CYBER 70
MODEL 72
MODEL 73
MODEL 74

Computer Systems

REFERENCE MANUAL

Instruction Descriptions

VOLUME 2

CONTROL DATA
CORPORATION

INSTRUCTION INDEX

INSTRUCTION INDEX

CENTRAL PROCESSOR

PERIPHERAL PROCESSORS

CYBER 70

MODEL 72
MODEL 73
MODEL 74

Computer Systems

REFERENCE MANUAL

Instruction Descriptions

VOLUME 2

CONTROL DATA
CORPORATION

REVISION RECORD

REVISION	DESCRIPTION
A	Manual Released.
(2-22-71)	
B	Manual revised, editorial corrections.
(3-1-71)	
Publication No. 60347300	

Address comments concerning this manual to:

Control Data Corporation
Technical Publications Department
4201 North Lexington Avenue
Arden Hills, Minnesota 55112

or use Comment Sheet in the back of this manual.

PREFACE

The CONTROL DATA® CYBER 70 series reference manuals are published in a series of volumes. This manual is volume 2 of the series.

The detailed system description is in volume 1 of the series. Publication number 60347000 covers CYBER 72 systems, 60347200 covers CYBER 73 systems, and 60347400 covers CYBER 74 systems.

Information about the ECS (Extended Core Storage) is in volume 3 of the series, publication number 60347100.

The publications listed are available through the nearest Control Data Corporation sales office.

CONTENTS

TABLES

CENTRAL PROCESSOR INSTRUCTIONS 3.

INSTRUCTION FORMATS

This section describes the Central Processor instructions. The CPU instructions tend to fall into two categories: those causing computation and those causing storage references or program branching. The instructions causing only computation are generally executed in a fixed amount of time after they have been issued. Instructions involving storage references for operands or program branching require variable amounts of time and cannot be precisely timed.

Careful coding of critical program loops can produce substantial improvements in execution time. Detailed timing information is provided in the applicable CYBER 70 series System Description Reference Manual.

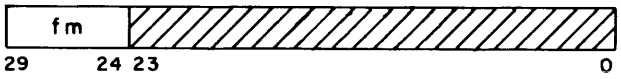
Preceding the description of each instruction is the octal code, the instruction name, the number of bits in the instruction, and a diagram showing the instruction format. Slanted parallel lines within a format diagram indicate unused bit positions. Table 3-1 defines the Central Processor instruction designators.

TABLE 3-1. CENTRAL PROCESSOR INSTRUCTION DESIGNATORS

Designator	Use
A	Specifies one of eight 18-bit address registers.
B	Specifies one of eight 18-bit index registers; B0 is fixed and equal to zero.
C1	The offset (character address) of the first character in the first word of the source field.
C2	The character address of the first character in the first word of the result field.
fm	A 6-bit instruction code.
i	A 3-bit code specifying one of eight designated registers (e.g., Ai).
j	A 3-bit code specifying one of eight designated registers (e.g., Bj).
jk	A 6-bit constant, indicating the number of shifts to be taken.
k	A 3-bit code specifying one of eight designated registers (e.g., Bk).
K	An 18-bit constant, used as an operand or as a branch destination (address).
K1	An 18-bit address indicating the memory location of the first (left-most) character of the source field.
K2	An 18-bit address indicating the memory location of the first (left-most) character of the result field.
LL	The lower 4-bits of a character count, used as part of the 13-bit character count for the number of characters to be moved.
LU	The upper 9-bits of the character count, used with LL
X	Specifies one of eight 60-bit operand registers.

MONITOR, STOP

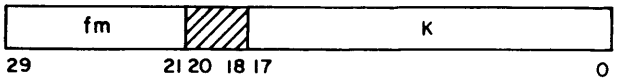
00 *Error Exit to MA or Program Stop* *(15 Bits)*



A panel switch determines which of the functions this instruction performs. In the stop position, the Central Processor is stopped. In the other position, an Error Exit occurs and causes an exchange jump to the monitor address (MA) in the exchange package.

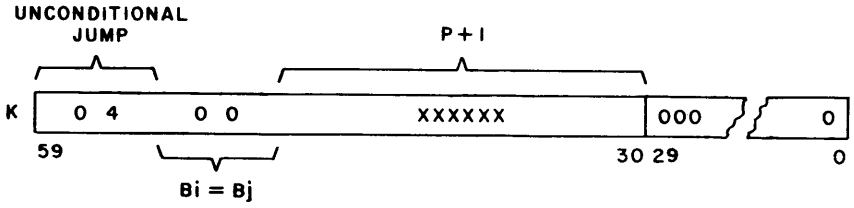
BRANCH

010 *Return Jump to K* *(30 Bits)*



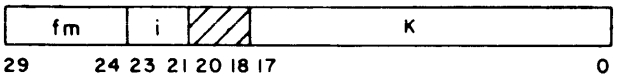
The instruction stores an 04 unconditional jump and the current address plus one [(P) + 1] in the upper half of address K, then branches to K + 1 for the next instruction. Note that this instruction is always out of the instruction stack, thus voiding the stack.

The octal word at K after the instruction appears as follows:



A jump to address K at the end of the branch routine returns the program to the original sequence.

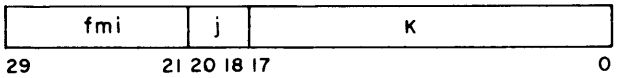
02 *Jump to (Bi) + K* *(30 Bits)*



This instruction adds the contents of increment register Bi to K and branches to the address specified by the sum. The branch address is K when i = 0. Addition is performed modulo $2^{18}-1$.

Note that this instruction is always out of the instruction stack, thus voiding the stack. For an unindexed, unconditional jump, the 04 instruction with i = j = 0 is a better choice. Thus, if this instruction is contained in a tight loop, the instruction at K can be obtained from the stack, if possible.

030	*Jump to K if (Xj) = 0*	*(30 Bits)*
031	*Jump to K if (Xj) ≠ 0*	*(30 Bits)*
032	*Jump to K if (Xj) = plus (positive)*	*(30 Bits)*
033	*Jump to K if (Xj) = negative*	*(30 Bits)*
034	*Jump to K if (Xj) is in range*	*(30 Bits)*
035	*Jump to K if (Xj) is out of range*	*(30 Bits)*
036	*Jump to K if (Xj) is definite*	*(30 Bits)*
037	*Jump to K if (Xj) is indefinite*	*(30 Bits)*



These instructions branch to K when the 60-bit word in operand register X_j meets the condition specified by the i digit. The instruction allows zero, sign, and indefinite forms tests for fixed or floating point words.

The following applies to tests made in this instruction group:

a) The 030 and 031 operations test the full 60-bit word in Xj. The words 000...000 and 777...777 are treated as zero. All other words are non-zero.

b) The 032 and 033 operations examine only the sign bit (2^{59}) of Xj. If the sign bit is zero, the word is positive; if the sign bit is one, thw word is negative. Thus, the sign test is valid for fixed point words or for coefficient in floating point words.

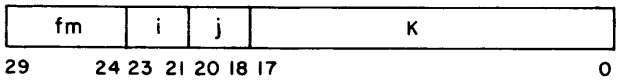
c) The 034 and 035 operations examine the upper-order 12 bits of Xj. Both plus and minus infinity are detected:

3777XX...XX and 4000XX...XX are out of range; all other words are in range.

d) The 036 and 037 operations examine the upper-order 12 bits of Xj. Both plus and minus indefinite forms are detected:

1777XX...XX and 6000XX...XX are indefinite; all other words are definite.

04	*Jump to K if (Bi) = (Bj)*	*(30 Bits)*
05	*Jump to K if (Bi) ≠ (Bj)*	*(30 Bits)*
06	*Jump to K if (Bi) ≥ (Bj)*	*(30 Bits)*
07	*Jump to K if (Bi) < (Bj)*	*(30 Bits)*



These instructions test an 18-bit word from register Bi against an 18-bit word from register Bj (both words signed quantities) for the condition specified and branch to address K on a successful test. All tests against zero (all zeros) can be made by setting Bj = B0.

The following rules apply in the tests made by these instructions:

a) Positive zero is recognized as unequal to negative zero, and

b) Positive zero is recognized as greater than negative zero, and

c) A positive number is recognized as greater than a negative number.

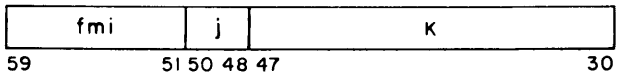
Note that the 06 and 07 instructions first perform a sign test on Bi and Bj and the Branch/No Branch determination is based on the above rules. If Bi and Bj are of the same sign, a subtract test is performed (in the Increment Unit) and the sign of the result (Bi-Bj) determines whether a Branch is made.

EXTENDED CORE STORAGE COMMUNICATION

This category of instructions provides the ability to communicate with Extended Core Storage (ECS). This section describes Extended Core Storage instructions. A more detailed description of the instructions can be found in the Extended Core Storage volume of the Reference Manual (volume 3, Pub. No. 60347100).

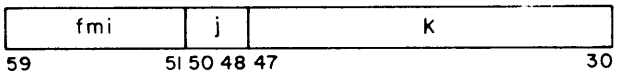
These instructions must be located in the upper order position of the instruction word. If they are not, any attempt at execution will cause an exit to RA_{CM} regardless of the error mode bits. This will also happen if the instructions are used in a system that does not have ECS.

011 *Read Extended Core Storage* *(30 Bits)*



This instruction initiates a Read operation to transfer [(Bj) + K] 60-bit words from Extended Core Storage to Central Memory. The initial Extended Core Storage address is [(X0) + RA_{ECS}]; the initial Central Memory address is [(A0) + RA_{CM}].

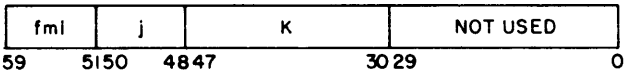
012 *Write Extended Core Storage* *(30 Bits)*



This instruction initiates a Write operation to transfer [(Bj) + K] 60-bit words from Central Memory to Extended Core Storage. The initial Central Memory address is [(A0) + RA_{CM}]; the initial Extended Core Storage address is [(X0) + RA_{ECS}].

