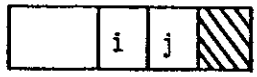
Move Index to Special MXS



$S^i \leftarrow X^j$

Exceptions	Exception bit
$i \geq 3$ and in problem mode	PV

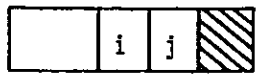
Move Special to Index MSX



$X^i \leftarrow S^j$

Exceptions	Exception bit
$j \geq 3$ and in problem mode	PV

Move Special to Index and Zero MSXZ

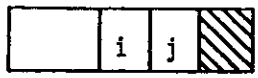


$X^i \leftarrow S^j$

$S^j \leftarrow 0$ [24]

Exceptions	Exception bit
$j \geq 3$ and in problem mode	PV

Move Index to Special by Oring MXSO

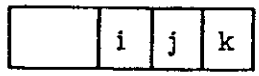


$S^i \leftarrow S^i \vee X^j$

Exceptions	Exception bit
$i \geq 3$ and in problem mode	PV

Move Index Bit to Condition Bit

MXC



$n \leftarrow X^k$

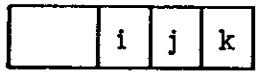
$c_i \leftarrow X^j_n$

If n exceeds 23, c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Move Condition Bit to Index Bit

MCX



$n \leftarrow X^k$

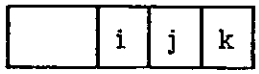
$X^i_n \leftarrow c_j$

If n exceeds 23, no bit is set.

Exceptions: none

Move Arithmetic Bit to Condition Bit

MAC



$n \leftarrow A^k$

$c_i \leftarrow A^j_n$

If n exceeds 47, c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

FLOATING POINT ARITHMETIC

The purpose of the floating point instruction set is to perform calculations using data with a wide range of magnitude and yielding results scaled to preserve precision.

Floating point numbers consist of an exponent E and a fraction F. The quantity expressed by this number is the product of the fraction and the number 2 raised to the power of the exponent, that is:

$$\text{value} = \pm F \times 2^{E}$$

Instruction Format

Most floating point instructions have the following format:

where j and k designate the arithmetic registers containing the source operands and i specifies the result register(s). The remaining floating point operations ignore the k field of the instruction.

The arithmetic registers containing the source operands are not changed as a result of floating point instructions unless they are also specified by the i field to be result registers.

Number Representation

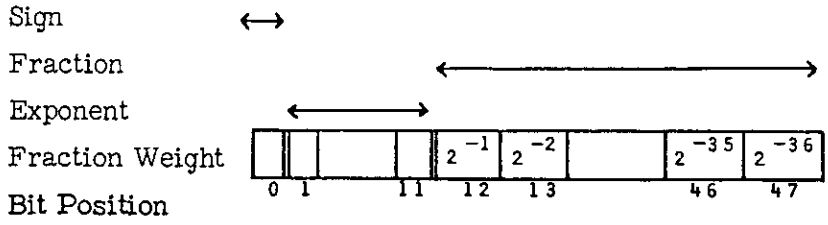
Floating point numbers may be in 48 bit single precision form or in 96 bit double precision form. Single precision numbers may occupy any of the arithmetic registers. Double precision numbers may occupy any even-odd pair of arithmetic registers. Arithmetic register A^0 is specified to be a source of 0's. When A^0 is specified as a source operand, 48 or 96 0's will be provided depending on whether a single or double precision operand was called for by the instruction. If A^0 is specified as the result register, the result will be lost, and the only effect of the operation will be a possible change in the exception register.

Whenever a double precision number is specified by an instruction, the value of the i-, j-, or k-field (as appropriate) is assumed to be even. If it is not, the low order bit of the field is forced to 0, exception bit RS is set, and the operation proceeds. Thus fifteen non-zero double precision quantities are specifiable; namely, the data in register pairs specified by $2, 4, 6, \ldots, 30$. Note that register A^1 is not the low order half of any double precision quantity.

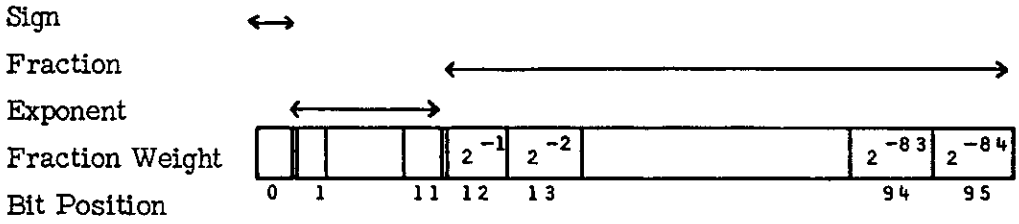
The representation of a floating point number consists of a one bit sign field, an 11 bit exponent field, and a 36 bit or 84 bit fraction field. The sign field occupies positions 0 of the floating point number. A 0 signifies a positive fraction and a 1 a negative fraction. The exponent field occupies positions 1 through 11. The exponent field contains the sum of the exponent, E, and the number 1024, the bias value. The fraction field occupies positions 12 through 47 for a single precision number, and positions 12 through 95 for a double precision number.

Pictorially, with arithmetic bit weights given for the fraction, the formats are:

48 bit floating point number



96 bit floating point number



Number Range

The range of exponents which can be represented is +1023 through -1024. Whenever a floating point operation results in an exponent which cannot be represented in this range, an exception condition exists. If the exponent exceeds +1023, the condition is called overflow. If the exponent is less than -1024, the condition is called underflow. An appropriate exception bit is set to 1 on occurrence of these exceptional conditions.

Figure 3.1 illustrates the range of normalized results possible from a single precision floating point operation. The range of result magnitude is approximately 2.8×10^{-309} to 9.0×10^{307}.

Numbers in the exponent overflow range have the properties of undefined numbers and will be symbolized by u. Furthermore, results in this overflow range are changed to a specific bit configuration: bit zero is set to 1 and the remaining bits are set to 0.

Numbers in the exponent underflow range may be considered to have the properties of the number zero, and are symbolized by 0. Hence, results in the exponent underflow range are changed to the bit configuration which is all 0's.

Since the bit configuration of a leading 1 followed by all 0's represents the u range, special definition of arithmetic using this configuration as an operand is necessary. Figure 3.2 summarizes the results for initial operands as specified. N represents a non-zero value in the valid number range. N* represents a result which is normally in the N range but may be in the u or 0 ranges due to exponent overflow or underflow. Results when using these operands in the compare and sign change operations are given in the sections of this manual dealing with those instructions.

In order to inform the programmer when results approach to the limits of representability, there are overflow and underflow warning bits. These bits are set when a machine with a ten bit exponent would overflow or underflow. The resulting operand is not affected by the setting of these exception bits.

Normalization

A quantity can be represented with the greatest precision by a floating point number, with a given fraction length, when that number is normalized. A floating point number is normalized when it is zero or when $1/2 \leq |F| < 1$. Thus, a non-zero floating point number is normalized if bit 12 (the high order fraction bit) is a 1. The process or normalization consists of shifting the fraction left until the high order fraction bit is a 1, and reducing the exponent by the number of bits shifted.

Instructions are provided which allow floating point arithmetic to be performed with either normalized or unnormalized results. The normalized addition and subtraction instructions yield a normalized result regardless of whether or not the input operands were normalized. The normalized multiplication instructions only guarantee a normalized result if both input operands were normalized. Division may not be done with an unnormalized divisor and does not guarantee a normalized result if the dividend is not normalized.

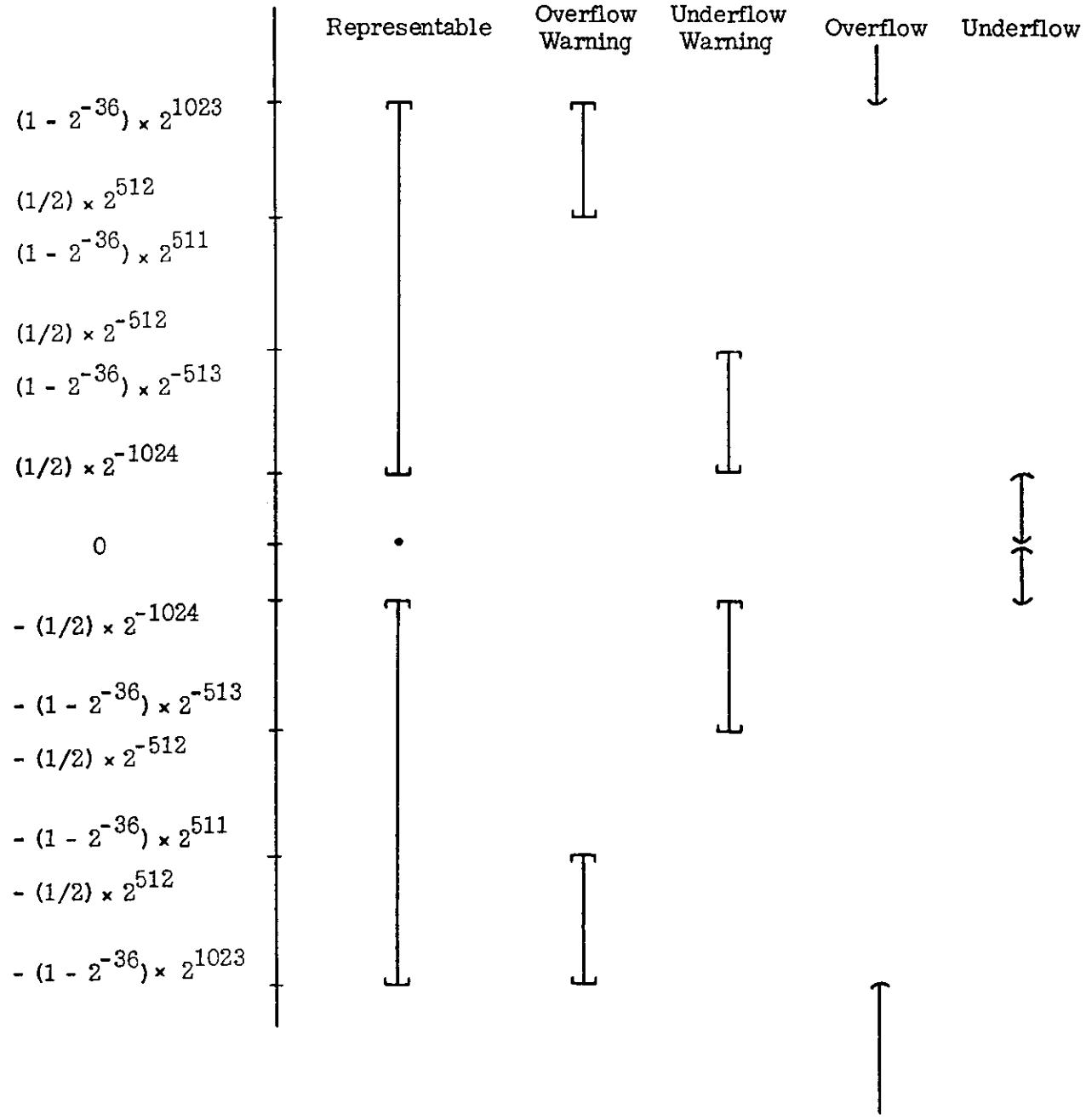


Figure 3.1. Range of Normalized Numbers

Addition Table

Augend	Addend O	Addend N	Addend u
O	O	N	u
N	N	N*	u
u	u	u	u

Multiplication Table

Multiplier	Multiplicand O	Multiplicand N	Multiplicand u
O	O	O	u
N	O	N*	u
u	u	u	u

Division Table

Divisor	Dividend O	Dividend N	Dividend u
O	u	u	u
N	O	N*	u
u	u	u	u

Figure 3.2. Result Ranges

Truncation and Rounding

During the execution of floating point arithmetic operations, low order bits may be truncated in intermediate calculations or when placing the result in the result register. In single precision arithmetic the intermediate result fraction is truncated with the high order 37 bits retained. The low order bit of the retained information is called the guard bit. Normalization, if specified by the instruction, takes place by shifting the intermediate fraction including the guard bit. Following normalization the fraction is truncated to 36 bits and placed in the result register. Double precision normalized arithmetic is similar except the intermediate fraction is truncated with 85 bits retained, and the final result is truncated to 84 bits.

Normalized floating point arithmetic instructions are also provided which specify the result fraction to be statistically rounded. If any of the bits truncated, during either of the truncation processes described above, were a 1, the low order bit of the fraction field is forced to a 1.

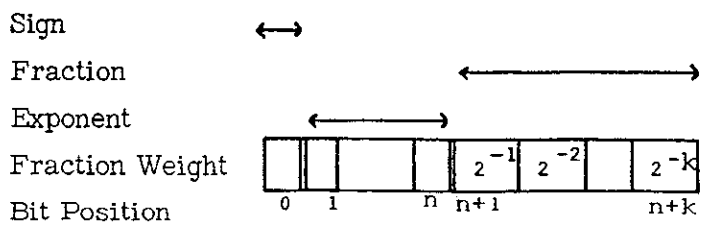
Rounding does not take place if a zero fraction, exponent overflow, or exponent underflow exception occurred.

Low Significance and Zero Fraction

Addition and subtraction may cause a loss of significant bits in cases where the operands are of nearly the same magnitude and differ in sign or when the operands have large numbers of leading 0's, so that the result, before normalization, has a large number of leading 0's. A warning is given when this occurs by setting a "low significance" exception bit to 1. The bit is set to 1 for both single and double precision operations when the leading 1 bit of the result fraction, prior to normalization, is in one of the eight least significant bit positions or is in the guard bit. The low significance exception bit is not set to 1 when the result fraction is all 0's. For this occurrence, a "zero fraction" exception bit is set to 1.

Short Word Floating Point Format

Special provision is made for allowing floating point numbers to be packed into less than 48 bits. In the short floating point format, the sign bit occupies the leading bit position, the exponent field the next n positions (where n is between one and eleven), and the fraction field the remaining positions. The following diagram illustrates this format:



Quantities in this short format are not acceptable operands for arithmetic operations and must be converted to the standard 48 bit format before use. Two special instructions CVF and CVS are provided for converting from short word to full word floating point and vice versa. These instructions which contract and expand the exponent will be described further in the instruction section.

In the short format the exponent, sign, and fraction fields have the same interpretation as in the long format except that the exponent is stored with a bias of 2^{n-1} in the short format (in the regular format, the bias is 2^{10}).

Operation Summary

The following table summarizes the floating point arithmetic instructions:

	Add	Subtract	Multiply	Divide
Normalized, Truncated				
Single	AN	SN	MN	DN
Double	ADN	SDN	MDN	DDN
Mixed	-	-	MMN	DMN
Normalized, Rounded				
Single	AR	SR	MR	DR
Double	ADR	SDR	MDR	DDR
Mixed	-	-	-	DMR
Unnormalized, Truncated				
Single	AU	SU	MU	-
Double	ADU	SDU	MDU	-
Mixed	-	-	MMU	-

Addition and Subtraction Instructions

All the addition and subtraction operations are performed as follows:

The notation 37/85 indicates 37 in the case where the result is to be in single precision and 85 in the case where the result is to be in double precision.

1. If either operand is a u, the result is set to u and the remaining steps of the operations are omitted.

2. If either operand is 48/96 0's, step 3 is omitted. If both of the operands are 48/96 0's, the result is set to 48/96 0's and the remaining steps are omitted.

3. The exponents of the operands in A^j and A^k are compared and the fraction of the operand with the smaller exponent is shifted right a number of positions equal to the difference in exponents. 0's are inserted in the high order vacated bits.

4. If the instruction specifies subtraction, the sign of the second operand is changed.

5. A signed fraction addition then takes place with the high order bit of the shifted fraction aligned with the high order bit of the unshifted fraction. Conceptually, addition takes place before any truncation and the intermediate result is then truncated to 37/85 bits. In fact, truncation takes place first, with sufficient information retained from the truncated bits to make the above property hold.