CENTRAL EXCHANGE JUMP

013 *Central Exchange Jump* *(60 Bits)*



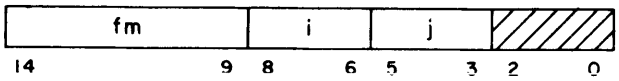
This instruction unconditionally exchange jumps the Central Processor, regardless of the state of the Monitor Flag bit. Instruction action differs, however, depending on whether the Monitor Flag bit is set or clear. Operation is as follows:

a) Monitor Flag bit clear. The starting address for the exchange is taken from the 18-bit Monitor Address register. Note that this starting address is an absolute address. During the exchange, the Monitor Flag bit is set.

b) Monitor Flag bit set. The starting address for the exchange is the 18-bit result formed by adding K to the contents of register Bj. Note that this starting address is an absolute address. During the exchange, the Monitor Flag bit is cleared.

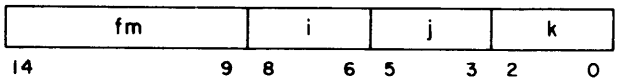
LOGICAL

10 *Transmit (Xj) to Xi* *(15 Bits)*



This instruction transfers a 60-bit word from operand register Xj to operand register Xi.

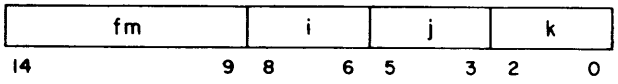
11 *Logical product of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the logical product (AND function) of 60-bit words from operand registers Xj and Xk and places the product in operand register Xi. Bits of register Xi are set to "1" when the corresponding bits of the Xj and Xk registers are "1" as in the following example:

(Xj) = 0101
(Xk) = 1100
Xi = 0100

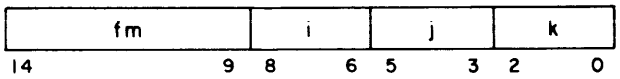
12 *Logical sum of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the logical sum (inclusive OR) of 60-bit words from operand registers Xj and Xk and places the sum in operand register Xi. Bits of register Xi are set to "1" if the corresponding bit of the Xj or Xk register is a "1" as in the following example:

(Xj) = 0101
(Xk) = 1100
Xi = 1101

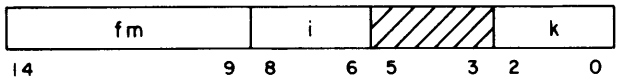
13 *Logical difference of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the logical difference (exclusive OR) of 60-bit words from operand registers Xj and Xk and places the difference in operand register Xi. Bits of register Xi are set to "1" if the corresponding bits in the Xj and Xk registers are unlike as in the following example:

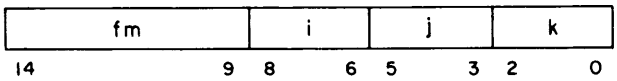
$$\begin{aligned} (Xj) &= 0101 \\ (Xk) &= 1100 \\ \hline Xi &= 1001 \end{aligned}$$

14 *Transmit the complement of (Xk) to Xi* *(15 Bits)*



This instruction extracts the 60-bit word from operand register Xk, complements it, and transmits this complemented quantity to operand register Xi.

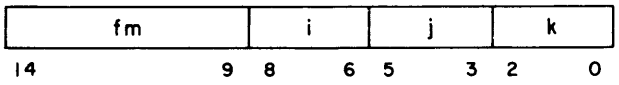
15 *Logical product of (Xj) and complement of (Xk) to Xi* *(15 Bits)*



This instruction forms the logical product (AND function) of the 60-bit quantity from operand register Xj and the complement of the 60-bit quantity from operand register Xk, and places the result in operand register Xi. Thus, bits of Xi are set to "1" when the corresponding bits of the Xj register and the complement of the Xk register are "1" as in the following example:

$$\begin{aligned} (Xj) &= 0101 \\ \text{Complemented } (Xk) &= 0011 \\ \hline Xj &= 0001 \end{aligned}$$

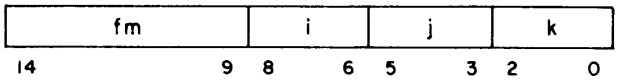
16 *Logical sum of (Xj) and complement of (Xk) to Xi* *(15 Bits)*



This instruction forms the logical sum (inclusive OR) of the 60-bit quantity from operand register Xj and the complement of the 60-bit word from operand register Xk, and places the result in operand register Xi. Thus, bits of Xi are set to "1" if the corresponding bit of the Xj register or complement of the Xk register is a "1" as in the following example:

```
(Xj)                = 0101
Complemented (Xk)   - 0011
Xi                  = 0111
```

17 *Logical difference of (Xj) and complement of (Xk) to Xi* *(15 Bits)*

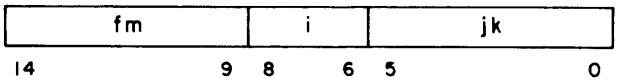


This instruction forms the logical difference (exclusive OR) of the quantity from operand register Xj and the complement of the 60-bit word from operand register Xk, and places the result in operand register Xi. Thus, bits of Xi are set to "1" if the corresponding bits of register Xj and the complement of register Xk are unlike as in the following example:

```
(Xj)                = 0101
Complemented (Xk)   = 0011
Xi                  = 0110
```

SHIFT

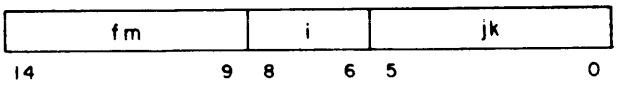
20 *Left shift (Xi), jk places* *(15 Bits)*



This instruction shifts the 60-bit word in operand register Xi left circular jk places. Bits shifted off the left end of operand register Xi replace those from the right end.

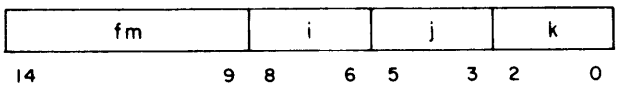
The 6-bit shift count jk allows a complete circular shift of register Xi.

21 *Arithmetic right shift (Xi), jk places* *(15 Bits)*



This instruction shifts the 60-bit word in operand register Xi right jk places. The rightmost bits of Xi are discarded and the sign bit is extended.

22 *Left shift (Xk) nominally (Bj) places to Xi* *(15 Bits)*

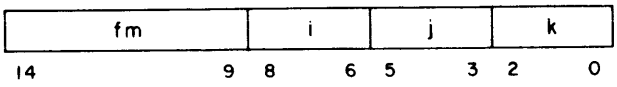


This instruction shifts the 60-bit quantity from operand register Xk the number of places specified by the quantity in increment register Bj and places the result in operand register Xi.

1) If Bj is positive (i.e., bit 17 of Bj = 0), the quantity from Xk is shifted left-circular. (The low order six bits of Bj specify the shift count.)

2) If Bj is negative (i.e., bit 17 of Bj = 1), the quantity from Xk is shifted right (end off with sign extention). (The one's complement of the low order eleven bits of Bj specify the shift count.) If any of bits 2^6-2^{10}, after complementing, are "1's", the shift is not performed and the result register Xi is cleared to all zeros.

23 *Right shift (Xk) nominally (Bj) places to Xi* *(15 Bits)*

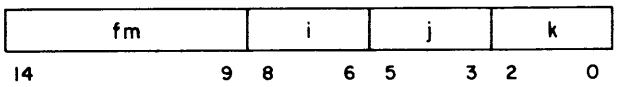


This instruction shifts the 60-bit quantity from operand register Xk the number of places specified by the quantity in increment register Bj and places the result in operand register Xi.

1) If Bj is positive (i.e., bit 17 of Bj = 0), the quantity from register Xk is shifted right (end-off with sign extension). (The low order eleven bits of Bj specify the shift count.) If any of bits 2^6-2^{10} are "1's", the shift is not performed and the result register Xi is cleared to all zeros.

2) If Bj is negative (i.e., bit 17 of Bj = 1), the quantity from register Xk is shifted left circular. (The complement of the lower order six bits of Bj specify the shift count.)

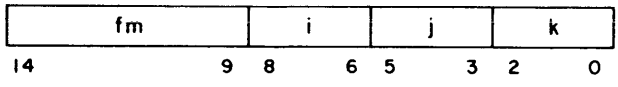
24 *Normalize (Xk) to Xi and Bj* *(15 Bits)*



This instruction normalizes the floating point quantity from operand register Xk and places it in operand register Xi. The number of left shifts necessary to normalize the quantity is entered in increment register Bj. A Normalize operation may cause underflow which will clear Xi to all zeros regardless of the original sign of Xk. Normalizing either a plus or minus zero coefficient sets the shift count (Bj) to 48_{10} and clears Xi to all zeros.

If Xk contains an infinite quantity (3777X...X or 4000X...X) or an indefinite quantity (1777X..,X or 6000X...X), no shift takes place. The contents of Xk are copied into Xi and Bj is set equal to zero. Optional error exits do occur.

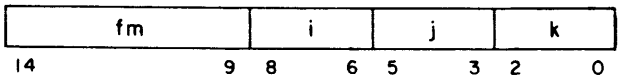
25 *Round and normalize (Xk) to Xi and Bj* *(15 Bits)*



This instruction performs the same operation as instruction 24 except that the quantity from operand register Xk is rounded before it is normalized. Rounding is accomplished by placing a "1" round bit immediately to the right of the least significant coefficient bit. Normalizing a zero coefficient places the round bit in bit 47 and reduces the exponent by 48. Note that the same rules apply for underflow.

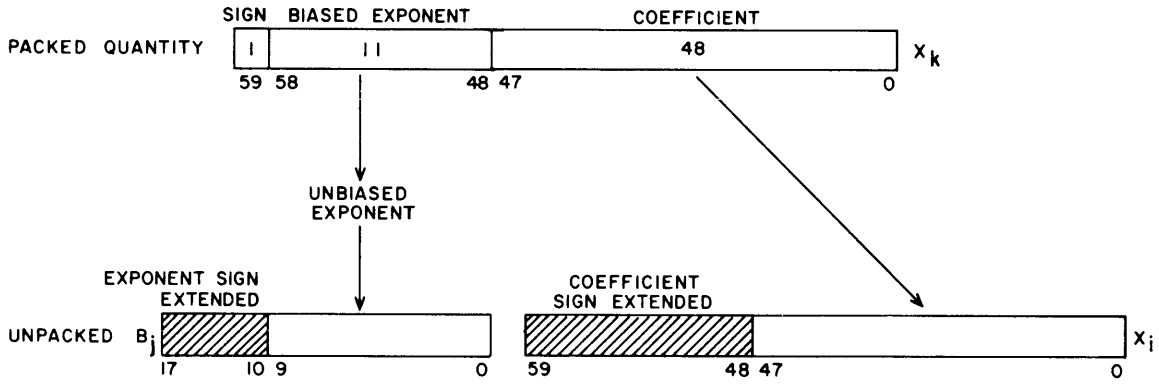
If Xk contains an infinite quantity (3777X...X or 4000X...X) or an indefinite quantity (1777X...X or 6000X...X), no shift takes place. The contents of Xk are copied into Xi and Bj is set equal to zero. Optional error exits do occur.

26 *Unpack (Xk) to Xi and Bj* *(15 Bits)*

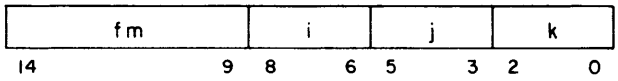


This instruction unpacks the floating point quantity from operand register Xk and sends the 48-bit coefficient to operand register Xi and the 11-bit exponent to increment register Bj. The exponent bias is removed during Unpack so that the quantity in Bj is the true one's complement representation of the exponent.

The exponent and coefficient are sent to the low-order bits of the respective registers as shown below:



27 *Pack Xi from (Xk) and (Bj)* *(15 Bits)*

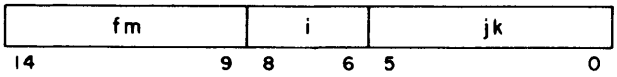


This instruction packs a floating point number in operand register Xi. The coefficient of the number is obtained from operand register Xk and the exponent from increment register Bj. Bias is added to the exponent during the Pack operation. The instruction does not normalize the coefficient.

Exponent and coefficient are obtained from the proper low-order bits of the respective registers and packed as shown in the illustration for the Unpack (26) instruction. Thus, bits 48 to 58 of Xk and bits 11 to 17 of Bj are ignored. There is no test for overflow or underflow.

Note that if Xk is positive, the packed exponent occupying positions 48 to 58 of Xi is obtained from bits 0 to 10 of Bj by complementing bit 10; if Xk is negative, bit 10 is not complemented but bits 0 to 9 are.

43 *Form mask in Xi, jk bits* *(15 Bits)*

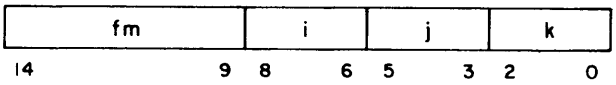


This instruction forms a mask in operand register Xi. The 6-bit quantity jk defines the number of "1's" in the mask as counted from the highest order bit in Xi.

The contents of operand register i = 0 when jk = 0.

FLOATING POINT ARITHMETIC

30 *Floating sum of (Xj) and (Xk) to Xi* *(15 Bits)*

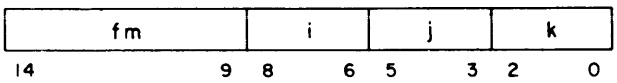


This instruction forms the sum of the floating point quantities from operand registers Xj and Xk and packs the result in operand register Xi. The packed result is the upper half of a double precision sum.

At the start both arguments are unpacked, and the coefficient of the argument with the smaller exponent is entered into the upper half of a 98-bit accumulator. The coefficient is shifted right by the difference of the exponents. The other coefficient is then added into the upper half of the accumulator. If overflow occurs, the sum is right-shifted one place and the exponent of the result increased by one. The upper half of the accumulator holds the coefficient of the sum, which is not necessarily in normalized form. The exponent and upper coefficient are then repacked in operand register Xi.

If both exponents are zero (2000_8) and no overflow occurs, the instruction causes an ordinary integer addition. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

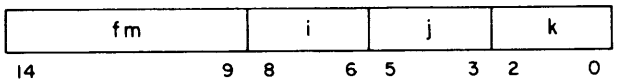
31 *Floating difference (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the difference of the floating point quantities from operand registers Xj and Xk and packs the result in operand register Xi. Alignment and overflow operations are similar to the Floating Sum (30) instruction, and the difference is not necessarily normalized. The packed result is the upper half of a double precision difference.

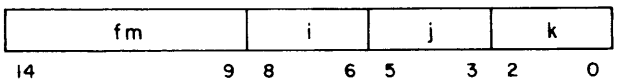
An ordinary integer subtraction is performed when the exponents are zero. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

32 *Floating DP sum of (Xj) and (Xk) to Xi* *(15 Bits)*



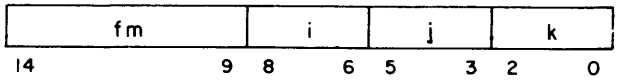
This instruction forms the sum of two floating point numbers as in the Floating Sum (30) instruction, but packs the lower half of the double precision sum with an exponent 48 less than the upper sum. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

33 *Floating DP difference of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the difference of two floating point numbers as in the Floating Difference (31) instruction, but packs the lower half of the double precision difference with an exponent of 48 less than the upper sum. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

34 *Round floating sum of (Xj) and (Xk) to Xi* *(15 Bits)*

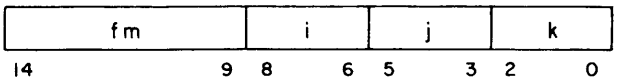


This instruction forms the round sum of the floating point quantities from operand registers Xj and Xk and packs the upper sum of the double precision result in operand register Xi. The sum is formed in the same manner as the Floating Sum instruction but the operands are rounded before the addition, as shown below, to produce a round sum.

1) A round bit is attached at the right end of both operands if:
   a) both operands are normalized, or
   b) the operands have unlike signs.
2) A round bit is attached at the right end of the operand with the larger exponent for all other cases.
3) In the event that the operands have equal exponents, a round bit is attached to the coefficient for only one of the operands.

For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

35 *Round floating difference of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the round difference of the floating point quantities from operand registers Xj and Xk and packs the upper difference of the double precision result in operand register Xi. The difference is formed in the same manner as the Floating Difference (31) instruction but the operands are rounded before the subtraction, as shown below, to produce a round difference.

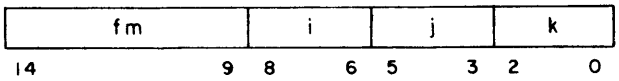
1) A round bit is attached at the right end of both operands if:
   a) both operands are normalized, or
   b) the operands have like signs.

2) A round bit is attached at the right end of the operand with the larger exponent for all other cases.

3) In the event that the operands have equal exponents, a round bit is attached to the coefficient for only one of the operands.

For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

40 *Floating product of (Xj) and (Xk) to Xi* *(15 Bits)*



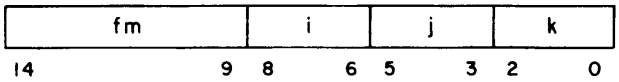
This instruction multiplies two floating point quantities obtained from operand registers Xj (multiplier) and Xk (multiplicand) and packs the upper product result in operand register Xi. When both operands are 48-bit integers with the upper 12-bits sign extended, the integer condition is detected. Integer multiplication takes place and the upper 48-bits of the product are entered into Xi.

The two 48-bit coefficients are multiplied together to form a 96-bit product. The upper 48 bits of the product (bits 48-95) are then packed together with the resulting exponent. Note that when using unnormalized quantities, the entire result could lie in the lower-order 48 bits of the product; hence, this result would be lost when packing occurs.

The result is a normalized quantity only when both operands are normalized; the exponent in this case is the sum of the exponents plus 47 (or 48).

The result is unnormalized when either or both operands are unnormalized; the exponent in this case is the sum of the exponents plus 48. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

41 *Round floating product of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction multiplies the floating point number from operand register Xk (multiplicand), by the floating point number from operand register Xj. The upper product result is packed in operand register Xi. (No lower product available.) The multiply operation is identical to that of instruction 40 with the following exception:

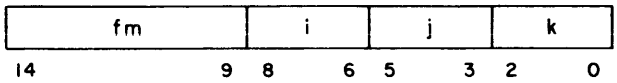
Before the left shift of the final product and during the merge operation to form the final product, a "1" bit is added to bit 2^{46}. The following rounded result is the net effect of this action:

- for products $\geq 2^{95}$, round is by one-fourth
- for all other products, round is by one-half
- when one or both operands are unnormalized, round is by one-fourth.

The result is a normalized quantity only when both operands are normalized; the exponent in this case is the sum of the exponents plus 47 (or 48).

The result is unnormalized when either or both operands are unnormalized; the exponent in this case is the sum of the exponents plus 48. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

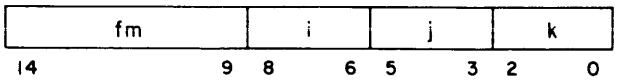
42 *Floating DP product of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction multiplies two floating point quantities obtained from operand registers Xj and Xk and packs the lower product in operand register Xi. The two 48-bit coefficients are multiplied together to form a 96-bit product. The lower-order 48 bits of this product (bits 47-00) are then packed together with the resulting exponent. The result is not necessarily a normalized quantity. The exponent of this result is 48 less than the exponent resulting from a 40 instruction using the same operands. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

This instruction performs an integer multiply if the 12 upper bits (exponents) of both operands are sign extended. The result, the lower 48 bits of the 96-bit product, is entered into Xi with sign extension.

44 *Floating divide (Xj) by (Xk) to Xi* *(15 Bits)*



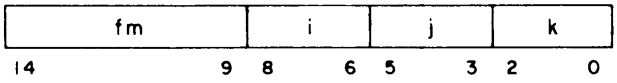
This instruction divides two normalized floating point quantities obtained from operand registers Xj (dividend) and Xk (divisor) and packs the quotient in operand register Xi.