6. The larger of the two operand exponents is taken as the exponent of the intermediate result.

7. If the fraction addition caused a fraction overflow, the intermediate fraction is shifted right one position and a 1 is inserted in the high order position. One is then added to the intermediate exponent. If this causes the exponent to exceed 1023 an exponent overflow has occurred. The result is then set to u, the AO (add overflow) exception bit is set to 1, and the remaining steps are omitted.

8. If normalization is specified and if the 37/85 bit intermediate result fraction was all 0's, the result is set to zero (i. e. 48/96 0's), the ZF (zero fraction) exception bit is set to 1, and the remaining steps are omitted.

9. If normalization is not specified and the 36/84 bit intermediate result fraction was all 0's, the ZF exception bit is set to 1 and step 10 is omitted.

10. If the high order 28/76 bits of the fraction are all 0's the LS (low significance) exception bit is set to 1.

11. If the instruction specified normalization, the intermediate fraction is now normalized. The intermediate fraction is shifted left until its high order bit is a 1. 0's are inserted into the low order vacated bits. The intermediate exponent is decreased by the amount of the shift. If this causes the exponent to be less than -1024, an exponent underflow has occurred. The AU (add underflow) exception bit is set to 1, the result is set to 48/96 0's and the remaining steps are omitted.

12. If the resulting exponent is greater than 511, the OW (overflow warning) exception bit is set to 1. If the resulting exponent is less than -512, the UW (underflow warning) exception bit is set to 1.

13. This intermediate result fraction is truncated to 36/84 bits.

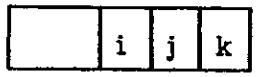
14. If rounding was specified by the operation, the low order bit of the fraction is forced to a 1 if any of the truncated bits, including the bit shifted out in step 7, were a 1.

Add Normalized AN

The contents of register A^i are replaced by the normalized sum of the single precision floating point numbers in A^j and A^k.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF

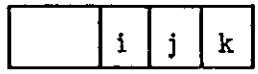
Add Double Normalized ADN



The contents of register pair $A^{i,i+1}$ are replaced by the normalized sum of the double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF
i, j, or k odd	RS

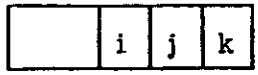
Add Rounded AR



The contents of register A^i are replaced by the normalized and statistically rounded sum of the single precision floating point numbers in A^j and A^k.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF

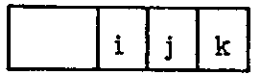
Add Double Rounded ADR



The contents of register pair $A^{i,i+1}$ are replaced by the normalized and statistically rounded sum of the double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF
i, j, or k odd	RS

Add Unnormalized AU



The contents of register A^i are replaced by the unnormalized sum of the single precision floating point numbers in A^j and A^k.

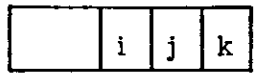
Exceptions	Exception bit
result exponent $>$ +1023	AO
+511 $<$ result exponent $\leq$ +1023	OW
low significance	LS
zero fraction	ZF

Add Double Unnormalized ADU

The contents of register pair $A^{i,i+1}$ are replaced by the unnormalized sum of the double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$.

Exceptions	Exception bit
result exponent $>$ +1023	AO
+511 $<$ result exponent $\leq$ +1023	OW
low significance	LS
zero fraction	ZF
i, j, or k odd	RS

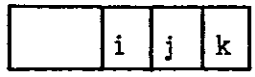
Subtract Normalized SN



The contents of register A^i are replaced by the normalized result of the subtraction of the single precision floating point number in A^k from the single precision floating point number in A^j.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF

Subtract Double Normalized SDN



The contents of register pair $A^{i,i+1}$ are replaced by the normalized result of the subtraction of the double precision floating point number in $A^{k,k+1}$ from the double precision floating point number in $A^{j,j+1}$.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF
i, j, or k odd	RS

Subtract Rounded SR

The contents of register A^i are replaced by normalized and statistically rounded result of the subtraction of the single precision floating point number in A^k from single precision floating point number in A^j.

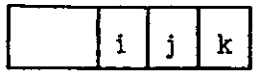
Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF

Subtract Double Rounded SDR

The contents of register pair $A^{i,i+1}$ are replaced by the normalized and statistically rounded result of the subtraction of the double precision floating point number in $A^{k,k+1}$ from the double precision floating point number in $A^{j,j+1}$.

Exceptions	Exception bit
result exponent > +1023	AO
result exponent < -1024	AU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
low significance	LS
zero fraction	ZF
i, j, or k odd	RS

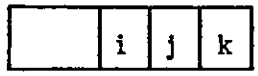
Subtract Unnormalized SU



The contents of register A^i are replaced by the unnormalized result of the subtraction of the single precision floating point number in A^k from the single precision floating point number in A^j.

Exceptions	Exception bit
result exponent > +1023	AO
+511 < result exponent ≤ +1023	OW
low significance	LS
zero fraction	ZF

Subtract Double Unnormalized SDU



The contents of register pair $A^{i,i+1}$ are replaced by the unnormalized result of the subtraction of the double precision floating point number in $A^{k,k+1}$ from the double precision floating point number in $A^{j,j+1}$.

Exceptions	Exception bit
result exponent > +1023	AO
+511 < result exponent ≤ +1023	OW
low significance	LS
zero fraction	ZF
i, j, or k odd	RS

Multiplication Instruction

The multiplication operations are performed as follows:

1. If either operand is u, the result is set to u and the remaining steps of this operation are omitted.

2. If either operand is 48/96 0's, the result is set to zero and the remaining steps are omitted.

3. If either operand is unnormalized and the instruction calls for normalization, the UO is set to 1.

4. The exponents of the operands are added to form an intermediate exponent.

5. The 36/84-bit fraction of the operand in A^j and the 36/84-bit fraction of the operand in A^k are multiplied to form a 72/168-bit product which is then truncated with the high order 37/85 bits being retained.

6. If normalization was called for, the high order bit of the intermediate fraction is compared to 0. If it is a 0, a left shift of one position takes place and the intermediate exponent is decreased by one. This is sufficient to normalize the result if the operands were normalized.

7. If the intermediate exponent is greater than 1023, the MO (multiply overflow) exception bit is set to 1 and the result is set to u. The remaining steps are omitted.

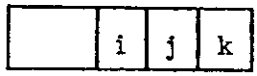
8. If the intermediate exponent is less than -1024, the MU (multiply underflow) exception bit is set to 1 and the result is set to 48/96 0's. The remaining steps are omitted.

9. If the exponent is greater than 511, the OW bit is set to 1. If it is less than -512, the UW bit is set to 1.

10. The intermediate fraction is truncated to 36/84 bits.

11. If rounding is called for and if any of the truncated bits were a 1, the low order bit of the fraction is set to 1.

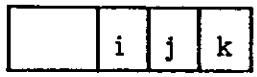
Multiply Normalized MN



The contents of register A^i are replaced by the normalized product of the single precision floating point numbers in A^j and A^k.

Exceptions	Exception bit
result exponent $>$ +1023	MO
result exponent $<$ -1024	MU
+511 $<$ result exponent $\leq$ +1023	OW
-512 $>$ result exponent $\geq$ -1024	UW
unnormalized operand	UO

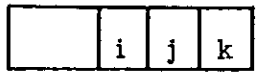
Multiply Double Normalized MDN



The contents of register pair $A^{i,i+1}$ are replaced by the normalized product of the double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$.

Exceptions	Exception bit
result exponent $>$ +1023	MO
result exponent $<$ -1024	MU
+511 $<$ result exponent $\leq$ +1023	OW
-512 $>$ result exponent $\geq$ -1024	UW
unnormalized operand	UO
i, j, or k odd	RS

Multiply Rounded MR



The contents of register A^i are replaced by the normalized and statistically rounded product of the single precision floating point numbers in A^j and A^k.

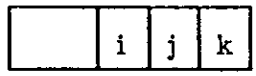
Exceptions	Exception bit
result exponent > +1023	MO
result exponent < -1024	MU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized operand	UO

Multiply Double Rounded MDR

The contents of register pair $A^{i,i+1}$ are replaced by the normalized and statistically rounded product of the double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$.

Exceptions	Exception bit
result exponent > +1023	MO
result exponent < -1024	MU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized operand	UO
i, j, or k odd	RS

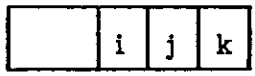
Multiply Unnormalized MU



The contents of register A^i are replaced by the unnormalized product of the single precision floating point numbers in A^j and A^k.

Exceptions	Exception bit
result exponent > +1023	MO
result exponent < -1024	MU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW

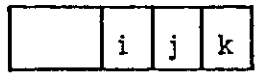
Multiply Double Unnormalized MDU



The contents of register pair $A^{i,i+1}$ are replaced by the unnormalized product of the double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$.

Exceptions	Exception bit
result exponent > +1023	MO
result exponent < -1024	MU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
i, j, or k odd	RS

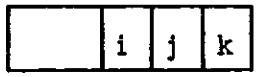
Multiply Mixed Normalized MMN



The contents of register pair $A^{i,i+1}$ are replaced by the double precision normalized product of the single precision floating point numbers in A^j and A^k.

Exceptions	Exception bit
result exponent > +1023	MO
result exponent < -1024	MU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
i odd	RS

Multiply Mixed Unnormalized MMU



The contents of register pair $A^{i,i+1}$ are replaced by the double precision unnormalized product of the single precision floating point numbers in Aj and A^k.

Exceptions	Exception bit
result exponent > +1023	MO
result exponent < -1024	MU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
i odd	RS

Division Instructions

The division operations are performed as follows:

1. If either operand is a u, the result is set to a u and the remaining steps are omitted.

2. If the fraction of the divisor is zero, the result is set to u, the DO bit is set to 1, and the remaining steps are omitted.

3. If the divisor is unnormalized, the result is set to u, the UD bit is set to 1, and the remaining steps are omitted.

4. If the fraction part of the dividend is zero, the result is set to zero, and the remaining steps are omitted.

5. The exponent of the divisor is subtracted from the exponent of the dividend to form an intermediate exponent.

6. The 36/84 bit dividend fraction is divided by the 36/84 bit divisor fraction to form the intermediate quotient fraction. Conceptually the quotient is computed to infinite precision and then truncated to 36/84 bits.

7. Since the divisor was normalized, the intermediate quotient fraction must be less than 2.0. If it is greater than or equal to 1.0, it is shifted right one position, a 1 is inserted in the high order position, and the intermediate exponent is increased by one. If the dividend was normalized, the fraction is now normalized.

8. If the intermediate exponent is greater than 1023, the result is set to u, the DO exception bit is set to 1, and the remaining steps are omitted.

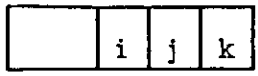
9. If the intermediate exponent is less than -1024, the result is set to all 0's, the DU exception bit is set to 1, and the remaining steps are omitted.

10. If the exponent is greater than 511, the OW bit is set to 1. If the exponent is less than -512, the UW bit is set to 1.

11. If rounding was specified and any of the bits truncated in steps 6 or 7 was a 1, the low order bit of the fraction is forced to a 1.

12. The sign of the result, determined by the rules of algebra, together with the intermediate exponent and fraction form the result.

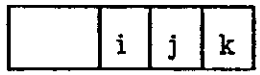
Divide Normalized DN



The contents of register A^i are replaced by the quotient formed by dividing the single precision floating point dividend in A^j by the single precision floating point divisor in A^k.

Exceptions	Exception bit
result exponent > +1023	DO
result exponent < -1024	DU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized divisor	UD
divisor fraction = 0	DO

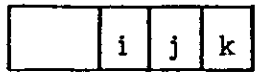
Divide Double Normalized DDN



The contents of register pair $A^{i,i+1}$ are replaced by the quotient formed by dividing the double precision floating point dividend in $A^{j,j+1}$ by the double precision floating point divisor in $A^{k,k+1}$.

Exceptions	Exception bit
result exponent > +1023	DO
result exponent < -1024	DU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized divisor	UD
divisor fraction = 0	DO
i, j, or k odd	RS

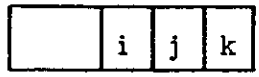
Divide Rounded DR



The contents of register A^i are replaced by the statistically rounded quotient formed by dividing the single precision floating point dividend in A^j by the single precision floating point divisor in A^k.

Exceptions	Exception bit
result exponent > +1023	DO
result exponent < -1024	DU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized divisor	UD
divisor fraction = 0	DO

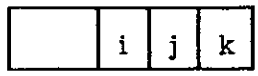
Divide Double Rounded DDR



The contents of register pair $A^{i,i+1}$ are replaced by the statistically rounded quotient formed by dividing the double precision floating point dividend in $A^{j,j+1}$ by the double precision floating point divisor in $A^{k,k+1}$.

Exceptions	Exception bit
result exponent > +1023	DO
result exponent < -1024	DU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized divisor	UD
divisor fraction = 0	DO
i, j, or k odd	RS

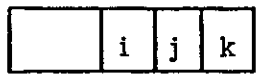
Divide Mixed Normalized DMN



The contents of register A^i are replaced by the quotient formed by dividing the double precision floating point dividend in $A^{j,j+1}$ by the single precision floating point divisor in A^k.

Exceptions	Exception bit
result exponent > +1023	DO
result exponent < -1024	DU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized divisor	UD
divisor fraction = 0	DO
j odd	RS

Divide Mixed Rounded DMR

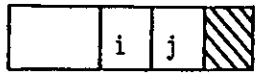


The contents of register A^i are replaced by the statistically rounded quotient formed by dividing the double precision floating point dividend in $A^{j,j+1}$ by the single precision floating point divisor in A^k.

Exceptions	Exception bit
result exponent > +1023	DO
result exponent < -1024	DU
+511 < result exponent ≤ +1023	OW
-512 > result exponent ≥ -1024	UW
unnormalized divisor	UD
divisor fraction = 0	DO
j odd	RS

Miscellaneous Instructions

Round RND



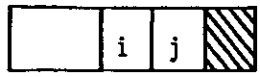
The double precision floating point number is register pair $A^{j,j+1}$ is rounded to form a single precision number which replaces the contents of A^i.

This round which is a "true round", rather than the statistical round which was described earlier, is performed as follows:

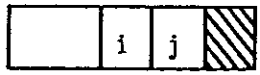
1. If the input operand is a double precision representation of u, the result is set to a single precision u and the remaining steps are omitted.
2. The exponent of the input forms the intermediate exponent.
3. If bit A^{j+1}_0 (i. e. bit 37 of the 84 bit fraction) is a 1, the magnitude of the fraction of A^j is increased by 2^{-36} to form an intermediate fraction.
4. If the addition in step 3 caused a fraction overflow, the intermediate fraction is shifted right one position, with its low order bit lost, and a 1 is inserted into the high order position of the fraction. Then the intermediate exponent is increased by one. If this causes an exponent overflow, the AO exception bit is set to 1, the result is set to u, and the remaining steps are omitted.
5. If the intermediate fraction is 36 0's, the result is set to 48 0's and the ZF exception bit is set to 1, and the remaining steps are omitted.
6. If there are no 1's in the high order 28 bits of the fraction, the LS bit is set to 1.
7. If the intermediate exponent is greater than 511 the OW exception bit is set to 1.
8. The sign of the input together with the intermediate exponent and fraction form the result.

Exceptions	Exception bit
result exponent $>$ +1023	AO
+511 $<$ result exponent $\leq$ +1023	OW
low significance	LS
zero fraction	ZF
j odd	RS

Set Positive, Floating SPF



Set Negative, Floating SNF

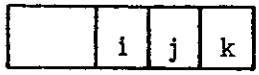


The 47 low-order bits of A^i are replaced by the 47 low-order bits of A^j. Bit A^i_0 is set to 0 for SPF; it is set to 1 for SNF.

If A^j is the u configuration, then A^i is set to the u configuration. If A^j is true zero (all 0's), then A^i is set to true zero.

Exceptions: none

Convert to Short Floating CVS



The single precision floating point number in A^j is converted to a short floating point number and replaces the contents of A^i. The length, n, of the exponent field of the short floating point number is given by the 5-bit 2's complement literal in the k field.

The conversion is performed as follows:

1. If n is negative, zero, or greater than eleven, the result is set to a u, the ILO (illegitimate operand) exception bit is set to 1, and the remaining steps are omitted.
2. If the input is u, the result is set to u, and the remaining steps are omitted.
3. If the input is 48 0's, the result is set to 48 0's and the remaining steps are omitted.
4. The 46 bits A^j_2, A^i_3, ..., A^j_{47} are shifted left 11-n positions. 0's fill the vacated positions. If any of the bits shifted out through A^j_2 are the same as A^j_1, this indicates that the exponent cannot be represented in 11-n bits. The SO bit is then set to 1, the result is set to u, and the remaining steps are omitted.
5. The two unshifted bits A^j_0, A^j_1 together with the 46 bits of the shifted quantity form the result.

Exceptions	Exception bit
$n > 11$	ILO
$n < 1$	ILO
unrepresentable exponent	SO

Convert to Full Floating CVF

The short, left justified, floating point number in A^j is converted to a single precision floating point number and replaces the contents of A^i. The length n, of the exponent field of the short floating point number is given by the 5-bit 2's complement literal in the k field.

The conversion is performed as follows:

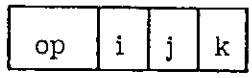
1. If n is negative, zero, or greater than eleven, the result is set to u, the ILO exception bit is set to 1, and the remaining steps are omitted.
2. If the input is a u, the result is set to u, and the remaining steps omitted.
3. If the input is 48 0's, the result is set to 48 0's and the remaining steps are omitted.
4. The 46 bits A^j_2, A^j_3, ..., A^j_{47} are shifted right 11-n positions. The vacated bit positions are filled with the complement of the bit A^j_i.
5. The two unshifted bits A^j_0, A^j_1 together with the high order 46 bits from the shifted quantity form the result.

Exceptions	Exception bit
$n > 11$	ILO
$n < 1$	ILO

INTEGER ARITHMETIC

Instructions and facilities are included in the integer arithmetic class of operations which allow elementary arithmetic to be performed on single length and multiple length operands.

The integer arithmetic instructions have the following format:



where j and k designate the arithmetic registers containing the source operands and i specifies the result register(s).

The instructions are divided into three groups:

1. The single length integer arithmetic operations (add, subtract, multiply and divide) operate on 48-bit operands and yield a 48-bit result. In each operation the specified function is performed between two arithmetic registers A^j and A^k. The result replaces the 48-bit contents of arithmetic register A^i. The contents of A^j and A^k are not changed.

2. Mixed length integer arithmetic operations (multiply and divide) have a 96-bit operands or 96-bit result. The 96-bit operand occupies an even-odd pair of 48-bit arithmetic registers.

3. The "continued" integer arithmetic operations (high order continued add, high order continued subtract, low order continued add, and low order continued subtract) use an implied third operand. In each operation the specified function is performed between the single precision integer contents of arithmetic registers A^j and A^k and the 3-bit contents of Multiprecision Carry Register (MPC). The result replaces the contents of A^i and MPC. Registers A^j and A^k are not changed.

Register A^0 is specified to be a source of 0's. When A^0 is specified as a source operand, 48 or 96 0's will be provided depending on whether a single or mixed length instruction is used. If A^0 is specified as the result register, the result will be lost, and the only effect of the operation will be a possible change in the Multiprecision Carry Register or the exception register.

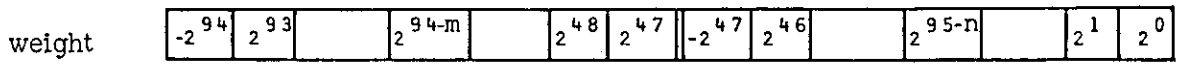
Number Representation

Integer operands are represented in two's-complement form. The formats are as follows:

Single Precision Integer

for $1 \leq m \leq 47$

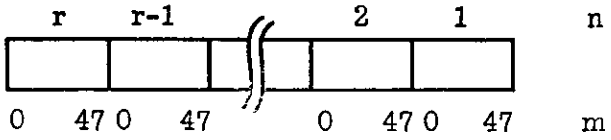
Double Precision Integer



for $1 \leq m \leq 47$
$49 \leq n \leq 95$

Multiple Precision Integer

Multi-length operands are a multiple of 48 bits in length and have the format:



$$I = \sum_{m,n} b_m^n \, w_{m,n}$$

where I is the value of the integer operand of multiplicity r; b_m^n is the binary value of the m^{th} bit of the n^{th} 48-bit register; and $w_{m,n}$ is the weighting factor of bit b_m^n as follows:

for $m = 0$ $\quad w_{0,n} = -2^{47n}$

for $m \neq 0$ $\quad w_{m,n} = 2^{47n-m}$

Note that the single and double precision integer formats can be considered special cases of the above format with r equal to 1 and 2, respectively.

Standard Form

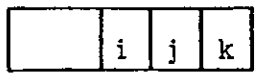
In the multi-length format the magnitude of the weights of bit 0 of register n and bit 47 of register n+1 are the same, but the signs are different. Thus the combination 00 has the same value as the combination 11. A "standard form" is defined to circumvent this non-unique format. A number is defined to be in standard form if bit zero of every register, except the high order register, is equal to 0. Except for a few numbers near the negative limit of the representable range, all numbers have a standard form. For example, the number which has a 1 in bit zero of every word and 0's elsewhere has no standard form.

Multiprecision addition and multiplication can be performed on operands without requiring the operands to be in standard form. The operation will yield a result in standard form. The subtrahend in multiprecision subtraction must be in standard form. The requirement for multiprecision dividends and divisors to be in standard form is dependent on the programming algorithm to be used. The 96-bit product formed in mixed length multiply is in standard form and mixed length divide requires the 96-bit dividend to be in standard form.

Overflow

The integers which can be represented in 48 bits in two's-complement form range from -2^{47} to $2^{47}-1$. Wherever a single length add, subtract, or multiply; a mixed length multiply or divide; a high order continued add or subtract; or an arithmetic shift results in a number which cannot be represented in the satisfactory range of -2^{47} to $2^{47}-1$, an integer overflow condition exists and the appropriate exception bit is set to 1.

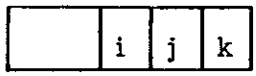
Add Integer AI



The contents of register A^i are replaced by the low order 48 bits of the sum formed by the addition of the single precision integers in A^j and A^k.

Exception	Exception bit
result $> 2^{47}-1$	AO
result $< -2^{47}$	AO

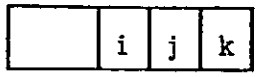
Subtract Integer SI



The contents of register A^i are replaced by the low order 48 bits of the difference formed by the subtraction of the single precision integer in A^k from the single precision integer in A^j.

Exception	Exception bit
result $> 2^{47}-1$	AO
result $< -2^{47}$	AO

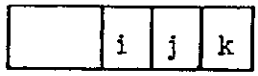
Multiply Integer MI



The contents of A^i are replaced by the low order 48 bits of the product of the single precision integers in A^j and A^k.

Exception	Exception bit
result $> 2^{47}-1$	MO
result $< -2^{47}$	MO

Multiply Integer, Mixed Length MMI



The contents of register pair $A^{i,i+1}$ are replaced by the product in standard form of the single precision integers in A^j and A^k.

In the special case where A^j and A^k are both -2^{47} the product cannot be represented by a double precision integer; then $A^{i,i+1}$ are set to 0's and the MO exception bit is set to 1.

Exception	Exception bit
result $> 2^{94}-1$	MO
i odd	RS

Integer Divide Instructions

The integer divide instructions are performed as follows:

1. If the divisor is zero, the result is set to zero, the divide overflow exception bit (DO) is set to 1, and the remaining steps are omitted.

2. If the divide is a DMI and the dividend is not in standard form, the ILO exception bit is set to 1, and the operation proceeds as if the dividend was in standard form.

3. The dividend is divided by the divisor to form an exact quotient.

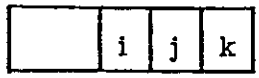
4. If the exact quotient is an integer, it forms the intermediate result.

5. If the exact quotient was not an integer, the integer part of the exact quotient forms the intermediate result. That is, the exact quotient is rounded toward zero.

6. If the intermediate result cannot be represented in 48 bits, the divide overflow exception bit is set to 1.

7. The result is set to the low order 48 bits of the intermediate result.

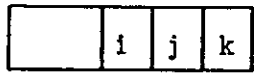
Divide Integer DI



The contents of A^i are replaced by the single precision integer quotient formed by dividing the single precision integer dividend in A^j by the single precision integer divisor in A^k.

Exceptions	Exception bit
$A^k = 0$	DO
result $> 2^{47}-1$	DO

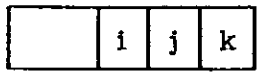
Divide Integer, Mixed Length DMI



The contents of register A^i are replaced by the single precision integer quotient formed by dividing the double precision integer dividend in $A^{j,j+1}$ by the single precision integer divisor in A^k.

Exceptions	Exception bit
$A^k = 0$	DO
result $> 2^{47}-1$	DO
result $< -2^{47}$	DO
$A^{j+1}_0 = 1$	ILO
j odd	RS

Continued Add, Low Order ACL

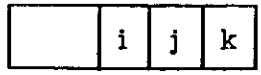


The contents of $A^i_{1,\ldots,47}$ are replaced by the low order 47 bits of the sum of the integers in A^j and A^k and the contents of the Multiprecision Carry Register MPC. A^i_0 is set to 0. The MPC is set so that:

$$2^{47} \times MPC_{new} + A^i = A^j + A^k + MPC_{old}$$

Exception	Exception bit
$MPC_{old} \neq -3, -2, -1, 0,$ or $+1$	ILO

Continued Subtract, Low Order SCL



The contents of $A^i_{1,\ldots,47}$ are replaced by the low order 47 bits formed by subtracting the integer in A^k from the integer in A^j and adding the contents of MPC to the difference. A^i_0 is set to 0. The MPC is set so that:

$$2^{47} \times MPC_{new} + A^i = A^j - A^k + MPC_{old}$$

Exception	Exception bit
$MPC_{old} \neq -3, -2, -1,$ or 0	ILO

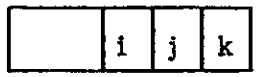
Continued Add, High Order ACH

The contents of A^i are replaced by the low order 48 bits of the sum formed by adding the integers in A^j and A^k and the contents of MPC. The MPC is then set to zero.

Except for the participation of MPC, this instruction is identical to AL.

Exceptions	Exception bit
result $< -2^{47}$	AO
result $> 2^{47}-1$	AO
$MPC_{old} \neq -3,-2,-1,0,$ or $+1$	ILO

Continued Subtract, High Order SCH



The contents of A^i are replaced by the low order 48 bits of the difference formed by subtracting the integer in A^k from the integer in A^j and adding the contents of MPC to the result. The MPC is then set to zero.

Except for the participation of MPC, this instruction is identical to SL.

Exceptions	Exception bit
result $< -2^{47}$	AO
result $> 2^{47}-1$	AO
$MPC_{old} \neq -3,-2,-1,$ or 0	ILO

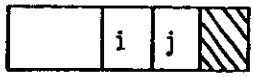
Set Positive, Integer SPI

The contents of register A^i are replaced by the absolute value of the integer represented by the contents A^j.

If the integer in A^j is -2^{47}, an overflow exception condition exists. The integer add overflow exception bit AO is set to 1, and the contents of A^i are replaced by the value zero.

Exception	Exception bit
$A^j = -2^{47}$	AO

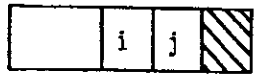
Set Negative, Integer SNI



The contents of register A^i are replaced by the negative of the absolute value of the integer represented by the contents of A^j.

Exceptions: none

Convert to Normalized CVN

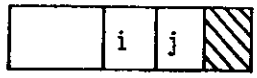


The single precision integer in A^j is converted to a normalized single precision floating point number; the floating point number replaces the contents of A^i.

If the conversion results in an intermediate fraction length greater than 36 bits, the fraction is truncated to 36 bits.

Exceptions: none

Convert to Integer CVI



The single precision floating point number in A^j is converted to a single precision integer which replaces the contents of A^i.

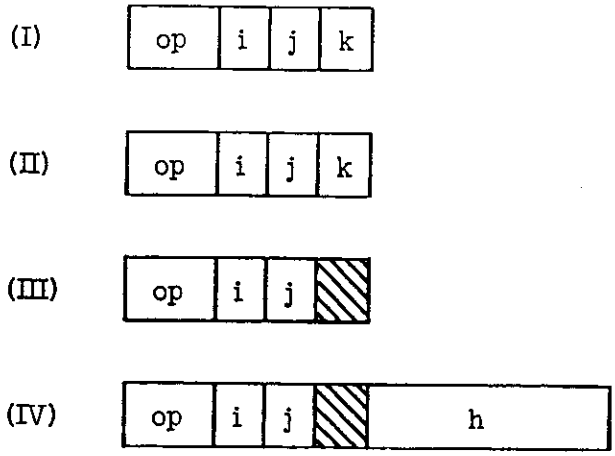
If the absolute value of the number in A^j is greater than $2^{48}-1$ or if it is u, the conversion will not be done correctly. In this case the AO exception bit is set to 1 and no meaning should be given to the result.

Exception	Exception bit
$\lvert A^j \rvert > 2^{48}-1$	AO
$A^j = u$	AO

INDEX ARITHMETIC

The index arithmetic instruction set performs binary arithmetic on operands serving as addresses, index quantities, and counts. Except in literal index arithmetic, operands are 24 bits long. In literal index arithmetic one of the operands is contained in the instruction and is either 5 or 24 bits long.

The index arithmetic instructions have the following formats:



Operations are performed between index register X^j and X^k (format I), between X^j and the 5-bit k-literal (format II), or between X^j and the 24-bit h-literal (format IV). When format III is used, the single operand is X^j. The result replaces the contents of register X^i. Four index divide operations also replace the contents of X^{i+1} with part of its result. Three add-and-test operations set condition bit c_i and X^j. Except for these three the contents of X^j and X^k are not changed.

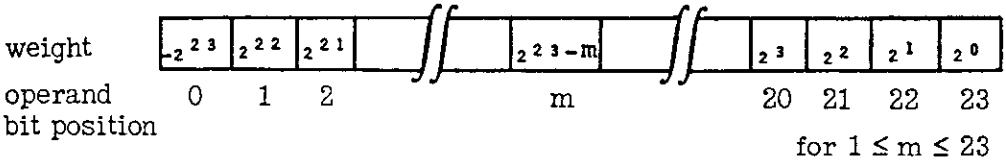
Index register X^0 is identically equal to zero. Since the contents of X^0 is always zero, results which are placed in X^0 are not recoverable.

Number Representation

The number representation used for 24-bit index quantities has the property that the operands and results for the instructions add, subtract, and multiply can be interpreted either as 2's complement integers in the range -2^{23} to $2^{23}-1$ or as positive integers, modulo 2^{24}, the range 0 to $2^{24}-1$. These forms are termed "signed" and "unsigned" index integers. Since signed divide is different from unsigned divide two sets of divide instructions are provided. Similarly both signed and unsigned compare instructions are provided where necessary; see Section 6.

2's Complement Arithmetic

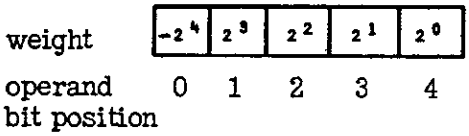
In 2's complement arithmetic the bits of the 24-bit index operands and results have the following arithmetic weights:



Numbers in this format are called signed index integers.

The integers which can be represented in 2's complement form in 24 bits range from -2^{23} to $2^{23}-1$. No special indication is given if a result is outside this range (overflows).