The exponent of the result in a no-overflow case is the difference of the dividend and divisor exponents minus 48.

A one-bit overflow is compensated for by adjusting the exponent and right shifting the quotient one place. In this case the exponent is the difference of the dividend and divisor exponents minus 47.

The result is a normalized quantity when both the dividend and the divisor are normalized. A divide fault occurs when the coefficient of the dividend is two or more times as large as the coefficient of the divisor. This forces an indefinite result (17770...0). To avoid this, normalize both operands before executing this instruction. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

45 *Round floating divide (Xj) by (Xk) to Xi* *(15 Bits)*



This instruction divides the floating quantity from operand register j (dividend) by the floating point quantity from operand register Xk (divisor) and packs the round quotient in operand register Xi. Rounding is accomplished by adding one-third during the division process. In effect, the quantity "$2525....2525_8$" resides immediately to the right of the dividend binary point prior to starting the divide operation. On the first iteration, a "1" is added to the least significant bit of the dividend. After each iteration (subtraction of divisor from partial dividend) a two-place left shift occurs and a "1" is again added to the least significant bit of the partial dividend. Thus, successive iterations gradually bring in the one-third round "quantity" ($25....25_8$).

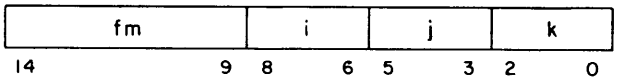
The result exponent in a no-overflow case is the difference of the dividend and divisor exponents minus 48.

A one-bit overflow is compensated for by adjusting the exponent and right shifting the quotient one place; in this case the exponent is the difference of the dividend and divisor exponents minus 47.

The result is a normalized quantity when both the dividend and the divisor are normalized. A divide fault occurs when the coefficient of the dividend is two or more times as large as the coefficient of the divisor. This forces an indefinite result (17770...0). To avoid this, normalize both operands before executing this instruction. For treatment of special operands and/or indefinite forms, refer to the programming information in volume 1.

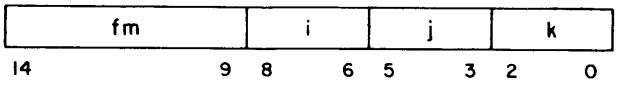
FIXED POINT ARITHMETIC

36 *Integer sum of (Xj) and (Xk) to Xi* *(15 Bits)*



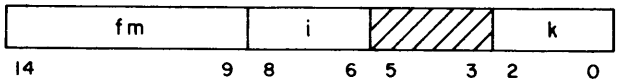
This instruction forms a 60-bit one's complement sum of the quantities from operand registers Xj and Xk and stores the result in operand register Xi. An overflow condition is ignored.

37 *Integer difference of (Xj) and (Xk) to Xi* *(15 Bits)*



This instruction forms the 60-bit one's complement difference of the quantities from operand registers Xj (minuend) and Xk (subtrahend) and stores the result in operand register Xi. An overflow condition is ignored.

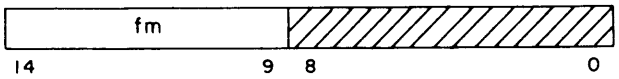
47 *Count the number of "1's" in (Xk) to Xi* *(15 Bits)*



This instruction counts the number of "1's" in operand register Xk and stores the count in the lower order 6 bits of operand register Xi. Bits 6 through 59 are cleared to zero.

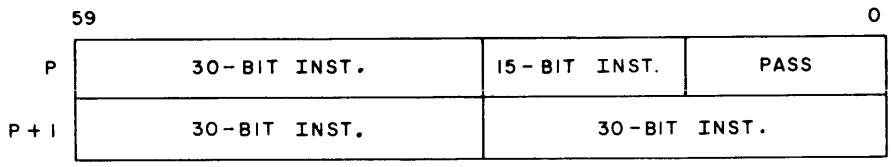
PASS

46 *No operation (Pass)* *(15 Bits)*



This instruction is a "do-nothing" instruction that is typically used to pad the program between certain program steps.

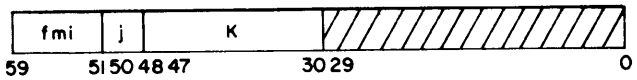
EXAMPLE:



In this example, a Pass instruction is used to pad the remainder of the word at P. Since the next instruction is 30 bits, it cannot fit in P and must be placed in P + 1.

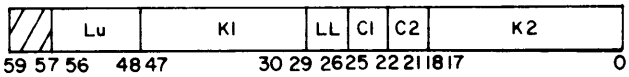
MOVE, COMPARE DATA HANDLING

464 *Move Indirect* *(60 Bits)*



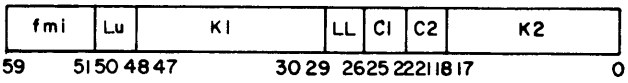
This instruction moves the source field to the result field as specified by the descriptor. The quantity Bj + K is the address of the descriptor. Any instructions located in bit positions 0-29 will not be executed.

60-Bit Descriptor Word



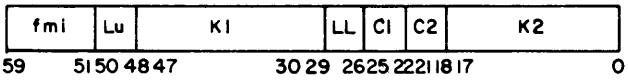
The move is from left to right through the field. Register X0 is cleared at the end of execution.

465 *Move Direct* *(60 Bits)*



This instruction moves the source field to the result field as specified by the descriptor, which must be part of the instruction word. The field length is limited to a 7 bit count. The maximum field length is 177 octal or 127 decimal. If the field length is zero the instruction is a pass, at the end of execution.

466 *Compare Collated* *(60 Bits)*



This instruction compares the field designated by K1, C1 with the field designated by K2, C2 and sets X0 as follows:

If (field K1) is greater than (field K2), set X0 to 00 - 0XXX
If (field K1) is equal to (field K2), set X0 to 00 - 000
If (field K1) is less than (field K2), set X0 to 77 - 7YYY
where YYY is the complement of XXX

Ll and Lu are the 7 bit lengths of the fields (in 6 bit characters) being compared. The maximum length of a field for this instruction is 177 octal or 127 decimal.

If L = 0 the instruction is a pass.

The compare is made left to right through the fields until two unequal characters are found. These two characters are then collated. If the value found for the two unequal characters is the same, the compare continues until another pair of characters are unequal or until the field length is exhausted.

The value of the two octal numbers XX, stored in X0 is determined by the equation L - N = XX, where L is the length of the field and N is the number of pairs of characters that were collated equal, prior to instruction termination. In other words XX is the number of pairs of characters not yet compared plus one.

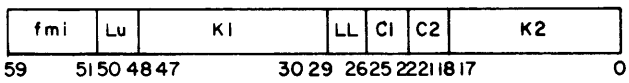
Register A0 contains the starting word address of an 8 word, 64 character, collating table. This table must have been previously stored in consecutive memory locations.

Address	Collating Character Locations							
A0	00	01	02	03	04	05	06	07
A0+1	10	11	12	13	14	15	16	17
A0+2	20	21	22	23	24	25	26	27
A0+3	30	31	32	33	34	35	36	37
A0+4	40	41	42	43	44	45	46	47
A0+5	50	51	52	53	54	55	56	57
A0+6	60	61	62	63	64	65	66	67
A0+7	70	71	72	73	74	75	76	77

59 12 11 0

The value of the character being collated is found by examination of the table. Suppose the character under examination were an octal 63. The 6 would be added to the contents of register A0 to form the word address and the 3 would be used to pick the correct character from that word. Note that this number corresponds to the character address.

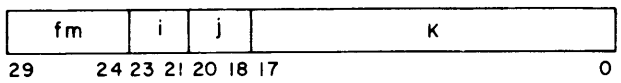
467 *Compare Uncollated* *(60 Bits)*



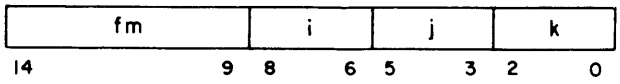
This instruction is identical to the Compare Collated instruction with one exception. The collating table is not used. X0 is set when the first pair of unequal characters is encountered or when the field length is exhausted.

INCREMENT

50 *Set Ai to (Aj) + K* *(30 Bits)*
51 *Set Ai to (Bj) + K* *(30 Bits)*
52 *Set Ai to (Xj) + K* *(30 Bits)*



53	*Set Ai to (Xj) + (Bk)*	*(15 Bits)*
54	*Set Ai to (Aj) + (Bk)*	*(15 Bits)*
55	*Set Ai to (Aj) − (Bk)*	*(15 Bits)*
56	*Set Ai to (Bj) + (Bk)*	*(15 Bits)*
57	*Set Ai to (Bj) − (Bk)*	*(15 Bits)*



These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in address register Ai. Overflow, in itself, is ignored, but an address range fault may result from overflow in this set of instructions.

Operands are obtained from address (A), increment (B), and operand (X) registers as well as the instruction itself (K = 18-bit signed constant). Operands obtained from an Xj operand register are the truncated lower 18 bits of the 60-bit word.

Note that an immediate memory reference is performed to the address specified by the final content of address registers A1 - A7. The operand read from memory address specified by A1 - A5 is sent to the corresponding operand register X1 - X5. When A6 or A7 is referenced, the operand from the corresponding X6 or X7 operand register is stored at the address specified by A6 or A7.

NOTE

If, in this category of instructions, the result placed in address register Ai is an address out of range, the following occurs: (Note that this action is independent of an Exit selection on Address Out of Range.)

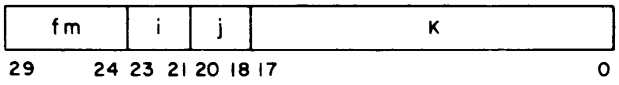
If i = 1-5: Operand register Xi is loaded with the contents of absolute address zero and the contents of memory location (Ai) are unchanged.

If i = 6 or 7: Operand register Xi retains its original contents and the contents of memory location (Ai) are unchanged.