The 5-bit literal in instruction format II is considered as an integer in 2's complement form. Thus the bits have the following arithmetic weights:

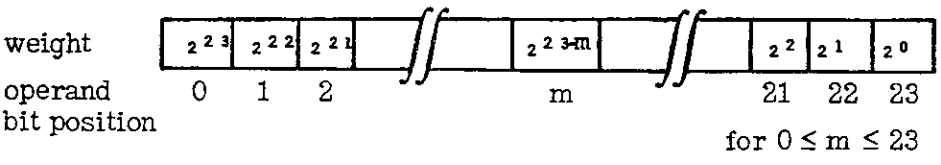


Thus the range of representable literal values is -2^4 to 2^4-1.

Before participating in an operation the 5-bit quantity is extended to a 24-bit quantity by appending 19 high order bits equal in value (i. e. 0 or 1) to the high order bit of the k field. This operation transforms the 5-bit literal to a 24-bit signed index integer, and does not change the value of the number.

Modulo 2^{24} Arithmetic

In modulo 2^{24} arithmetic the bits of the 24-bit index operands and results have the following arithmetic weights:

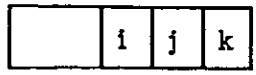


Numbers in this format are called unsigned index integers.

The integers which can be represented in modulo 2^{24} form in 24 bits range from 0 to $2^{24}-1$; and by definition of modulo 2^{24} arithmetic all results are within this range.

The instructions with format II can be used in modulo 2^{24} arithmetic. If the high order bit of the k field is 0, the representable literal values are in the range 0 to $2^{4}-1$. If the high order bit of the k field is 1, the representable literals are in the range $2^{24}-16$ to $2^{24}-1$.

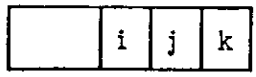
Add Index AX



The contents of X^i are replaced by the low order 24 bits of the sum formed by the addition of the index integers in X^j and X^k.

Exceptions: none

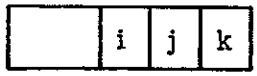
Subtract Index SX



The contents of X^i are replaced by the low order 24 bits of the difference formed by the subtraction of the index integer in X^k from X^j.

Exceptions: none

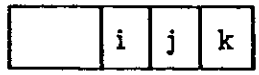
Multiply Index MX



The contents of X^i are replaced by the low order 24 bits of the product of the index integers in X^j and X^k.

Exceptions: none

Divide with Remainder, Index DRX



The contents of X^i is replaced by the signed quotient formed by dividing the signed index integer dividend in X^j by the signed index integer divisor in X^k, and the contents of X^{i+1} are replaced by the signed remainder. The value of the i-field is assumed to be even; if it is not, the low order bit of the i-field is forced to 0, bit RS is set, and the operation proceeds.

The signed index divide is performed as follows:

1. If the divisor is zero, both the quotient and remainder are set to zero, the index divide by zero exception bit (XDZ) is set to 1, and the remaining steps are omitted.

2. If the dividend is -2^{23} and the divisor is -1 (so that the true quotient 2^{23} cannot be represented), the quotient is set to zero, the remainder is set to -2^{23}, and the remaining steps are omitted.

3. The dividend is divided by the divisor to form an exact quotient.

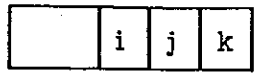
4. If the exact quotient is an integer, it forms the result quotient. The result remainder is zero.

5. If the exact quotient was not an integer, the integer part of the exact quotient forms the result quotient. That is, the exact quotient is rounded toward zero. The remainder is defined as:

$$\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor})$$

Exceptions	Exception bit
$X^k = 0$	XDZ
i odd	RS

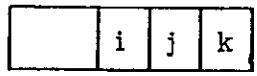
Divide Index DX



The contents of X^i is replaced by the signed quotient formed by the division of the signed index integer dividend in X^j by the signed index integer divisor in X^k. The quotient is defined as in DRX.

Exceptions	Exception bit
$X^k = 0$	XDZ

Remainder Index RX



The contents of X^i is replaced by the signed remainder formed by the division of the signed index integer dividend in X^j by the signed index integer divisor in X^k. The remainder is defined as in DRX.

Exceptions	Exception bit
$X^k = 0$	XDZ

Divide with Remainder, Unsigned Index DRUX

The contents of X^i is replaced by the unsigned quotient formed by dividing the unsigned index integer dividend in X^j by the unsigned index integer divisor in X^k, and the contents of X^{i+1} are replaced by the unsigned remainder. The value of the i-field is assumed to be even; if it is not, the lower order bit of the i-field is forced to 0, bit RS is set, and the operation proceeds.

The unsigned index divide is performed as follows:

1. If the divisor is zero, both the quotient and remainder are set to zero, the index divide by zero exception bit (XDZ) is set to 1, and the remaining steps are omitted.

2. The dividend is divided by the divisor to form an exact quotient.

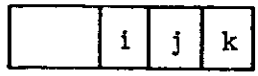
3. If the exact quotient is an integer, it forms the result quotient. The result remainder is zero.

4. If the exact quotient was not an integer, the integer part of the exact quotient forms the result quotient. That is, the exact quotient is rounded toward zero. The remainder is defined as:

remainder = dividend - (quotient × divisor)

Exception	Exception bit
$X^k = 0$	XDZ
i odd	RS

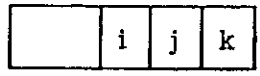
Divide Unsigned Index DUX



The contents of X^i is replaced by the quotient formed by the division of the unsigned index integer dividend in X^j by the unsigned index integer divisor in X^k. The quotient is defined as in DRUX.

Exception	Exception bit
$X^k = 0$	XDZ

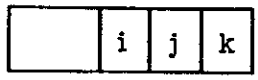
Remainder Unsigned Index RUX



The contents of X^i is replaced by the remainder formed by the division of the unsigned index integer dividend in X^j by the unsigned index integer divisor in X^k. The remainder is defined as in DRUX.

Exception	Exception bit
$X^k = 0$	XDZ

Add Index to Short Constant AXC



The contents of X^i are replaced by the low order 24 bits of the sum formed by the addition of the 24 bit number in X^j and the number in the literal k-field. The 5-bit k-field is extended to a 24-bit quantity before the addition by appending 19 high-order bits equal in value to the high order bit of the k-field.

Exceptions: none

Add Index to Constant AXK

The contents of X^i are replaced by the low order 24 bits of the sum formed by the addition of the index integers in X^j and the literal h-field.

Exceptions: none

Multiply Index by Constant MXK

The contents of X^i are replaced by the low order 24 bits of the product of the index integers in X^j and in the literal h-field.

Exceptions: none

Divide with Remainder Index by Constant DRXK

The signed index integer in X^j is divided by the signed index integer divisor in the literal h-field. The signed index integer quotient replaces the contents of X^i and the signed index integer remainder replaces the contents of X^{i+1}. The value of the i-field is assumed to be even.

The quotient, remainder, and exception are defined as in DRX.

Exception	Exception bit
h-field = 0	XDZ
i odd	RS

Divide Index by Constant DXK

The signed index integer dividend in X^j is divided by the signed index integer divisor in the literal h-field. The signed index integer quotient replaces the contents of X^i.

The quotient and exception are defined as in DRX.

Exception	Exception bit
h-field = 0	XDZ

Remainder Index by Constant RXK

The signed index integer dividend in X^j is divided by the signed index integer divisor in the literal h-field. The signed index integer remainder replaces the contents of X^i.

The remainder and exception are defined as in DRX.

Exception	Exception bit
h-field = 0	XDZ

Divide with Remainder Unsigned Index by Constant DRUXK

The unsigned index integer in X^j is divided by the unsigned index integer divisor in the literal h-field. The unsigned index integer quotient replaces the contents of X^i and the unsigned index integer remainder replaces the contents of X^{i+1}. The value of the i-field is assumed to be even.

The quotient, remainder, and exception are defined as in DRUX

Exceptions	Exception bit
h-field = 0	XDZ
i odd	RS

Divide Unsigned Index by Constant DUXK

The unsigned index integer dividend in X^j is divided by the unsigned index integer divisor in the literal h-field. The unsigned index integer quotient replaces the contents of X^i.

The quotient and exception are defined as in DRUX.

Exception	Exception bit
h-field = 0	XDZ

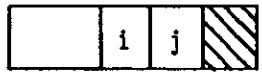
Remainder Unsigned Index by Constant RUXK

The unsigned index integer dividend in X^j is divided by the unsigned index integer divisor in the literal h-field. The unsigned index integer remainder replaces the contents of X^i.

The remainder and exception are defined as in DRUX.

Exception	Exception bit
h-field = 0	XDZ

Set Positive, Index SPX

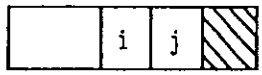


The contents of register X^i are replaced by the absolute value of the signed index integer in X^j.

If the integer in X^j is -2^{23}, the contents of X^i are replaced by the value zero.

Exceptions: none

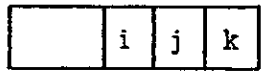
Set Negative, Index SNX



The contents of register X^i are replaced by the negative of the absolute value of the signed index integer in X^j.

Exceptions: none

Add Index and Test AXT

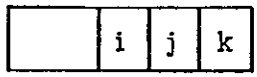


The contents of X^j are replaced by the low-order 24 bits of the sum formed by the addition of the signed index integers in X^j and X^k.

If the original value of X^j_0 is different from the new X^j_0, condition bit c_i is set to 1; otherwise c_i is set to 0. Thus the condition bit is set to 1 when the addition causes the sign of the index integer in X^j to change (with zero considered positive).

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Add Index to Short Constant and Test AXCT

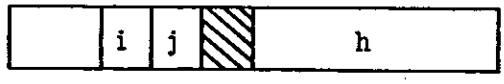


The contents of X^j are replaced by the low-order 24 bits of the sum formed by the addition of the signed index integer in X^j and the signed integer in the literal k-field. The 5-bit k-field is extended to a 24-bit quantity before the addition by appending 19 high-order bits equal in value to the high order bit of the k-field.

If the original value of X^j_0 is different from the new X^j_0, condition bit c_i is set to 1; otherwise, c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Add Index to Constant and Test AXKT



The contents of X are replaced by the low order 24 bits of the sum formed by the addition of the signed index integers in X^j and the literal h-field.

If the original value of X^j_0 is different from the new X^j_0, condition bit c_i is set to 1; otherwise, c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

COMPARE OPERATIONS

Compare instructions are provided to test specified relations between two numeric quantities and to provide byte testing capabilities.

The effect of the compare instructions is to set a bit called a condition bit. Twenty-four condition bits are provided and are grouped together to form special register S^0. The individual condition bits are identified as $c_0, c_1, \ldots, c_{23}$.

The compare instructions have the following formats:

(I)

(II)

The compare is done between the contents of registers R^j and R^k in format I and between the contents register X^j and the literal h in format II. In both formats the i field designates the bit (or bits) of the condition register which is to be set.

If a compare contains an i field greater than 23, that is, specifies a nonexistent condition bit, the result of the compare is lost. However, if an attempt is made to set c_{24} to 0, or c_{25} to 1, the condition check exception signal CC is generated.

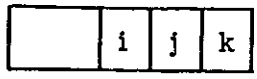
Although only two basic numerical comparison relations are provided in the instruction set (greater than or equal to, and equal to), all six possible relations can be tested either by interchanging the names in the j- and k-fields or by using the negation of the test result. Specifically:

Basic relation to test	Test for	Basic relation true if condition bit has value
$a > b$	$b \geq a$	0
$a \geq b$	$a \geq b$	1
$a = b$	$a = b$	1
$a \neq b$	$a = b$	0
$a \leq b$	$b \geq a$	1
$a < b$	$a \geq b$	0

In the floating point arithmetic section a bit configuration is defined to represent floating point numbers in the exponent overflow range. These numbers are symbolized by u and have the configuration of a 1 in bit zero and 0's in the remaining bits. When one or both operands are u in any of the floating point comparison operations, the result of the compare is made false (0).

The floating point compare operations may give an incorrect result if either or both operands are unnormalized. If either operand is unnormalized, the UO (unnormalized operand) exception bit is set to 1.

Compare, Greater or Equal, Normalized CGEN



The normalized single precision floating point numbers in A^j and A^k are compared. If the number in A^j is greater than or equal to the number in A^k, condition bit c_i is set to 1; otherwise c_i is set to 0.

For the special case when either or both operands are u, condition bit c_i is set to 0.

This instruction may give an incorrect result if either or both operands are unnormalized.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC

Compare, Equal, Normalized CEQN

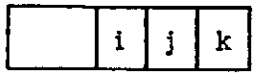
The normalized single precision floating point numbers in A^j and A^k are compared. If the numbers are equal, condition bit c_i is set to 1; otherwise c_i is set to 0.

For the special case when either or both operands are u, condition bit c_i is set to 0.

This instruction may give an incorrect result if either or both operands are unnormalized.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC

Compare, Greater or Equal, Double CGED

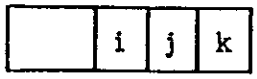


The normalized double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$ are compared. If the number in $A^{j,j+1}$ is greater than or equal to the number in $A^{k,k+1}$, condition bit c_i is set to 1; otherwise c_i is set to 0. The values of the j- and k-fields are assumed to be even.

For the special case when either or both operands are u, condition bit c_i is set to 0.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC
j or k odd	RS

Compare, Equal, Double CEQD

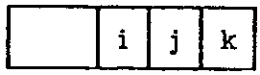


The normalized double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$ are compared. If the numbers are equal, condition bit c_i is set to 1; otherwise c_i is set to 0. The values of the j- and k-fields are assumed to be even.

For the special case when either or both operands are u, condition bit c_i is set to 0.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC
j or k odd	RS

Compare Magnitude, Greater or Equal, Normalized — CMGEN



The magnitudes of the normalized single precision floating point numbers in A^j and A^k are compared. If the magnitude of the number in A^j is greater than or equal to the magnitude of the number in A^k, condition bit c_i is set to 1. Otherwise c_i is set to 0.

For the special case when either or both operands are u, condition bit c_i is set to 0.

This instruction may give an incorrect result if either or both operands are unnormalized.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC

Compare Magnitude, Equal, Normalized — CMEQN

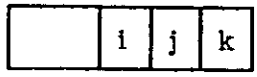
The magnitudes of the normalized single precision floating point numbers in A^j and A^k are compared. If the magnitudes of the numbers are equal, condition bit c_i is set to 1. Otherwise c_i is set to 0.

For the special case when either or both operands are u, condition bit c_i is set to 0.

This instruction may give an incorrect result if either or both operands are unnormalized.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC

Compare Magnitude Double, Greater or Equal — CMGED

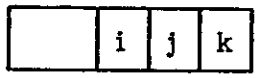


The magnitudes of the normalized double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$ are compared. If the magnitudes of the number in $A^{j,j+1}$ is greater than or equal to the magnitude of the number in $A^{k,k+1}$, condition bit c_i is set to 1. Otherwise, c_i is set to 0. The values of the j- and k-fields are assumed to be even.

For the special case when either or both operands are u, condition bit c_i is set to 0.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC
j or k odd	RS

Compare Magnitude Double, Equal — CMEQD

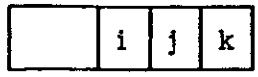


The magnitudes of the normalized double precision floating point numbers in $A^{j,j+1}$ and $A^{k,k+1}$ are compared. If the magnitudes of the numbers are equal, condition bit c_i is set to 1. Otherwise, c_i is set to 0. The values of the j- and k-fields are assumed to be even.

For the special case when either or both operands are u, condition bit c_i is set to 0.

Exceptions	Exception bit
unnormalized operand	UO
c_{24} set to 0 or c_{25} set to 1	CC
j or k odd	RS

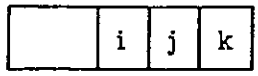
Compare, Greater or Equal,Integer CGEI



The single precision integers in A^j and A^k are compared. If the number in A^j is greater than or equal to the number in A^k, condition bit c_i is set to 1; otherwise c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare, Equal, Integer CEQI



The single precision integers in A^j and A^k are compared. If the numbers are equal, condition bit c_i is set to 1; otherwise c_i is set to 0.

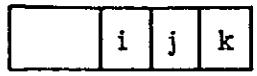
Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Unsigned, Greater or Equal, Integer CUGEI

The contents of registers A^j and A^k are considered as 48-bit unsigned integers. If the number in A^j is greater than or equal to the number in A^k, condition bit c_i is set to 1; otherwise c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

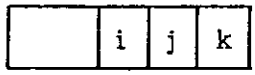
Compare, Greater or Equal, Index CGEX



The index integers in X^j and X^k are compared. If the number in X^j is greater than or equal to the number in X^k, condition bit c_i is set to 1; otherwise c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

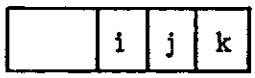
Compare, Equal, Index CEQX



The index integers in X^j and X^k are compared. If the numbers are equal, condition bit c_i is set to 1; otherwise c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Unsigned, Greater or Equal, Index CUGEX



The contents of registers X^j and X^k are considered as 24-bit unsigned integers. If the number in X^j is greater than or equal to the number in X^k, condition bit c_i is set to 1; otherwise c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Index with Constant, Greater or Equal — CGEXK

The index integers in X^j and in the literal h-field are compared. If the number in X^j is greater than or equal to the number in the h-field, condition bit c_i is set to 1; otherwise, c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Index with Constant, Equal — CEQXK

The index integers in X^j and in the literal h-field are compared. If the numbers are equal, condition bit c_i is set to 1; otherwise, c_i is set to 0.

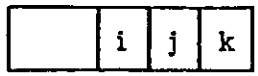
Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Unsigned Index with Constant, Greater or Equal — CUGEXK

The contents of register X^j and the literal h-field are considered as 24-bit unsigned integers. If the number in X^j is greater than or equal to the number in the h-field, condition bit c_i is set to 1; otherwise c_i is set to 0.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Bytes, Arithmetic CBA

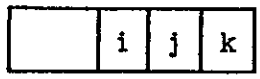


The 48-bit contents of register A^k are considered as six 8-bit operand bytes: the first byte is $A^k_{0,1,\ldots,7}$; the second byte is $A^k_{8,9,\ldots,15}$; and so on. The low order 8 bits of register A^j are considered as one test byte. The test byte is compared with each of the six operand bytes.

Condition bit c_i is set to 1 if the test byte is identical to one or more of the operand bytes; it is set to 0 if the test byte is not identical to any operand byte.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Bytes, Multiple, Arithmetic CBMA



The 48-bit contents of register A^k are considered as six 8-bit operand bytes: the first byte is $A^k_{0,1,\ldots,7}$; the second byte is $A^k_{8,9,\ldots,15}$; and so on. The low order 8 bits of register A^j are considered as one test byte. The test byte is compared with each of the six operand bytes.

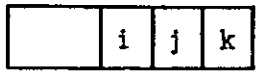
Condition bit c_i is set to 1 if the test byte is identical to one or more of the operand bytes; it is set to 0 if the test byte is not identical to any operand byte.

Condition bit c_{i+1} is set to 1 or 0 according as the test byte is identical to the first operand byte or not; bit c_{i+2} is set to 1 or 0 according as the test byte is identical to the second operand byte or not; and so on through bit c_{i+6}.

Only the leading two bits of the i-field of the instruction are interpreted to determine which condition bits are set, thereby effectively partitioning the condition register into segments: bits 0 to 6, bits 8 to 14, bits 16 to 22, and bits 24 to 30.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Bytes, Index CBX

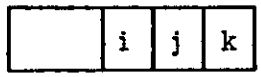


The 24-bit contents of register X^k are considered as three 8-bit operand bytes: the first byte is $X^k_{0,1,\ldots,7}$; the second byte is $X^k_{8,9,\ldots,15}$; and so on. The low order 8 bits of register X^j are considered as one test byte. The test byte is compared with each of the three operand bytes.

Condition bit c_i is set to 1 if the test byte is identical to one or more of the operand bytes; it is set to 0 if the test byte is not identical to any operand byte.

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

Compare Bytes, Multiple, Index CBMX



The 24-bit contents of register X^k are considered as three 8-bit operand bytes: the first byte is $X^k_{0,1,\ldots,7}$; the second byte is $X^k_{8,9,\ldots,15}$; and so on. The low order 8 bits of register X^j are considered as one test byte. The test byte is compared with each of the three operand bytes.

Condition bit c_i is set to 1 if the test byte is identical to one or more of the operand bytes; it is set to 0 if the test byte is not identical to any operand byte.

Condition bit c_{i+1} is set to 1 or 0 according as the test byte is identical to the first operand byte or not; bit c_{i+2} is set to 1 or 0 according as the test byte is identical to the second operand byte or not; and bit c_{i+3} is set to 1 or 0 according as the test byte is identical to the third operand byte or not.

Only the leading three bits of the i-field of the instruction are interpreted to determine which condition bits are set, thereby effectively partitioning the condition register into segments: bits 0 to 3, bits 4 to 7, and so on.

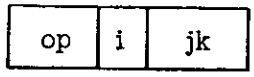
Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

SHIFT OPERATIONS

There are three functionally different shift operations: logical shift, insert field, and integer shift. Within the logical and integer shift classes either single or double length operands may be used. There also are two ways of specifying the direction and amount of shift: either directly from a literal field in the instruction or indirectly by the contents of a register.

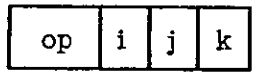
Shift Amount and Direction

When the shift amount is specified by the literal field of the instruction, the following instruction format is used:



The 10-bit literal jk-field is interpreted as a 2's complement integer, so that numbers in the range -512 to +511 are representable.

When the shift amount is specified by the contents of a register, the following instruction format is used:



The contents of register A^k or X^k is interpreted as a 2's complement integer. Only the low order 10 bits are used to specify the shift amount; the remaining bits are ignored.

The integer specifies both the direction and amount of the shift. Its absolute value specifies the amount of the shift. Its sign indicates the shift direction: a positive integer specifies a left shift, a negative integer specifies a right shift.

For the insert field instructions register A^k or X^k contains three 8-bit parameters.

Source and Result Operands

When the shift is specified by the literal field, the i-field specifies both the source and result operands; that is,

Source operand: R^i or $R^{i,i+1}$

Result operand: R^i or $R^{i,i+1}$

(where R may be interpreted as either X or A).

When the shift amount is specified by R^k, the i- and j-fields specify the source and result operands as follows:

Source operand: R^j or $R^{j,j+1}$

Result operand: R^i or $R^{i,i+1}$

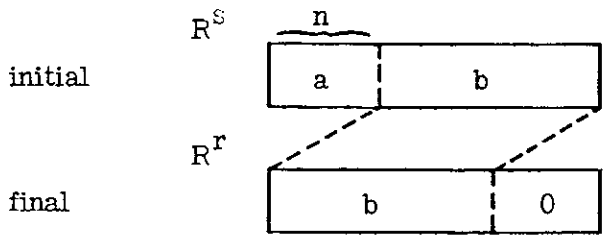
In the explanations, R^s is used to indicate the source register, and R^r the result register. The shift amount is denoted by n. The notation 48/24 is to be interpreted as 48 for the A-unit shift instructions and 24 for the X-unit shift instructions.

Logical Shift, Single Register

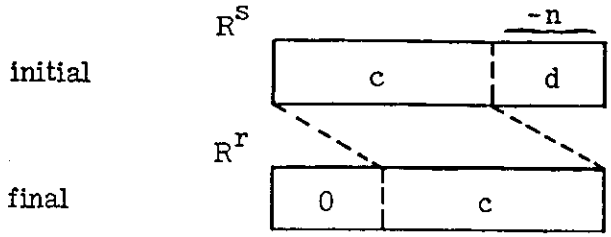
The contents of register R^s are shifted left or right the specified number of bit positions. The direction of the shift is determined by the sign of the shift amount. Bits which are shifted out of R^s are lost; vacated positions are filled with 0's. The 48/24-bit shifted quantity then replaces the contents of register R^r. The contents of register R^s are unchanged unless due to the operation type or the specification of the i and j field, R^s is the same register as R^r. If the shift amount is greater than or equal to 48/24, register R^r is set to 0's.

Pictorially the logical shift, single register, instructions are:

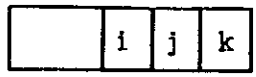
single shift left



single shift right



Logic Shift, Arithmetic SHA

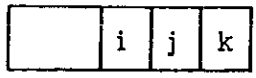


shift amount ← $A^{k}_{38,\ldots,47}$

A^{i} ← logic shift (A^{j})

Exceptions: none

Logic Shift, Index SHX

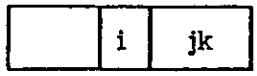


shift amount ← $X^{k}_{14,\ldots,23}$

X^{i} ← logic shift (X^{j})

Exceptions: none

Logic Shift, by Constant, Arithmetic SHAC

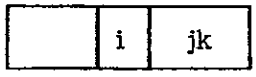


shift amount ← jk

A^{i} ← logic shift (A^{i})

Exceptions: none

Logic Shift, by Constant, Index SHXC



shift amount ← jk

X^{i} ← logic shift (X^{i})

Exceptions: none

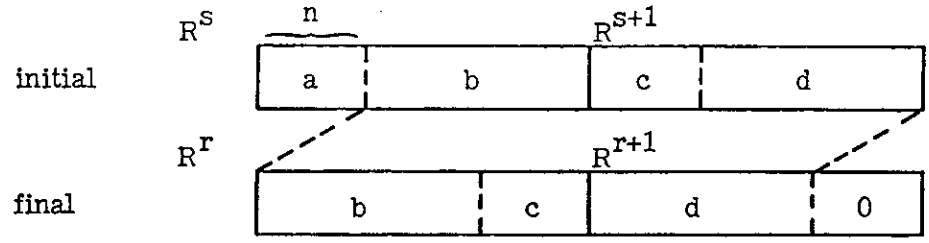
Logical Shift, Double Registers

Registers R^s and R^{s+1} are coupled and are considered as one 96/48 bit quantity. This 96/48 bit quantity is shifted left or right the specified number of bit positions to form an intermediate result. The direction of the shift is determined by the sign of the shift quantity. Bits which are shifted out are ignored; vacated positions are filled with 0's. The 96-bit intermediate result then replaces the contents of registers R^r and R^{r+1}.

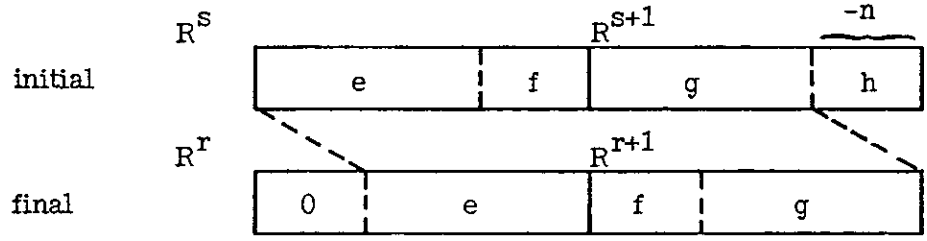
If the shift amount is greater than or equal to 96/48, registers R^r and R^{r+1} are set to 0's.

Pictorially, the logical shift, double registers, instructions are as follows:

double shift left



double shift right



The value of s must be even. If it is not, the low order bit specifying s is forced to 0, exception bit RS is set, and the operation proceeds.

Logic Shift, Double Arithmetic

SHD

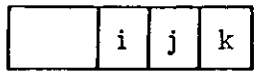
shift amount ← $A^{k}_{38,\ldots,47}$

$A^{i,i+1}$ ← logic shift ($A^{j,j+1}$)

Exception	Exception bit
i or j odd	RS

Logic Shift, Double Index

SHDX



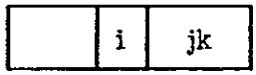
shift amount ← $X^{k}_{14,\ldots,23}$

$X^{i,i+1}$ ← logic shift ($X^{j,j+1}$)

Exception	Exception bit
i or j odd	RS

Logic Shift by Constant, Double Arithmetic

SHDC



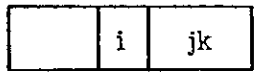
shift amount ← jk

$A^{i,i+1}$ ← logic shift ($A^{i,i+1}$)

Exception	Exception bit
i odd	RS

Logic Shift by Constant, Double Index

SHDXC



shift amount ← jk

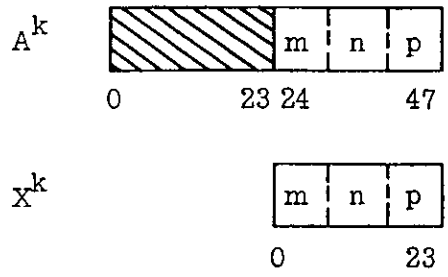
$X^{i,i+1}$ ← logic shift ($X^{i,i+1}$)

Exception	Exception bit
i odd	RS

This page has been deleted.

Insert Field

Register R^k supplies three 8-bit integer parameters m, n, and p. These parameters are packed in R^k as shown:



The contents of register R^j are rotated left m positions. Bits rotated out of position 0 are inserted into position 47/23.

The p-n bits of this rotated quantity numbered n, n+1, n+2,..., p-1 are then inserted into the corresponding bits of register R^i.

The remaining bits of R^i (namely those numbered 0, 1, 2,..., n-1 and p, p+1, p+2,..., 47/23) either are left unaltered for the instructions IFX and IFA or are set to 0's for the instructions IFZX and IFZA. The contents of R^j and R^k are not changed.

The parameter m is interpreted as a positive integer modulo 48/24. The normal ranges for the positive integer parameters n and p are:

$$0 \leq n \leq 47/23$$

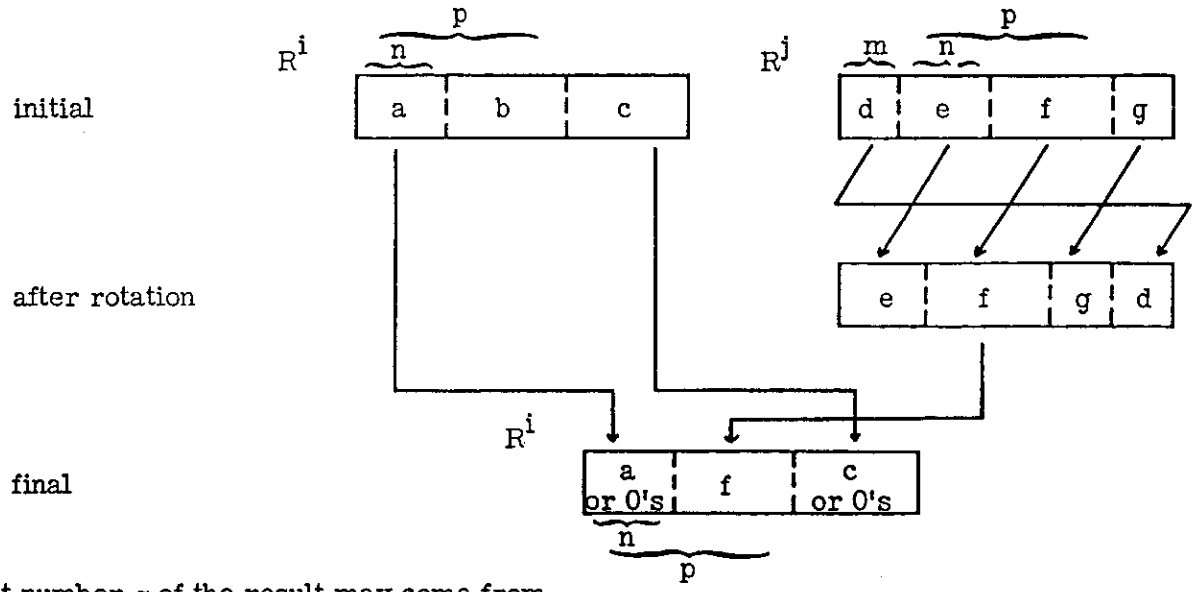
$$1 \leq p \leq 48/24$$

$$n < p$$

If $p \geq 49/25$, the operation proceeds as if $p = 48/24$. If $n \geq 48/24$ or if $p = 0$ or if $n = p$, the contents of R^i are left unaltered for IFA and IFX or are set to 0's for IFZA and IFZX.