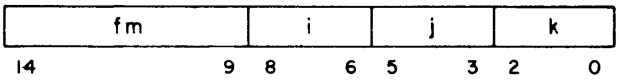
EXAMPLE:

		Initial Quantities
50	$SA_1 = A_j + K \quad i = 4$	$K = 234567_8$
	$SA_4 = A_6 + K \quad j = 6$	$A_4 = 321110_8$
	$SA_4 = 032100_8 + 234567_8$	$A_6 = 032100_8$
	$SA_4 = 266667_8$	X_4 - 00..... 00_8
		Storage location 266667 = 7 ... 75342104600_8
		Final Quantities:
		$A_4 = 266667_8$
		$A_6 = 032100_8$
		$X_4 = 7$ ... 75342104600_8

60	*Set Bi to (Aj) + K*	*(30 Bits)*
61	*Set Bi to (Bj) + K*	*(30 Bits)*
62	*Set Bi to (Xj) + K*	*(30 Bits)*



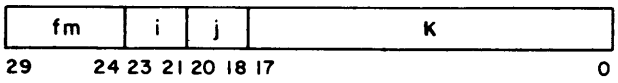
63	*Set Bi to (Xj) + (Bk)*	*(15 Bits)*
64	*Set Bi to (Aj) + (Bk)*	*(15 Bits)*
65	*Set Bi to (Aj) − (Bk)*	*(15 Bits)*
66	*Set Bi to (Bj) + (Bk)*	*(15 Bits)*
67	*Set Bi to (Bj) − (Bk)*	*(15 Bits)*



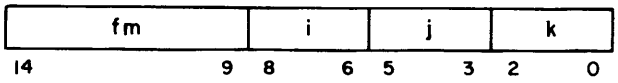
These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in increment register Bi. An overflow condition is ignored.

Operands are obtained from address (A), increment (B), and operand (X) registers as well as the instruction itself (K = 18-bit signed constant). Operands obtained from an Xj operand register are the truncated lower 18 bits of the 60-bit word.

70	*Set Xi to (Aj) + K*	*(30 Bits)*
71	*Set Xi to (Bj) + K*	*(30 Bits)*
72	*Set Xi to (Xj) + K*	*(30 Bits)*



73	*Set Xi to (Xj) + (Bk)*	*(15 Bits)*
74	*Set Xi to (Aj) + (Bk)*	*(15 Bits)*
75	*Set Xi to (Aj) − (Bk)*	*(15 Bits)*
76	*Set Xi to (Bj) + (Bk)*	*(15 Bits)*
77	*Set Xi to (Bj) − (Bk)*	*(15 Bits)*



These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result into the lower 18 bits of operand register Xi. The sign of the result is extended to the upper 42 bits of operand register Xi. An overflow condition is ignored.

Operands are obtained from address (A), increment (B), and operand (X) registers as well as the instruction itself (K = 18-bit signed constant). Operands obtained from an Xj operand register are the truncated lower 18 bits of the 60-bit word.

EXAMPLE:

```
73      SXi   Xj + Bk      i = 2
        SX2   X3 + B1      j = 3, k = 1
        SX2 = 0 ... 0652224310(8) + 511245(8)
        SX2 = 7 ... 7777735555(8)
```

Initial Quantities:

$X_2 = 0 \ldots 0745321402_8$

$X_3 = 0 \ldots 0652224310_8$

$B_1 = 511245_8$

Final Quantities:

$X_2 = 7 \ldots 7777735555_8$

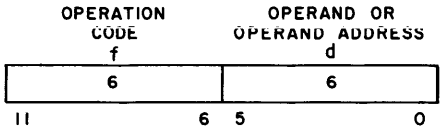
$X_3 = 0 \ldots 0652224310_8$

$B_1 = 511245_8$

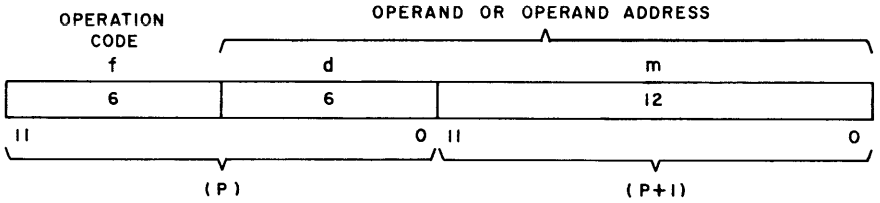
PERIPHERAL PROCESSOR INSTRUCTIONS 4.

INSTRUCTION FORMATS

Two formats are used; 12-bit and 24-bit. The 12-bit format has a 6-bit operation code f and a 6-bit operand or operand address d.



The 24-bit format uses the 12-bit quantity m, the contents of the next program address (P + 1), with d to form an 18-bit operand or operand address.



ADDRESS MODES

Program indexing is accomplished and operands are manipulated in several modes. The two instruction formats provide for 6-bit or 18-bit operands and 6-bit, 12-bit or 18-bit addresses.

NO ADDRESS MODE

In this mode d or dm is used as an operand. This mode eliminates the need for storing constants. The d quantity is considered a 12-bit number, the upper six bits of which are zero. The dm quantity has d as the upper six bits and m as the lower 12 bits.

DIRECT ADDRESS MODE

In this mode, d or m + (d) is used as the address of the operand. The d quantity specifies one of the first 64 addresses in memory ($0000\text{-}0077_8$). The m + (d) quantity generates a 12-bit address for referencing all possible peripheral memory locations ($0000\text{-}7777_8$). If $d \neq 0$, the content of address d is added to m to produce an operand address (indexed addressing). If d = 0, m is taken as the operand address.

EXAMPLE: Address Modes

Given: d = 25
m = 100
contents of location 25 = 0150
contents of location 150 = 7776
contents of location 250 = 1234

Then:

MODE	INSTRUCTION	A REGISTER
No Address	14	000025
	20	250100
Direct Address	30	000150
	50	001234
Indirect Address	40	007776

INDIRECT ADDRESS MODE

In this mode, d specifies an address which contains the address of the desired operand. Thus the operand is indirectly obtained. Indirect addressing and indexed addressing require an additional memory reference beyond that required by direct addressing.

The description of instructions uses the expression (d) to define the contents of memory location d. An expression with double parentheses ((d)) refers to indirect addressing. The expression (m + (d)) refers to direct addressing when d = 0 and to indexed direct addressing when $d \neq 0$. Table 4-1 summarizes the addressing modes used for the Peripheral Processor instructions.

TABLE 4-1. ADDRESSING MODES FOR PERIPHERAL PROCESSOR INSTRUCTIONS

Instruction Type	Address Mode		
	Direct	Indirect	No Address
Load	30, 50	40	14, 20
Add	31, 51	41	16, 21
Subtract	32, 52	42	17
Logical Difference	33, 53	43	11, 23
Store	34, 54	44	
Replace Add	35, 55	45	
Replace Add One	36, 56	46	
Replace Subtract One	37, 57	47	
Long Jump	01		
Return Jump	02		
Unconditional Jump			03
Zero Jump			04
Non-Zero Jump			05
Positive Jump			06
Minus Jump			07
Shift			10
Logical Product			12, 22
Selective Clear			13
Load Complement			15

DESCRIPTION OF INSTRUCTIONS

This section describes the Peripheral Processor instructions. Table 4-2 lists designators used throughout the section.

TABLE 4-2. PERIPHERAL PROCESSOR INSTRUCTION DESIGNATORS

Designator	Use
A	The A register.
d	A 6-bit operand or operand address.
f	A 6-bit instruction code.
m	A 12-bit quantity used with d to form an 18-bit operand or operand address.
P	The Program Address register.
Q	The Q register.
( )	Contents of a register or location
(())	Refers to indirect addressing.

Preceding the description of each instruction is the octal code, the instruction name and instruction length.

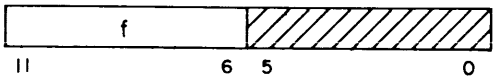
EXAMPLE:

52	Subtract (m + (d))	(24 Bits)
Octal Code	Instruction Name	Instruction Length

Instruction formats are also given; hashed lines within a format indicate bits which are not used in the operation.

NO OPERATION

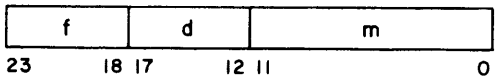
00	*Pass*	*(12 Bits)*
24	*Pass*	*(12 Bits)*
25	*Pass*	*(12 Bits)*



These instructions specify that no operation be performed. They provide the means for padding out a program.

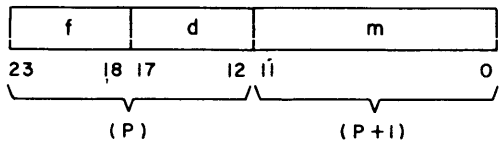
BRANCH

01 *Long Jump to m + (d)* *(24 Bits)*



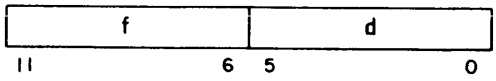
This instruction jumps to the sequence beginning at the address given by m + (d). If d = 0, then m is not modified.

02 *Return Jump to m + (d)* *(24 Bits)*



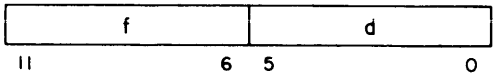
This instruction jumps to the sequence beginning at the address given by m + (d). If d = 0 then m is not modified. The current program address (P) plus two is stored at the jump address. The new program commences at the jump address plus one. This program should end with a long jump to, or normal sequencing into, the jump address minus one, which should in turn contain a long jump, 0100. The latter returns the original program address plus two to the P register.

03 *Unconditional Jump d* *(12 Bits)*



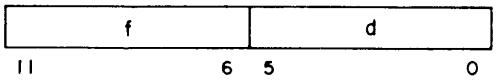
This instruction provides an unconditional jump to any instruction up to 31 steps forward or backward from the current program address. The value of d is added to the current program address. If d is positive (01 - 37), then 0001 (+1) - 0037 (+31) is added and the jump is forward. If d is negative (40 - 76) then 7740 (-31) - 7776 (-1) is added and the jump is backward. The program stops (a Dead Start is necessary to restart the machine) when d = 00 or 77.