If $n > p$, the n - p bits of R^i numbered p, p+1,..., n-1 are set to 0's; the remaining bits are left unaltered for IFA and IFX or are also set to 0's for IFZA and IFZX.

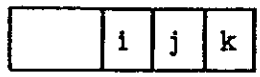
Pictorially the insert field instructions are as follows:



Bit number α of the result may come from

(1) a source of 0's
or (2) bit α of operand R^i
or (3) bit $\alpha + m$ (mod 48/24) of operand R^j

Insert Field, Arithmetic IFA

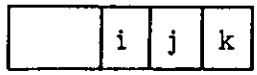


insertion parameters $\leftarrow A^{k}_{24,25,\ldots,47}$

$A^{i} \leftarrow \text{insert}(A^{i}, A^{j})$

Exceptions: none

Insert Field, Index IFX

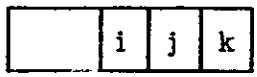


insertion parameters $\leftarrow X^{k}$

$X^{i} \leftarrow \text{insert}(X^{i}, X^{j})$

Exceptions: none

Insert Field and Zero, Arithmetic IFZA

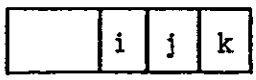


insertion parameters $\leftarrow A^{k}_{24,25,\ldots,47}$

$A^{i} \leftarrow \text{insert}(0, A^{j})$

Exceptions: none

Insert Field and Zero, Index IFZX



insertion parameters $\leftarrow X^{k}$

$X^{i} \leftarrow \text{insert}(0, X^{j})$

Exceptions: none

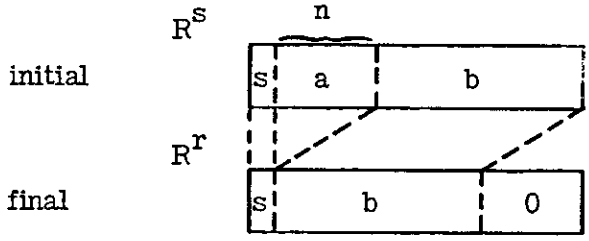
Integer Shift, Single Register

The contents of the last 47/23 positions of register R^S are shifted left or right the specified number of positions to form an intermediate result; position R^S_0 is not shifted. If the shift is to the right, bit values equal to R^S_0 are supplied to the vacated high-order positions; low-order bits are shifted out and ignored. If the shift is to the left, 0's are supplied to the vacated low-order positions; high-order bits are shifted out and are lost; however if the instruction is SIA or SIAC, and if one or more of the bits which is shifted out is unequal to A^S_0, the shift overflow exception bit SO is set to 1. The shifted quantity then replaces the contents of register R^r, bit R^r_0 being set to the value of R^S_0. The contents of register R^S are unchanged unless the operation type of the specification of the i and j field result in R^r being the same register as R^S.

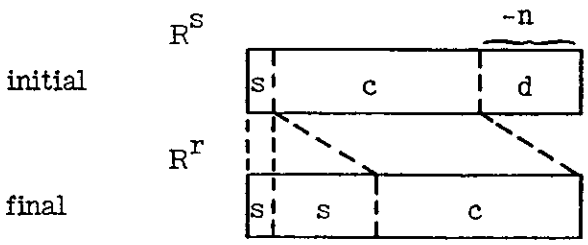
If the shift amount is greater than or equal to 47/23, the low order 47/23 bits of R^r are set to 0's for a left shift, or to the value of R^S_0 for a right shift.

Pictorially the integer shift, single register, instructions are:

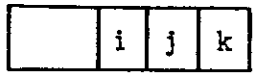
single shift left



single shift right



Integer Shift, Arithmetic SIA

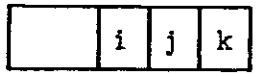


Shift amount ← $A^{k}_{38,\ldots,47}$

A^{i} ← integer shift (A^{j})

Exception	Exception bit
bit different from A^{j}_{0} shifted out during left shift	SO

Integer Shift, Index SIX

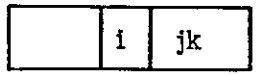


Shift amount ← $X^{k}_{14,\ldots,23}$

X^{i} ← integer shift (X^{j})

Exceptions: none

Integer Shift by Constant, Arithmetic SIAC

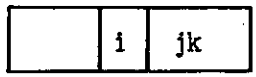


Shift amount ← jk

A^i ← integer shift (A^i)

Exception	Exception bit
bit different from A^i_0 shifted out during left shift	SO

Integer Shift by Constant, Index SIXC



Shift amount ← jk

X^i ← integer shift (X^i)

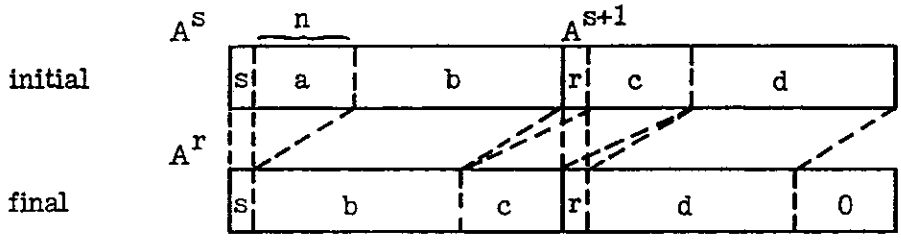
Exceptions: none

Integer Shift, Double Register

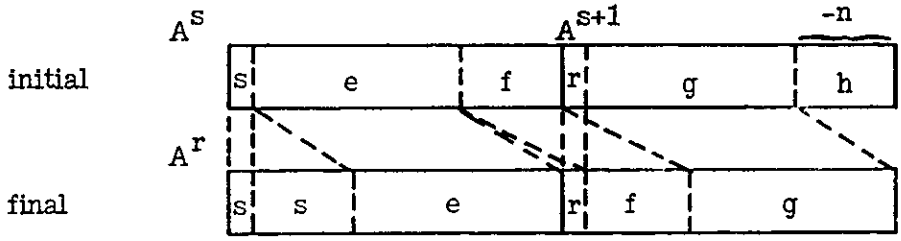
Registers A^s and A^{s+1} are coupled and are considered as one 96-bit quantity. Of this quantity 94 bits are shifted left or right the specified number of positions to form an intermediate result. The bits corresponding to A^s_0 and A^{s+1}_0 are not shifted. Bit A^{s+1}_0 specifies the value of the result bit A^{r+1}_0, but does not enter the operation in any other way. Except for the treatment of bits A^{s+1}_0 and A^{r+1}_0, the double register signed shift instruction is identical in function to the single register integer shift instruction when the latter is considered to operate on a 95-bit quantity instead of a 48-bit quantity.

Pictorially the integer shift, double register, instruction is:

double shift left

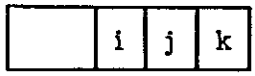


double shift right



The value of s must be even. If it is not, the low order bit specifying s is forced to 0, exception bit RS is set, and the operation proceeds.

Integer Shift, Double SID

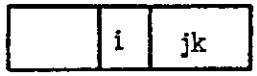


Shift amount $\leftarrow A^{k}_{38,\ldots,47}$

$A^{i,i+1} \leftarrow$ integer shift $(A^{j,j+1})$

Exceptions	Exception bit
bit different from A^{j}_{0} shifted out during left shift	SO
i or j odd	RS

Integer Shift, Double by Constant SIDC



Shift amount $\leftarrow$ jk

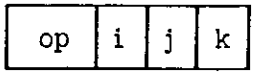
$A^{i,i+1} \leftarrow$ integer shift $(A^{i,i+1})$

Exceptions	Exception bit
bit different from A^{i}_{0} shifted out during left shift	SO
i odd	RS

LOGICAL OPERATIONS

A comprehensive set of logical operations is included on the arithmetic registers, the index registers, and condition bits. For most of the logical instructions the two operands are treated as either 1-, 24-, or 48-bit quantities and a logical connective is applied bit by bit. However, for the "count" instructions a function is computed, not on corresponding pairs of bits of different operands, but on all 24 or 48 bits of one operand.

All logical operations have the short format:



where the j- and k-fields designate the operand registers or bits and the i-field designates the result register or bit. The contents of the operand registers or bits are not changed by the execution of a logical operation.

The basic set of logical operations provides for eight logical connectives, applied bit by bit on the operands. The truth tables for these eight functions are:

function	function value a 0 0 1 1 b 0 1 0 1	common names of function	base mnemonic
$a \wedge b$	0 0 0 1	and, logical product	AND
$a \wedge \bar{b}$	0 0 1 0	logical difference	TAF
$\bar{a} \wedge \bar{b}$	1 0 0 0	nor, Peirce stroke	FAF
$a \vee b$	0 1 1 1	or, logical sum	OR
$a \vee \bar{b}$	1 0 1 1	cover	TOF
$\bar{a} \vee \bar{b}$	1 1 1 0	nand, Scheffer stroke	FOF
$a = b$	1 0 0 1	equivalence	EQ
$a \neq b$	0 1 1 0	not equal, exclusive or, modulo 2 sum	XOR

It should be noted that all sixteen possible Boolean functions of two variables can be computed by these eight operations by interchanging the names in the j- and k-fields or by setting k equal to j. In particular are the following common functions (where R may be interpreted as either A, X, or c):

move	$R^i \leftarrow R^j$	$R^i \leftarrow R^j \wedge R^j$
complement and move	$R^i \leftarrow \bar{R}^j$	$R^i \leftarrow \bar{R}^j \wedge \bar{R}^j$
set to 0's	$R^i \leftarrow 0$'s	$R^i \leftarrow R^i \wedge \bar{R}^i$
set to 1's	$R^i \leftarrow 1$'s	$R^i \leftarrow R^i \vee \bar{R}^i$

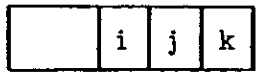
In addition to the operations included in this section, the shift instructions and certain move instructions provide logical (i. e. bit by bit) functions.

Logical Operations, Arithmetic Unit

ANDA	$A^i \leftarrow A^j \wedge A^k$
TAFA	$A^i \leftarrow A^j \wedge \bar{A}^k$
FAFA	$A^i \leftarrow \bar{A}^j \wedge \bar{A}^k$
ORA	$A^i \leftarrow A^j \vee A^k$
TOFA	$A^i \leftarrow A^j \vee \bar{A}^k$
FOFA	$A^i \leftarrow \bar{A}^j \vee \bar{A}^k$
EQA	$A^i \leftarrow A^j = A^k$
XORA	$A^i \leftarrow A^j \neq A^k$

Exceptions: none

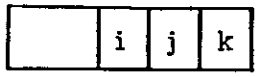
Logical Operations, Index Unit



ANDX	$X^i \leftarrow X^j \wedge X^k$
TAFX	$X^i \leftarrow X^j \wedge \bar{X}^k$
FAFX	$X^i \leftarrow \bar{X}^j \wedge \bar{X}^k$
ORX	$X^i \leftarrow X^j \vee X^k$
TOFX	$X^i \leftarrow X^j \vee \bar{X}^k$
FOFX	$X^i \leftarrow \bar{X}^j \vee \bar{X}^k$
EQX	$X^i \leftarrow X^j = X^k$
XORX	$X^i \leftarrow X^j \neq X^k$

Exceptions: none

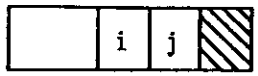
Logical Operations, Condition Bits



ANDC	$c_i \leftarrow c_j \wedge c_k$
TAFC	$c_i \leftarrow c_j \wedge \overline{c}_k$
FAFC	$c_i \leftarrow \overline{c}_j \wedge \overline{c}_k$
ORC	$c_i \leftarrow c_j \vee c_k$
TOFC	$c_i \leftarrow c_j \vee \overline{c}_k$
FOFC	$c_i \leftarrow \overline{c}_j \vee \overline{c}_k$
EQC	$c_i \leftarrow c_j = c_k$
XORC	$c_i \leftarrow c_j \neq c_k$

Exception	Exception bit
c_{24} set to 0 or c_{25} set to 1	CC

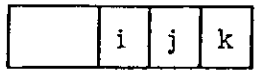
Count Total Ones, Arithmetic CNTT



The contents of register A^i are replaced by the number of bits of register A^j which have the value 1.

Exceptions: none

Count Leading Alike, Arithmetic CNTAA



The contents of register A^i are replaced by the number of leading bits of register A^j which have the value of the bit A^k_0. The bits of A^j are examined in the order A^j_0, A^j_1, A^j_2, and so on.

Note that if the k-field specifies A^0, the effect is to count leading 0's.

Exceptions: none

Count Leading Different, Arithmetic CNTDA

The contents of register A^i are replaced by the number of leading bits of register A^j which are different in value from the value of bit A^k_0 (that is, have the value $\bar{A}^k_0$). The bits of A^j are examined in the order A^j_0, A^j_1, A^j_2, and so on.

Note that if the k-field specifies A^0, the effect is to count leading 1's.

Exceptions: none

Count Leading Alike, Index CNTAX

The contents of register X^i are replaced by the number of leading bits of register X^j which have the value of the bit X^k_0. The bits X^j are examined in the order X^j_0, X^j_1, X^j_2, and so on.

Note that if the k-field specifies X^0, the effect is to count leading 0's.

Exceptions: none

Count Leading Different, Index CNTDX

The contents of register X^i are replaced by the number of leading bits of register X^j which are different in value from the value of bit X^k_0 (that is, have the value $\bar{X}^k_0$). The bits of X^j are examined in the order X^j_0, X^j_1, X^j_2, and so on.

Note that if the k-field specifies X^0, the effect is to count leading 1's.

Exceptions: none

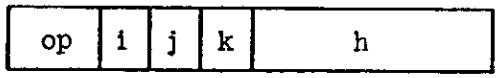
BRANCH AT EXIT OPERATIONS

Branch-at-Exit instructions form the basic set which permits alteration of sequential execution of instructions.

To specify a change in the sequence (i. e. , a branch) three decisions are required: (1) whether or not the branch is to be taken, that is, the condition determination; (2) when the branch is to be taken, the exit point specification; and (3) the address to which the branch is to be made, the effective address calculation.

Condition Determination

The conditional Branch-at-Exit instructions have the long format:



The i- and j-fields designate the bits of the condition register used to determine whether or not the branch is taken. The k-field designates an X-register which with the literal h-field is used to compute the effective branch address.

Whether or not the branch is to be taken is computed as a function of two bits selected from the condition register c (special register S^0). The i- and j-fields select the bits of c; the function which is computed is specified by the operation code. If the value of the function is TRUE (1), the branch is called successful and the alteration of sequence is effected at the next EXIT instruction. If the value of the function is FALSE (0), the branch is called unsuccessful and no alteration of sequence occurs.

Eight functions can be specified:

$c_i \wedge c_j$ $\quad$ $c_i \vee c_j$

$c_i \wedge \overline{c}_j$ $\quad$ $c_i \vee \overline{c}_j$

$\overline{c}_i \wedge \overline{c}_j$ $\quad$ $\overline{c}_i \vee \overline{c}_j$

$c_i = c_j$ $\quad$ $c_i \neq c_j$

A branch controlled by a single bit may be specified by setting j equal to i. An unconditional branch may be specified by the true function $c_i = c_i$ for any i.

If any of the (non-existent) condition bits 24 through 31 is addressed, the bit value 0 is used.

There is a single unconditional Branch-at-Exit instruction which has the short format:

The i- and j-fields of this instruction are ignored, and the condition value TRUE is used so that this branch is always successful.