04 *Zero Jump d* *(12 Bits)*



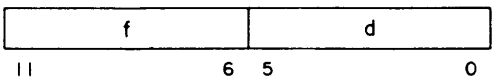
This instruction provides a conditional jump to any instruction up to 31 steps forward or backward from the current program address. If the content of the A register is zero, the jump is taken. If the content of A is non-zero, the next instruction is executed. Negative zero (777777) is treated as non-zero. For interpretation of d see instruction 03.

05 *Non-zero Jump d* *(12 Bits)*



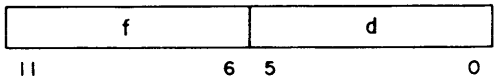
This instruction provides a conditional jump to any instruction up to 31 steps forward or backward from the current program address. If the content of the A register is nonzero, the jump is taken. If A is zero, the next instruction is executed. Negative zero (777777) is treated as nonzero. For interpretation of d see instruction 03.

06 *Plus Jump d* *(12 Bits)*



This instruction provides a conditional jump to any instruction up to 31 steps forward or backward from the current program address. If the content of the A register is positive, the jump is taken. If A is negative, the next instruction is executed. Positive zero is treated as a positive quantity; negative zero is treated as a negative quantity. For interpretation of d see instruction 03.

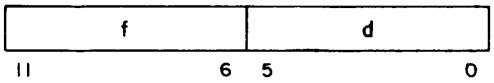
07 *Minus Jump d* *(12 Bits)*



This instruction provides a conditional jump to any instruction up to 31 steps forward or backward from the current program address. If the content of the A register is negative, the jump is taken. If A is positive, the next instruction is executed. Positive zero is treated as a positive quantity; negative zero is treated as a negative quantity. For interpretation of d see instruction 03.

SHIFT

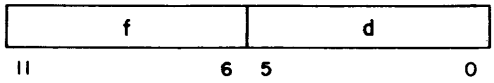
10 *Shift d* *(12 Bits)*



This instruction shifts the contents of A right or left d places. If d is positive (00-37) the shift is left circular; if d is negative (40-77) A is shifted right (end off with no sign extension). Thus, d = 06 requires a left shift of six places. A right shift of six places results when d = 71.

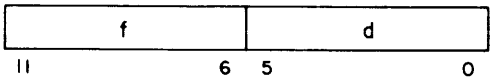
LOGICAL

11 *Logical difference d* *(12 Bits)*



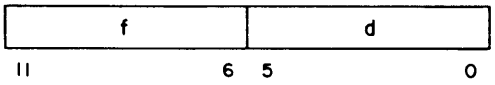
This instruction forms in A the bit-by-bit logical difference of d and the lower six bits of A. This is equivalent to complementing individual bits of A that correspond to bits of d that are one. The upper 12 bits of A are not altered.

12 *Logical product d* *(12 Bits)*



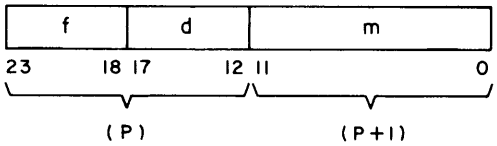
This instruction forms the bit-by-bit logical product of d and the lower six bits of the A register, and leaves this quantity in the lower 6 bits of A. The upper 12 bits of A are zero.

13 *Selective clear d* *(12 Bits)*



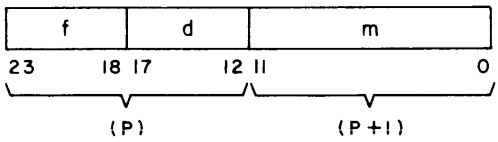
This instruction clears any of the lower six bits of the A register where there are corresponding bits of d that are one. The upper 12 bits of A are not altered.

22 *Logical product dm* *(24 Bits)*



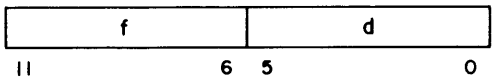
This instruction forms in the A register the bit-by-bit logical product of the contents of A and the 18-bit quantity dm. The upper six bits of this quantity consist of d and the lower 12 bits are the content of the location following the present program address.

23 *Logical difference dm* *(24 Bits)*



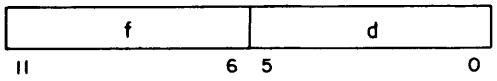
This instruction forms in A the bit-by-bit logical difference of the contents of A and the 18-bit quantity dm. This is equivalent to complementing individual bits of A which correspond to bits of dm that are one. The upper six bits of the quantity consist of d, and the lower 12 bits are the content of the location following the present program address.

33 *Logical difference (d)* *(12 Bits)*



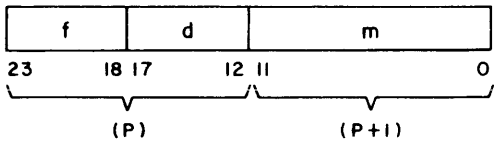
This instruction forms in A the bit-by-bit logical difference of the lower 12 bits of A and the contents of location d. This is equivalent to complementing individual bits of A which correspond to bits of (d) that are one. The upper six bits of A are not altered.

43 *Logical difference ((d))* *(12 Bits)*



This instruction forms in A the bit-by-bit logical difference of the lower 12 bits of A and the 12-bit operand obtained by indirect addressing. Location d is read out of memory, and the word obtained is used as the operand address. The upper six bits of A are not altered.

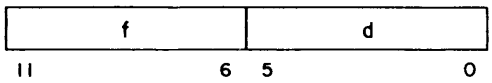
53 *Logical difference (m + (d))* *(24 Bits)*



This instruction forms in A the bit-by-bit logical difference of the lower 12-bits of A and a 12-bit operand obtained by indexed direct addressing. The upper six bits of A are not altered.

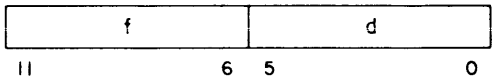
DATA TRANSMISSION

14 *Load d* *(12 Bits)*



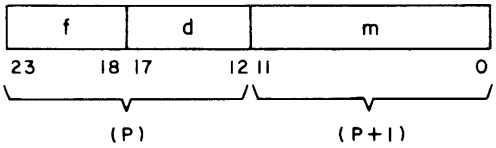
This instruction clears the A register and loads it with d. The upper 12 bits of A are zero.

15 *Load complement d* *(12 Bits)*



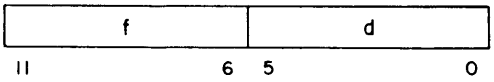
This instruction clears the A register and loads the complement of d. The upper 12 bits of A are set to one.

20 *Load dm* *(24 Bits)*



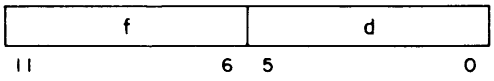
This instruction clears the A register and loads an 18-bit quantity consisting of d as the higher six bits and m as the lower 12 bits. The contents of the location following the present program address are read out to provide m.

30 *Load (d)* *(12 Bits)*



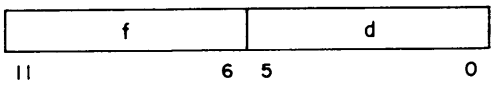
This instruction clears the A register and loads the contents of location d. The upper six bits of A are zero.

34 *Store (d)* *(12 Bits)*



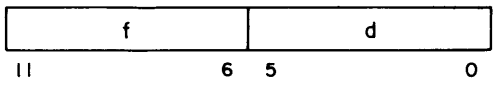
This instruction stores the lower 12 bits of A in location d.

40 *Load ((d))* *(12 Bits)*



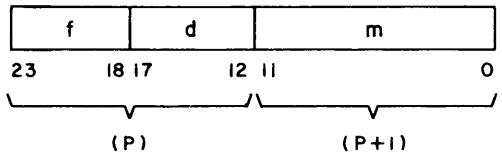
This instruction clears the A register and loads a 12-bit quantity that is obtained by indirect addressing. The upper six bits of A are zero. Location d is read out of memory, and the word obtained is used as the operand address.

44 *Store ((d))* *(12 Bits)*



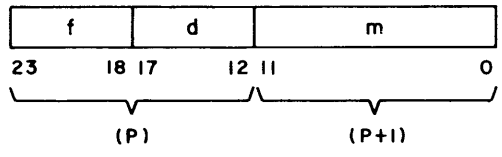
This instruction stores the lower 12 bits of A in the location specified by the contents of location d.

50 *Load (m + (d))* *(24 Bits)*



This instruction clears the A register and loads a 12-bit quantity. The upper six bits of A are zero. The 12-bit operand is obtained by indexed direct addressing. The quantity "m", read out of memory location P + 1 serves as the base operand address to which (d) is added. If d = 0, the operand address is m, but if d ≠ 0, then m + (d) is the operand address. Thus may location d be used for an index quantity to modify operand addresses.

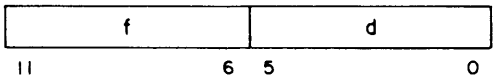
54 *Store (m + (d))* *(24 Bits)*



This instruction stores the lower 12 bits of A in the location determined by indexed addressing (see instruction 50).

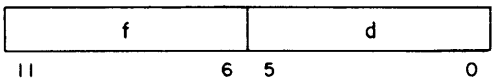
ARITHMETIC

16 *Add d* *(12 Bits)*



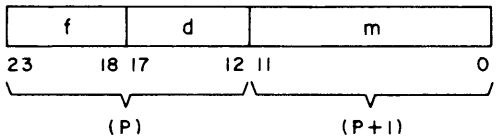
This instruction adds d (treated as a 6-bit positive quantity) to the contents of the A register.

17 *Subtract d* *(12 Bits)*



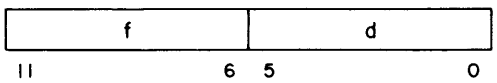
This instruction subtracts d (treated as a 6-bit positive quantity) from the contents of the A register.

21 *Add dm* *(24 Bits)*



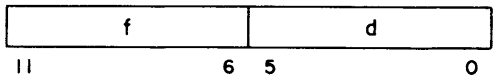
This instruction adds to the A register the 18-bit quantity consisting of d as the higher six bits and m as the lower 12 bits. The contents of the location following the present program address are read out to provide m.