Exit Point

The sequential nature of instruction execution is not altered by the Branch-at-Exit instruction itself. Rather, the branch point is marked by an EXIT instruction, and, when a branch is successful, the actual alteration of instruction flow occurs at the EXIT. Instructions between the branch instruction and the EXIT are executed normally, independent of whether the branch is successful or unsuccessful.

When two or more branch instructions occur without an intervening EXIT, the branch instructions are examined in order. The first branch which is successful governs the next EXIT; the other branch instructions which follow the successful branch but preceed the EXIT are ignored. The set of branch instructions which relate to a single EXIT need not be in adjacent storage locations but may be interspersed with other instructions (except EXITs).

If an EXIT occurs without a successful branch having been executed since the last previous EXIT, the instruction flow continues in a sequential manner.

Effective Branch Address

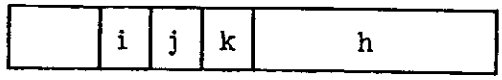
The effective branch address, eba, designates the location of the instruction to which the instruction execution sequence will be altered if the branch is successful. The point of alteration is determined by an EXIT instruction.

The eba may be specified in either of two ways: in the 24-bit unconditional branch instruction eba is given directly by the contents of index register k; in 48-bit instructions eba is the modulo 2^{24} sum of index register k and the 24-bit literal field of the instruction.

instruction format	eba calculation
short	eba ← X^k
long	eba ← $X^k + h$

If the branch is successful and if the eba designates a missing address, at the next EXIT exception bit MI is set to 1 and the program is interrupted (see the section on Sequencing for further details). If the branch is unsuccessful, no exception can occur.

Branch at Exit, Conditional



mnemonic	function
BAND	$c_i \wedge c_j$
BTAF	$c_i \wedge \bar{c}_j$
BFAF	$\bar{c}_i \wedge \bar{c}_j$
BOR	$c_i \vee c_j$
BTOF	$c_i \vee \bar{c}_j$
BFOF	$\bar{c}_i \vee \bar{c}_j$
BEQ	$c_i = c_j$
BXOR	$c_i \neq c_j$

Exceptions: none

Branch at Exit, Unconditional

mnemonic	function
BU	identically TRUE

Exceptions: none

Exit Operations

An EXIT instruction serves to mark a branch point, where one sequential pattern of instruction execution terminates and another sequential pattern begins.

Two exit operations are provided. The EXIT instruction serves only to designate a branch point. The EXITL instruction does three functions in the following logical order: it sets the skip state to "not skipping", it performs the function of the MLX instruction, and it designates a branch point.

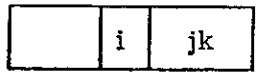
A branch point designation cannot be skipped. Thus, if an EXIT instruction is flagged as skippable, the flag is ignored. If an EXITL is flagged, its first two functions may be skipped but the branch point designation may not.

Exit EXIT

The branching action for any previous branch instruction occurs at the point designated by this instruction.

Exceptions: none

Exit, Save Location and Stop Skipping EXITL



This instruction is logically identical to the three instructions:

```
SKTAF  31,31
MLX    i,jk
EXIT
```

Exceptions: none

Skip Operations

Skip operations provide the ability to inhibit the execution of a set of instructions following the skip instruction. The skipping action is conditional on a function of two bits of the condition register; the instructions to be skipped are indicated by a special bit in the operation code. Thus, to specify a skip two parameters are required: (1) whether or not the skip is to be made, the condition determination; and (2) which instructions are to be skipped, the skip scope.

Condition Determination

Whether or not the skip is to be taken is computed as a function of two bits selected from the condition register c (special register S^0). The i- and j-fields select the bits of c; the function which is computed is specified by the operation code. If the value of the function is TRUE (1), the skip is called successful and the flagged instructions within the scope of the skip will be ignored. If the value of the function is FALSE (0), the skip is called unsuccessful and instructions within the scope of the skip are executed normally.

Eight functions can be specified:

$c_i \wedge c_j$	$c_i \vee c_j$
$c_i \wedge \bar{c}_j$	$c_i \vee \bar{c}_j$
$\bar{c}_i \wedge \bar{c}_j$	$\bar{c}_i \vee \bar{c}_j$
$c_i = c_j$	$c_i \neq c_j$

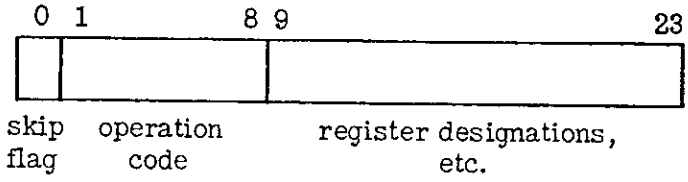
A skip controlled by a single bit may be specified by setting j equal to i.

If any of the (non-existent) condition bits 24 through 31 are addressed, the bit value 0 is used.

It will be noted that skip condition is determined exactly the same as the branch-at-exit condition.

Scope of the Skip

The scope of a skip instruction is those instructions between the SKIP and the next SKIP instruction which is not skipped. Those instructions within the scope which may be skipped are designated by setting a special bit in the instruction to 1. One bit position in the operation code of all instructions is designated as the skip flag; it is bit number 0 in the format which is common to all instructions:



The mnemonic means of designating an instruction with its skip flag set to is to proceed the instruction's menmonic by an asterisk (*).

If the skip condition is TRUE, all instructions within the scope with this skip flag set to 1 are ignored.

If the skip condition is FALSE, all instructions within the scope are executed normally (independent of the value of their skip flag).

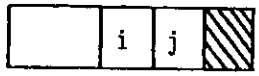
The instructions within the scope of a SKIP which are designated as skippable by having their skip flags set to 1 need not be in adjacent storage locations. They may be interspersed with other unflagged (and hence unconditionally executed) instructions.

All instructions except an EXIT instruction may be flagged as skippable. In particular a skip or branch instruction may be skipped.

The skip state (i. e., "skipping": ignore flagged instructions, or "not skipping": execute all instructions) is altered only as shown in the following table:

Instruction	New Skip State
SKIP	determined by condition determination
EXITL	not skipping
SVC, IC	not skipping
SVR, IC	determined by bit S_6^{11}
SCAN	determined by scan-in data

Skip



menmonic	function
SKAND	$c_i \wedge c_j$
SKTAF	$c_i \wedge \bar{c}_j$
SKFAF	$\bar{c}_i \wedge \bar{c}_j$
SKOR	$c_i \vee c_j$
SKTOF	$c_i \vee \bar{c}_j$
SKFOF	$\bar{c}_i \vee \bar{c}_j$
SKEQ	$c_i = c_j$
SKXOR	$c_i \neq c_j$

Exceptions: none

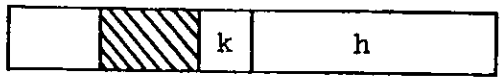
Special Purpose Branch Instructions

The special purpose branch instructions are included primarily for use in interruption servicing routines and for changing the status of the supervisory-problem mode. These situations require special treatment because the concurrency of operation in the MPM creates circumstances not normally encountered in a non-overlapped computer. To treat these situations without these instructions would be both awkward and excessively time consuming.

Many instructions in this class are essentially unconditional branch instructions. The formation of the effective branch address is different for each instruction. However the point at which the branch is to occur is marked by an EXIT, as usual.

Invalidate Instruction Buffers and Branch

IVIB



At the next EXIT the contents of all instruction buffers are invalidated. Any instructions which had been prefetched into the instruction buffers and any instructions in the dispatch registers or contender registers following the EXIT are fetched from storage again.

For the branching action, IVIB appears as a successful branch instruction. That is, unless there is an outstanding successful branch instruction, a branch occurs at the next EXIT to the location designated by the effective branch address, eba. The eba is calculated as

$$eba \leftarrow X^{k} + h.$$

If a successful branch is outstanding, all IVIB functions are suppressed.

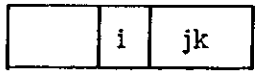
Exceptions: none

Pause PAUSE

The execution of all instructions preceding PAUSE are completed. Any interruptions occasioned by these instructions are also taken. After all these actions are accomplished, the execution of the next instruction in sequence is begun.

Exceptions: none

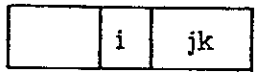
Pause with Exception PI



Index register X^i is replaced by the value specified by the 10-bit literal jk-field. Before the replacement the 10-bit quantity is extended to 24 bits by appending 14 high order bits equal in value to the high order bit of the jk-field. Also the PI exception bit is set to 1. Then a PAUSE is executed, so that an interruption is taken before the execution of the next instruction in sequence is begun.

Exception	Exception bit
always set	PI

Supervisor Call SVC



If there are no outstanding successful branch instructions, SVC is performed as follows:

1. Index register X^i is replaced by the value specified by the 10-bit literal jk-field. Before the replacement the 10-bit quantity is extended to 24 bits by appending 14 high order bits equal in value to the high order bit of the jk-field.

At the next EXIT the following are also performed:

2. The current values of the MPM mode bits $S^{11}_{0,1,2}$ replace the values of the bits $S^{11}_{13,14,15}$.

3. The MPM is placed in the following mode:

   a. supervisory
   b. concurrent

(Note that the disable/enable mode is not altered.)

4. The internal branch-skip-MPC state is saved in bits 3 through 9 of S^{11}. (Note that the recorded branch state is always "no outstanding branches".)

5. The internal branch-skip-MPC state is set as follows:

   a. no outstanding branches
   b. not skipping
   c. no carry

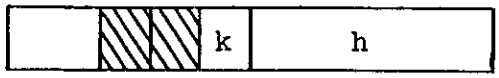
6. A branch is taken to fixed location 256 with respect to the supervisory normal key.

The setting of the mode bits is interlocked with the execution of other instructions to give the effect of sequential execution, so that concurrency problems associated with the entrance to the supervisory mode are avoided.

If a successful branch is outstanding, all SVC functions are suppressed.

Exceptions: none

Supervisor Return SVR



If there are no outstanding successful branch instructions, SVR is performed as follows at the next EXIT:

1. The MPM mode bits $S^{11}_{0,1,2}$ are set to the values $S^{11}_{13,14,15}$.

2. The internal branch-skip-MPC state is set to the values designated by bits 3 through 9 of the machine state register S^{11}. This setting of the branch state neither effects nor is effected by the branching action of step 3.

3. A branch is taken to the address designated by the eba where

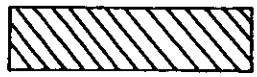
$$eba \leftarrow X^k + h.$$

The branch is with respect to the normal key of the mode specified by S^{11}_{13}.

The setting of the mode bits is interlocked with the execution of other instructions to give the effect of sequential execution, so that concurrency problems associated with the return are avoided.

If a successful branch is outstanding, all SVR functions are suppressed.

Exception	Exception bit
in problem mode	PV

Interrupt Call IC 

The interrupt call instruction is internally generated and inserted into the instruction stream to effect an interruption. IC is not available for use as a programmed instruction. IC is performed as follows:

1. The current values of the MPM mode bits $S^{11}_{0,1,2}$ replace the values of bits $S^{11}_{10,11,12}$.

2. The MPM is placed in the following mode:

   a. supervisory
   b. disabled
   c. concurrent

3. The interruption return address register S^{9} is set to the address to which a return should be made in order to resume the interrupted program in its proper logical sequence.

4. The internal effective branch address is saved in register S^{10}.

5. The internal branch-skip-MPC state is saved in bits 3 through 9 of register S^{11}.

6. The internal branch-skip-MPC state is set as follows:

   a. no outstanding branches
   b. not skipping
   c. no carry

7. A branch is taken to fixed location 0, with respect to the supervisory normal key.

IC is interlocked so that it is executed in strict sequence with each stream; that is, it cannot pass any instructions ahead of it (instructions in the program being interrupted), nor can it be passed by any instructions behind it (instructions in the program at location 0).

Interrupt Return IR 

If there are no outstanding successful branch instructions, IR is performed as follows at the next EXIT:

1. The MPM mode bits $S^{11}_{0,1,2}$ are set to the values of $S^{11}_{10,11,12}$.
2. The internal branch-skip-MPC state is set to the values designated by bits 3 through 9 of the machine state register S^{11}.
3. The internal effective branch address is set to the value of S^{10}.
4. A branch is taken to the address designated by the interruption return address register S^{9}. The branch is with respect to the normal key of the mode specified by S^{11}_{10}. This branching action neither effects nor is effected by the actions of steps 2 and 3.

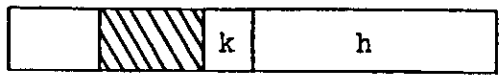
The setting of the mode bits and the branch-skip-MPC state are interlocked with the execution of other instructions to give the effect of sequential execution, so that concurrency problems associated with the return are avoided.

If a successful branch is outstanding, all IR functions are suppressed.

Exception	Exception bit
in problem mode	PV

Scan In

SCAN



The MPM registers and control triggers are reset to state specified by the contents of storage starting at the effective address ea, where

$$eal \leftarrow X^k + h$$

$$eak \leftarrow \text{alternate key}$$

The nine low order bits of ea are ignored and assumed to be 0's. Thus the scan data is assumed to be aligned on a 256-word boundary.

The storage arrangement of the registers and triggers is specified in the section "MPM Interruptions".

After completing SCAN, execution is resumed according to the state specified by the scanned-in data.

Exceptions	Exception bit
in problem mode	PV
missing address	MA

INPUT/OUTPUT OPERATIONS

The input/output (I/O) operations provide for the initiation, termination, and testing of data movement between storage and input/output devices. The actual data movement is controlled by channels and device control units. The T registers contain interruption and mask bits and other control and status data for channels as shown below.

The descriptions of the instructions SIO, SIOA, HIO, TC and RC given below are incomplete and specify only the basic function of the instruction. Complete descriptions of these instructions and the use of the T registers are included in the section "Input/Output Module".

T REGISTERS

Number	Name	Length in Bits	
0	Interruption Signal, Channels 0 to 47	48	Note 1
1	Interruption Signal, Channels 48 to 95	48	
2	Mask, Channels 0 to 47	48	Note 2
3	Mask, Channels 48 to 95	48	
4	Enable Search	1	Note 3
5	Channel Number	8	Note 4
6	Interruption Status I	48	
7	Interruption Status II	48	
8	Test Channel Status I	48	
9	Test Channel Status II	48	
10	Busy, Channels 0 to 47	48	Note 5
11	Busy, Channels 48 to 95	48	

Notes:

1. For the instruction MOT, the register pair $T^{0,1}$ is considered as a 96-bit register with bits numbered 0 to 95. Neither MXT nor MZT will modify $T^{0,1}$.

2. For the instructions MZT and MOT, the register pair $T^{2,3}$ is considered as a 96-bit register with bits numbered 0 to 95.

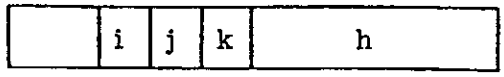
3. The unused bits of T^4 are bits 1 through 47.

4. The unused bits of T^5 are bits 0 through 39.

5. Register pair $T^{10,11}$ may be set only by channels. Hence the instructions MXT, MZT, and MOT have no effect.

Start I/O

SIO



$$\text{device number} \leftarrow X^{j}_{8,9,\ldots,15}$$

$$\text{channel number} \leftarrow X^{j}_{16,17,\ldots,23}$$

$$\text{eal} \leftarrow X^{k} + h$$

$$\text{eak} \leftarrow \text{normal key}$$

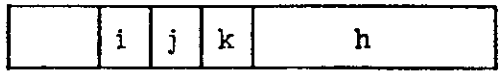
If the specified channel is not operational or is busy, bit c_i is set to 1. Otherwise, bit c_i is set to 0, and the channel command parameters (CCP) at storage location ea are sent to the channel. The CCP specifies the command to be executed by the channel and the device. The format of the CCP in storage is

command code (8), flags (4), key (12)	in location ea
address (24)	in location ea + 1
ignored (8), skip count (16)	in location ea + 2
ignored (8), transmission count (16)	in location ea + 3
FHF parameters (24)	in location ea + 4

The conditions for initiation, execution, and termination of both SIO and the command specified by the CCP are specified in the section "Input/Output Module".