31 *Add (d)* *(12 Bits)*



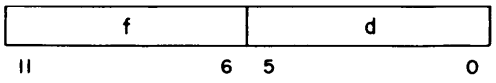
This instruction adds to the A register the contents of location d (treated as a 12-bit positive quantity).

32 *Subtract (d)* *(12 Bits)*



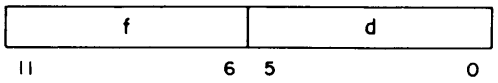
This instruction subtracts from the A register the contents of location d (treated as a 12-bit positive quantity).

41 *Add ((d))* *(12 Bits)*



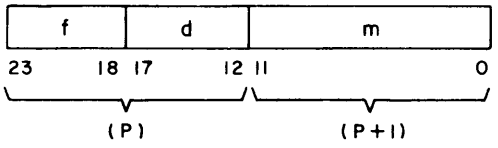
This instruction adds to the contents of A a 12-bit operand (treated as a positive quantity) obtained by indirect addressing. Location d is read out of memory, and the word obtained is used as the operand address.

42 *Subtract ((d))* *(12 Bits)*



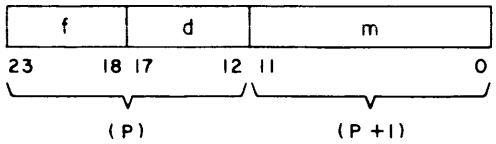
This instruction subtracts from the A register a 12-bit operand (treated as a positive quantity) obtained by indirect addressing. Location d is read out of memory, and the word obtained is used as the operand address.

51 *Add (m + (d))* *(24 Bits)*



This instruction adds to the contents of A a 12-bit operand (treated as a positive quantity) obtained by indexed direct addressing (see instruction 50).

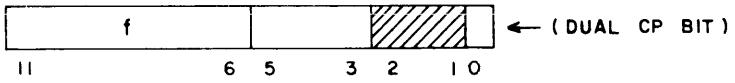
52 *Subtract (m + (d))* *(24 Bits)*



This instruction subtracts from the A register a 12-bit operand (treated as a positive quantity) obtained by indexed direct addressing (see instruction 50).

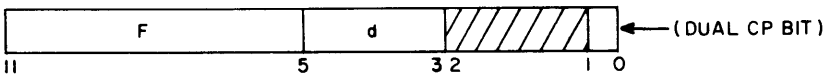
CENTRAL PROCESSOR AND CENTRAL MEMORY COMMUNICATIONS

260 *Exchange Jump* *(12 Bits)*



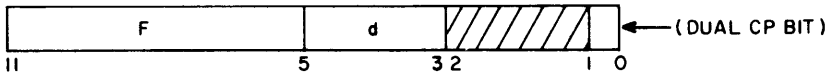
This instruction transmits an 18-bit (absolute) address (only 17 bits are used) from the A register to the Central Processor with a signal which tells the Central Processor to perform an Exchange Jump, with the address in A as the starting location of a file of 16 words containing information about the Central Processor program to be executed. The 18-bit initial address must be entered in A before this instruction is executed. The Central Processor replaces the file with similar information from the interrupted Central Processor program. The Peripheral Processor is not interrupted. In systems with dual Central Processors the lowest order bit specifies which Central Processor the Exchange Jump will interrupt.

261 *Monitor Exchange Jump* *(12 Bits)*



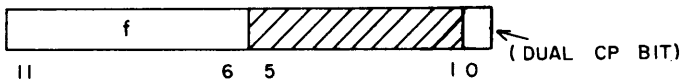
This instruction, typically used to initiate Central Processor Monitor activity, causes a conditional exchange jump to the Central Processor. If the Monitor Flag bit is clear, this instruction sets the flag and initiates the exchange. If the Monitor Flag bit is set, this instruction acts as a Pass instruction. The starting address for this exchange is the 18-bit address held in the Peripheral Processor A register. (The Peripheral Processor program must have loaded A with an appropriate address prior to executing this instruction.) Note that this starting address is an absolute address. This instruction is either 2610 (CPU-0) or 2611 (CPU-1).

262 *Monitor Exchange Jump to MA* *(12 Bits)*



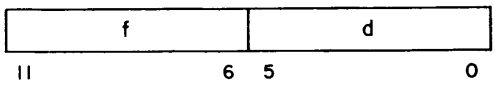
This instruction is a conditional exchange jump of the Central Processor. If the monitor flag bit is clear, this instruction sets the flag and initiates the exchange. If the monitor flag is set, this instruction acts as a Pass instruction. The starting address for this exchange jump is the 18-bit address held in the MA Register. Note that this starting address is an absolute address.

27 *Read program address* *(12 Bits)*



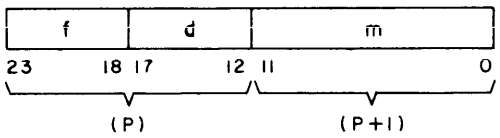
This instruction transfers the content of the Central Processor Program Address register, P, to the Peripheral Processor A register; this allows the Peripheral Processor to determine whether or not the Central Processor is running. In systems with dual central processors, the lowest order bit of the instruction format specifies which central processor P register is to be examined. The largest value that (P) may be is 17 bits. The remaining bit (bit 17) will appear set to this instruction when an ECS transfer is in progress. However, bit 17 is not set in P.

60 *Central read from (A) to d* *(12 Bits)*



This instruction transfers a 60-bit word from Central Memory to five consecutive locations in the processor memory. The 18-bit address of the Central Memory location must be loaded into A prior to executing this instruction. (Note that this is an absolute address.) The 60-bit word is disassembled into five 12-bit words beginning at the left. Location d receives the first 12-bit word. The remaining 12-bit words go to succeeding locations. This instruction will not interrupt an ECS transfer unless bit 17 of the A register is set (Access priority).

61 *Central read (d) words from (A) to m* *(24 Bits)*

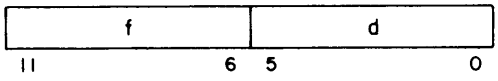


This instruction reads a block of 60-bit words from Central Memory. The content of location d gives the block length. The 18-bit address of the first central word must be loaded into A prior to executing this instruction. (Note that this is an absolute address.) During the execution of the instruction, (P) goes to processor address 0 and P holds m. Also, (d) goes to the Q register where it is reduced by one as each central word is processed. The original content of P is restored at the end of the instruction.

Each central word is disassembled into five 12-bit words beginning with the high-order 12 bits. The first word is stored at processor memory location m. The content of P (which is holding m) is advanced by one to provide the next address in the processor memory as each 12-bit word is stored. If P overflows, operation continues as P is advanced from 7777_8 to 0000_8. These locations will be written into as if they were consecutive.

The content of A is advanced by one to provide the next Central Memory address after each 60-bit word is disassembled and stored. Also, the contents of the Q register are reduced by one. The block transfer is complete when Q = 0. The block of Central Memory locations goes from address (A) to address (A) + (d) -1. The block of processor memory locations goes from address m to m + 5(d) -1. This instruction will not inturrupt an ECS transfer unless bit 17 of the A register is set (Access priority).

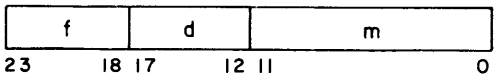
62 *Central write to (A) from d* *(12 Bits)*



This instruction assembles five successive 12-bit words into a 60-bit word and stores the word in Central Memory. The 18-bit address word designating the Central Memory location must be in A prior to execution of the instruction. (Note that this is an absolute address.)

Location d holds the first word to be read out of the processor memory. This word appears as the higher order 12 bits of the 60-bit word to be stored in Central Memory. The remaining words are taken from successive addresses. This instruction will not interrupt an ECS transfer unless bit 17 of the A register is set (Access priority).

63 *Central write (d) words to (A) from m* *(24 Bits)*



This instruction assembles a block of 60-bit words and writes them in Central Memory. The content of location d gives the number of 60-bit words. The content of the A register gives the beginning Central Memory address. (Note that this is an absolute address.) During the execution of this instruction (P) goes to processor address 0 and P holds m. Also, (d) goes to the Q register, where it is reduced by one as each central word is assembled. The original content of P is restored at the end of the instruction.

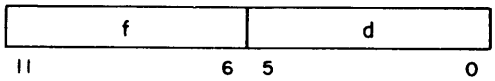
The content of P (the m portion of the instruction) gives the address of the first word to be read out of the processor memory. This word appears as the higher order 12 bits of the first 60-bit word to be stored in Central Memory.

The content of P is advanced by one to provide the next address in the processor memory as each 12-bit word is read. If P overflows, operation continues as P is advanced from 7777_8 to 0000_8. These locations will be read from as if they were consecutive.

The content of A is advanced by one to provide the next Central Memory address after each 60-bit word is assembled and Q is reduced by one. The block transfer is complete when Q = 0. This instruction will not interrupt an ECS transfer unless bit 17 of the A register is set (Access priority).

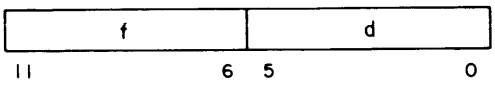
REPLACE

35 *Replace add (d)* *(12 Bits)*



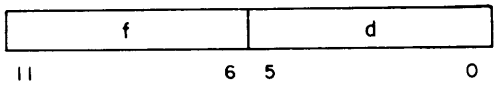
This instruction adds the quantity in location d to the contents of A and stores the lower 12 bits of the result at location d. The resultant sum is left in A at the end of the operation and the original contents of A are destroyed.

36 *Replace add one (d)* *(12 Bits)*



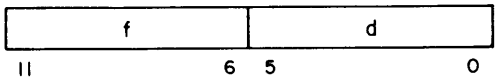
The quantity in location d is replaced by its original value plus one. The resultant sum is left in A at the end of the operation, and the original contents of A are destroyed.

37 *Replace subtract one (d)* *(12 Bits)*



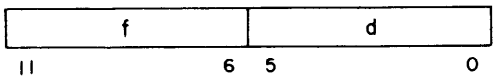
The quantity in location d is replaced by its original value minus one. The resultant difference is left in A at the end of the operation, and the original contents of A are destroyed.

45 *Replace add ((d))* *(12 Bits)*



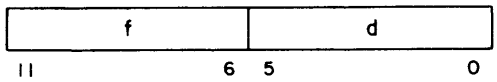
The operand which is obtained from the location specified by the contents of location d, is added to the contents of A, and the lower 12 bits of the sum replace the original operand. The resultant sum is also left in A at the end of the operation.

46 *Replace add one ((d))* *(12 Bits)*



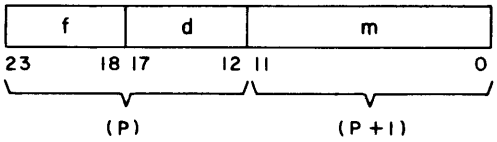
The operand, which is obtained from the location specified by the contents of location d, is replaced by its original value plus one. The resultant sum is also left in A at the end of the operation, and the original contents of A are destroyed.

47 *Replace subtract one ((d))* *(12 Bits)*



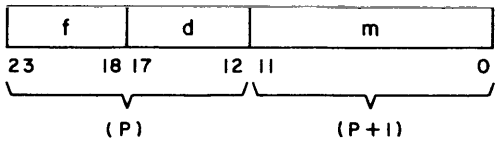
The operand, which is obtained from the location specified by the contents of location d, is replaced by its original value minus one. The resultant difference is also left in A at the end of the operation, and the original contents of A are destroyed.

55 *Replace add (m + (d))* *(24 Bits)*



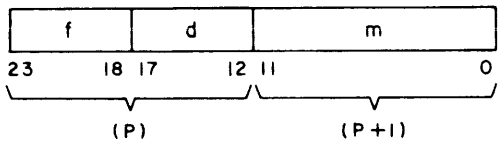
The operand, which is obtained from the location determined by indexed direct addressing, is added to the contents of A, and the lower 12 bits of the sum replace the original operand in memory. The resultant sum is also left in A at the end of the operation, and the original contents of A are destroyed.

56 *Replace add one (m + (d))* *(24 Bits)*



The operand, which is obtained from the location determined by indexed direct addressing, is replaced by its original value plus one (see instruction 50, for explanation of addressing). The resultant sum is also left in A at the end of the operation, and the original contents of A are destroyed.

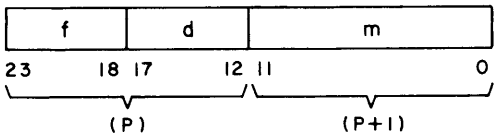
57 *Replace subtract one (m + (d))* *(24 Bits)*



The operand, which is obtained from the location determined by indexed direct addressing, is replaced by its original value minus one (see instruction 50 for explanation of addressing). The resultant difference is also left in A at the end of the operation, and the original contents of A are destroyed.

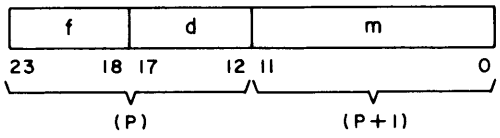
INPUT/OUTPUT

64 *Jump to m if channel d active* *(24 Bits)*



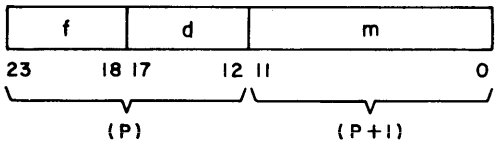
This instruction provides a conditional jump to a new program sequence beginning at an address given by the contents of m. The jump is taken if the channel specified by d is active. The current program sequence continues if the channel is inactive.

65 *Jump to m if channel d inactive* *(24 Bits)*



This instruction provides a conditional jump to a new program sequence beginning at an address given by m. The jump is taken if the channel specified by d is inactive. The current program sequence continues if the channel is active.

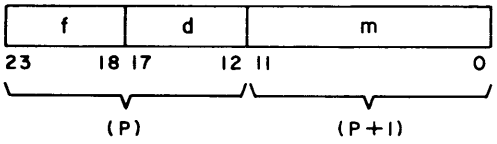
66 *Jump to m if channel d full* *(24 Bits)*



This instruction provides a conditional jump to a new program sequence beginning at an address given by m. The jump is taken if the channel designated by d is full. The present program sequence continues if the channel is empty.

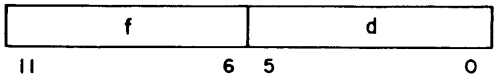
An input channel is full when the input equipment has placed a word on the channel and that word has not yet been sampled by a processor. The channel is empty when a word has been accepted. An output channel is full when a processor places a word on the channel. The channel is empty when the output equipment has sampled the word.

67 *Jump to m if channel d empty* *(24 Bits)*



This instruction provides a conditional jump to a new program sequence beginning at an address specified by m. The jump is taken if the channel specified by d is empty. The current program sequence continues if the channel is full. (See instruction 66 for the meaning of full and empty.)

70 *Input to A from channel d* *(12 Bits)*

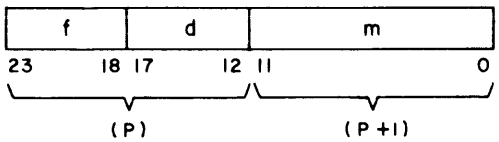


This instruction transfers a word from input channel d to the lower 12 bits of the A register. The upper 6 bits of the A register are cleared to zeros.

NOTE

This instruction will hang up the peripheral processor if executed when the channel is inactive.

71 *Input (A) words to m from channel d* *(24 Bits)*



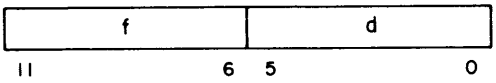
This instruction transfers a block of 12-bit words from input channel d to the processor memory. The content of A gives the block length. The first word goes to the processor address specified by m. The content of A is reduced by one as each word is read. The input operation is complete when A = 0 or the data channel becomes inactive. If the operation is terminated by the channel becoming inactive, the next location in the processor memory is set to all zeroes. However, the word count is not affected by this empty word. Therefore, the contents of the A register gives the block length minus the number of real data words actually read in.

During this instruction address 0000 temporarily holds P, while m is held in the P register. The content of P advances by one to give the address for the next word as each word is stored.

NOTE

If this instruction is executed when the data channel is inactive, no input operation is accomplished and the program continues at P + 2. However, the location specified by m is set to all zeroes.

72 *Output from A on channel d* *(12 Bits)*

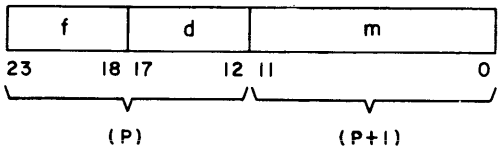


This instruction transfers a word from A (lower 12 bits) to output channel d.

NOTE

This instruction will hang up the peripheral processor if executed when the channel is inactive.

73 *Output (A) words from m on channel d* *(24 Bits)*



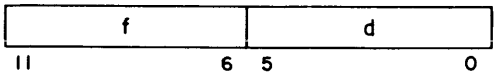
This instruction transfers a block of words from the processor memory to channel d. The first word comes from the address specified by m. The content of A specifies the number of words to be sent. The content of A is reduced by one as each word is read out. The output operation is complete when A = 0 or the channel becomes inactive.

During this instruction address 0000 temporarily holds P, while m is held in the P register. The content of P advances by one to give the address of the next word as each word is taken from memory.

NOTE

If this instruction is executed when the data channel is inactive, no output operation is accomplished and the program continues at P + 2.

74 *Activate channel d* *(12 Bits)*

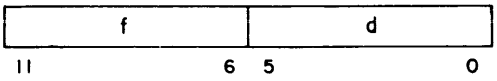


This instruction activates the channel specified by d and must precede any 70-73 instruction. Activating a channel alerts and prepares the I/O equipment for the exchange of data.

NOTE

Activating an already active channel causes the peripheral processor to hang up.

75 *Disconnect channel d* *(12 Bits)*

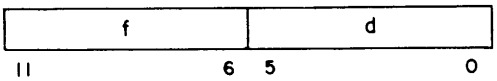


This instruction deactivates the channel specified by d. As a result, the I/O equipment stops and the buffer terminates.

NOTE

1) Do not attempt to deactivate an already inactive channel or the peripheral processor will hang up.
2) If an output instruction is followed by a disconnect instruction without first establishing that the information has been accepted by the input device (check for channel empty) the last word transmitted may be lost.
3) Do not deactivate a channel before putting a useful program in the associated processor. Processors other than 0 are hung up on an Input instruction (71). Deactivating a channel after Dead Start causes an exit to the address specified by the contents of location 0000 plus 1 and execution of that program. If the channel is deactivated without a valid program in that processor, the processor will execute what ever program was left in memory; it could, therefore, run wild.

76 *Function (A) on channel d* *(12 Bits)*

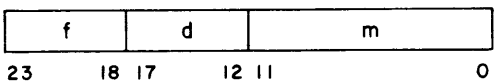


The external function code in the lower 12 bits of A is sent out on channel d.

NOTE

Do not execute this instruction when the channel is Active or the peripheral processor will hang up.

77 *Function m on channel d* *(24 Bits)*



The external function code specified by m is sent out on channel d.

COMMENT SHEET

MANUAL TITLE CONTROL DATA CYBER 70 Computer Systems
Reference Manual Instruction Descriptions Volume 2

PUBLICATION NO. 60347300 REVISION B

FROM: NAME: _____

BUSINESS ADDRESS: _____

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

AA3419 REV. 11/69 PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Technical Publications Department
4201 North Lexington Avenue
Arden Hills, Minnesota 55112

CUT ALONG LINE

FOLD

FOLD

CONTROL DATA® CYBER 70 SERIES

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS
8100 34TH AVENUE SOUTH
MINNEAPOLIS, MINNESOTA 55420

SALES OFFICES AND SERVICE CENTERS
IN MAJOR CITIES
THROUGHOUT THE WORLD