Exceptions	Exception bit
in problem mode	PV
c_{24} set to 0 or c_{25} set to 1	CC

Start I/O per Alternate Key SIOA



$$\text{device number} \leftarrow X^{j}_{8,9,\ldots,15}$$

$$\text{channel number} \leftarrow X^{j}_{16,17,\ldots,23}$$

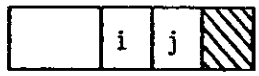
$$\text{eal} \leftarrow X^{k} + h$$

$$\text{eak} \leftarrow \text{alternate key}$$

This instruction is identical to SIO except that in forming the storage address the alternate key is used.

Exceptions	Exception bit
in problem mode	PV
c_{24} set to 0 or c_{25} set to 1	CC

Halt I/O HIO



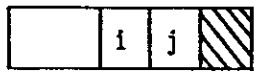
$$\text{device number} \leftarrow X^{j}_{8,9,\ldots,15}$$

$$\text{channel number} \leftarrow X^{j}_{16,17,\ldots,23}$$

If the specified channel is not operational, bit c_i is set to 1. Otherwise, bit c_i is set to 0, and the execution of the current operation at the specified channel and device is terminated. The conditions for initiation, execution, and termination of HIO are specified in the section "Input/Output Module".

Exceptions	Exception bit
in problem mode	PV
c_{24} set to 0 or c_{25} set to 1	CC

Test Channel TC

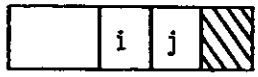


channel number $\leftarrow X^{j}_{16,17,\ldots,23}$

If the specified channel is not operational, bit c_i is set to 1. Otherwise, bit c_i is set to 0, and the contents of the channel status data (CSD) register of the specified channel replaces the contents of registers $T^{7,8}$. The operation of the channel is not effected. The conditions for initiation, execution, and termination of TC are specified in the section "Input/Output Module".

Exceptions	Exception bit
in problem mode	PV
c_{24} set to 0 or c_{25} set to 1	CC

Reset Channel RC



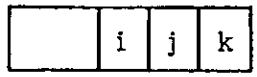
channel number $\leftarrow X^{j}_{16,17,\ldots,23}$

If the specified channel is not operational, bit c_i is set to 1. Otherwise, bit c_i is set to 0, and the specified channel and all devices attached to it are reset. The conditions for initiation, execution, and termination of RC are specified in the section "Input/Output Module".

Exceptions	Exception bit
in problem mode	PV
c_{24} set to 0 or c_{25} set to 1	CC

Move T Register to Index

MTX



$X^{i,j} \leftarrow T^{k}$

Exception

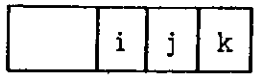
in problem mode

Exception bit

PV

Move Index to T Register

MXT



$T^{i} \leftarrow X^{j,k}$

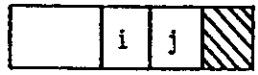
Exception

in problem mode

Exception bit

PV

Move Zero to T-Register Bit MZT



$n \leftarrow X^j$

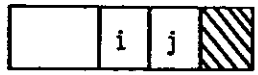
$T^i_n \leftarrow 0$

If n exceeds the length of T^i, no bit is set.

For this instruction, the register pairs $T^{0,1}$ (IO interrupt register) and $T^{2,3}$ (IO mask register) are each considered as a 96-bit register with bits numbered 0 through 95. The pair $T^{0,1}$ may be addressed by setting the i-field to either 0 or 1; similarly $T^{2,3}$ addressed by either 2 or 3.

Exception	Exception bit
in problem mode	PV

Move One to T-Register Bit MOT



$n \leftarrow X^j$

$T^i_n \leftarrow 1$

If n exceeds the length of T^i, no bit is set.

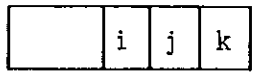
For this instruction, the register pairs $T^{0,1}$ (IO interrupt register) and $T^{2,3}$ (IO mask register) are each considered as a 96-bit register with bits numbered 0 through 95. The pair $T^{0,1}$ may be addressed by setting the i-field to either 0 or 1; similarly $T^{2,3}$ addressed by either 2 or 3.

Exception	Exception bit
in problem mode	PV

TAG AND DIRECTORY OPERATIONS

The tag and directory operations provide for the manipulation of the storage control portion of the Bus and Lining Module.

All tag and directory instructions are in the short format:



The i-, j-, and k-fields always refer to X-registers. Whenever a pair of X registers is specified, the value of the i-, j-, or k-field (as appropriate) is assumed to be even. If it is not, the low order bit of the field is forced to 0, exception bit RS is set, and the operation proceeds. The 48-bit quantity $X^{0,1}$ is defined as 48 0's.

Tag and Directory instructions (except ITUMA) may be executed only when the MPM is in the supervisory mode; if one is encountered in the problem mode, exception bit PV is set and the instruction execution is suppressed so that no X-registers or tag or directory entires are changed.

A complete description of these instructions is included in the section "Bus and Lining Module".

Invalidate Tag and Update MS per Alternate Key

ITUMA

eal ← X^j + X^k

eak ← alternate key

If the line containing the ea is present in HSS, its copy in MS is set equal to the HSS copy, and the tag corresponding to the line is made invalid. Otherwise no change takes place.

Exceptions: none

Invalidate Tag and Update MS

ITUM

The MS copy of each line in HSS is made equal to the HSS value. All tags are made invalid.

Exception	Exception bit
problem mode	PV

Directory Enter DEN

The contents of register pair $X^{k,k+1}$ specify a directory entry. A directory search is performed (using increasing counts appropriate to the page size) to locate an invalid entry. Then the contents of X^k and X^{k+1} replace that invalid entry.

The physical directory address (PDA) of the invalid entry and the count used to locate it are returned to register X^i in bit positions 0,1,...,11 and 12,13,...,17 respectively; bits 18,19,..., 23 are set to 0's.

If no invalid entry can be located, no directory entry is made; a count of 32 and a PDA of 0 are returned to X^i.

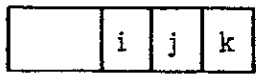
Exception	Exception bit
in problem mode	PV
k odd	RS

Directory Enter per Physical DENP

Bit 0,1,...,11 of X^k specify a PDA. The contents of register pair $X^{i,i+1}$ replace the directory entry at location PDA. No check is made that this is a legitimate PDA for this directory entry.

Exception	Exception bit
in problem mode	PV
i odd	RS

Directory Swap DSW



Bits 19,20,...,46 of register pair $X^{k,k+1}$ specify a virtual page address. A directory search is performed to locate the entry corresponding to this virtual address. The entry is returned to the register pair $X^{i,i+1}$. Then the contents of $X^{j,j+1}$ replace the contents of the entry just located. No check is made that this location is a legitimate PDA for the directory entry specified by $X^{j,j+1}$.

If the entry cannot be located, $X^{i,i+1}$ are set to 0's, and no new directory entry is made.

Exception	Exception bit
in problem mode	PV
i,j, or k odd	RS

Directory Move and Invalidate DM

Bits 0,1,...,11 of register X^k specify one PDA (pda_1); bits 12,13,...,23 specify a second PDA (pda_2).

The directory entry at location pda_2 replaces the directory entry at location pda_1, and the entry at pda_2 is replaced by the invalid pattern (forty-eight 0's are stored).

The move and invalidation are interlocked so that no intervening accesses to location pda_1 are permitted.

No check is made that the directory entry in pda_2 can be legitimately located in pda_1.

Exception	Exception bit
in problem mode	PV

Directory Examine DEX

Bits 19,20,...,46 of register pair $X^{k,k+1}$ specify a virtual address. The directory entry corresponding to this virtual page address replaces the contents of registers $X^{i,i+1}$.

If no entry can be located, $X^{i,i+1}$ are set to 0's.

Exception	Exception bit
in problem mode	PV
i or k odd	RS

Directory Examine per Physical DEXP

Bits 0,1,...,11 of X^k specify a PDA. The directory entry at location PDA replaces the contents of register pair $X^{i,i+1}$.

Exception	Exception bit
in problem mode	PV
i odd	RS

Directory Search for Smaller DSS

Bits 19,20,...,46,47 of register pair $X^{k,k+1}$ specify a virtual page address and page size. A directory search is performed to find either an invalid entry or an entry specifying a page size smaller than the page size of the search argument. Upon locating either type of entry, the PDA and the ID-PS field of the entry is returned to register pair $X^{i,i+1}$ in bit positions 0,1,...,11 and 18,19,...,47 respectively.

If an invalid entry was found, bits $X^{k}_{12,13}$ are set to 0,0. If a smaller page entry was found, bits $X^{i}_{12,13}$ are set to 0,1. If the search was unable to locate either type entry, bits $X^{i}_{12,13}$ are set to 1,0. In all cases bits $X^{i}_{14,15,16,17}$ are set to 0's.

Exception	Exception bit
in problem mode	PV
i or k odd	RS

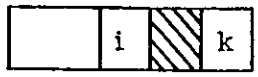
Directory Search for Invalid DSI

Bits 19,20,...,46 of register pair $X^{k,k+1}$ specify a virtual page address. A directory search is performed to find an invalid entry. The PDA of the invalid entry and the count used to locate it are returned to register X^{i} in bit positions 0,1,...,11 and 12,13,...,17 respectively; bits 18, 19,...,23 are set to 0's.

If no invalid entry can be located, the count returned is 32 and the PDA is 0.

Exception	Exception bit
in problem mode	PV
k odd	RS

Directory Search per Count DSC



Bits 19,20,...,46 of the register pair $X^{k,k+1}$ specify a virtual page address. Also bits $X^{k}_{13,14,...,17}$ specify a count. The hash function H (va, cnt) specifies a PDA. This PDA and the ID-PS field of the directory entry at location PDA are returned to register pair $X^{i,i+1}$ in bit positions 0,1,...,11 and 18,19,...,47 respectively; bits 12,13,...,17 are set to 0's.

Exception	Exception bit
in problem mode	PV
i or k odd	RS

SPECIAL REGISTERS

The set of status bits, control bits, and auxiliary registers required for the proper functioning of the MPM constitute the set of special registers. The designation of these registers is shown in Table 1.

Special registers S^{20} through S^{31} are sources of 0's; information loaded into them is not recoverable. Although all special registers are nominally 24 bits in length, not all special registers have 24 physical positions; the unused bits are noted in Table 1. If any register or bit which does not exist in the physical embodiment is addressed as a source operand, the value 0 is supplied; thus, if it is addressed as a result operand, the information is lost.

The special registers S^{3} through S^{31} are accessible only when the processor is in the supervisory mode. If these registers are addressed when in the problem mode, a privileged exception occurs, exception bit PV is set to 1, and the execution of the offending instruction is suppressed in such a way that the contents of all registers remain unchanged.

Each bit position of special registers PX, PM, SX, SM and MS has individual significance to delineate an exceptional condition, a mask, a mode, or machine status. The significance of these bits and their mnemonics are shown in Tables 2, 3, 4, and 5.

The contents of special registers IRA, EBA, and parts of MS have significance only when an interruption occurs. A complete discussion of these registers is given in the chapter "MPM Interruptions".

Special registers GP^{0}, GP^{1}, GP^{2}, and GP^{3} are not reserved for a particular function, but rather may be used as general purpose registers when the processor is in the supervisory mode. Unlike the remainder of the special registers, they are never altered or used except when explicitly addressed.

Special registers are used as operands in the instructions shown in Table 6 (instructions which may cause an exception bit to be set in PX or SX are not included).

Special Registers

Number	Name	Mnemonic	Length in bits	Unused bit positions	Privileged
0	Condition	C	24	-	no
1	Problem Exception	PX	21	21 through 23	no
2	Problem Mask	PM	21	21 through 23	no
3	Supervisory Exception	SX	11	11 through 23	yes
4	Supervisory Mask	SM	11	11 through 23	yes
5	Problem Normal Key	PNK	12	0 through 11	yes
6	Problem Alternate Key	PAK	12	0 through 11	yes
7	Supervisory Normal Key	SNK	12	0 through 11	yes
8	Supervisory Alternate Key	SAK	12	0 through 11	yes
9	Interruption Return Address	IRA	24	-	yes
10	Effective Branch Address	EBA	24	-	yes
11	Machine State	MS	16	16 through 23	yes
12	Cycle Count	CYC	24	-	yes
13	Instruction Count	INC	24	-	yes
14	Timer	TIME	24	-	yes
15	External Signal	ES	24	-	yes
16	General Purpose	GP^0	24	-	yes
17	General Purpose	GP^1	24	-	yes
18	General Purpose	GP^2	24	-	yes
19	General Purpose	GP^3	24	-	yes

TABLE 1

Problem Exception (S^1) and Mask (S^2) Registers

Number	Name	Mnemonic
0	Index Divide by Zero	XDZ
1	Add Overflow	AO
2	Add Underflow	AU
3	Multiply Overflow	MO
4	Multiply Underflow	MU
5	Divide Overflow	DO
6	Divide Underflow	DU
7	Shift Overflow	SO
8	Unnormalized Operand	UO
9	Unnormalized Divisor	UD
10	Illegitimate Operand	ILO
11	Zero Fraction	ZF
12	Low Significance	LS
13	Overflow Warning	OW
14	Underflow Warning	UW
15	Condition Check	CC
16	Address Boundary Violation	BV
17	Illegitimate Instruction Code	IIC
18	Privileged Instruction	PV
19	Register Specification	RS
20	Pause and Interrupt	PI

TABLE 2

Supervisory Exception (S^3) and Mask (S^4) Registers

Number	Name	Mnemonic
0	Input/Output	IO
1	External In	EI
2	Cycle Count Zero	CZ
3	Timer Zero	TZ
4	Instruction Count Zero	IZ
5	Missing Address, Data Load or Store	MA
6	Protected Address	PA
7	Missing Address, Instruction Execution	MI
8	Directory Interrupt	DI
9	Directory Interrupt Overrun	DIO
10	Machine Malfunction	MM

TABLE 3

Machine State Register (S^{11})

Number	Name
0	Supervisory/Problem Mode
1	Disable/Enable Mode
2	Concurrent/Sequential Mode
3	Branch State
4	Branch State
5	Branch State
6	Skip State
7	Multi-precision Carry
8	Multi-precision Carry
9	Multi-precision Carry
10	Previous Supervisory/Problem Mode for Interrupt
11	Previous Disable/Enable Mode for Interrupt
12	Previous Concurrent/Sequential Mode for Interrupt
13	Previous Supervisory/Problem Mode for SVC
14	Previous Disable/Enable Mode for SVC
15	Previous Concurrent/Sequential Mode for SVC

TABLE 4

Explanation of Branch, Skip and MPC States

Branch State $S^{11}_{3,4,5}$

000	No Outstanding Branch
001	Successful Branch Outstanding
010	IVIB Outstanding
011	SVC Outstanding
100	SVR Outstanding
101	IR Outstanding
110	Unused
111	Unused

Skip State S^{11}_{6}

0	Not Skipping
1	Skipping

Multi-precision Carry State $S^{11}_{7,8,9}$

000	No Carry	
001	+ 1	
010	--	exceptional
011	--	exceptional
100	--	exceptional
101	- 3	
110	- 2	
111	- 1	

TABLE 5

Special Registers Used as Operands

Instruction	S Registers as Source	S Registers as Result
MXS, MXSO	-	any
MSX	any	-
MSXZ	any	any
ACH, ACL, SCH, SCL	MS	MS
MCX	C	-
MAC, MXC	-	C
All Logic on Condition Bits	C	C
AXT, AXCT, AXKT	-	C
SIO, SIOA, HIO, TC, RC	-	C
All Compare	-	C
All Branch-at-Exit (except BU)	C	-
All Skip	C	-
SVC	MS	MS
SVR	MS	MS
IC	MS	EBA, IRA, MS
IR	EBA, IRA, MS	MS
SCAN	-	all

TABLE 6