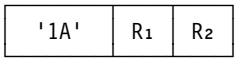
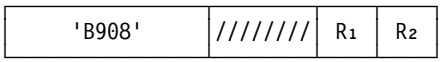
ADD

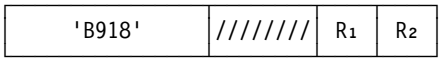
AR R1,R2 [RR]



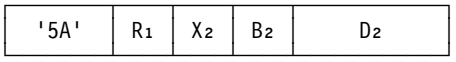
AGR R1,R2 [RRE]



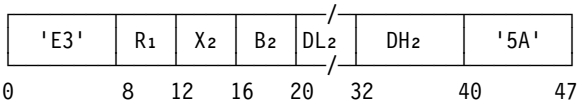
AGFR R1,R2 [RRE]



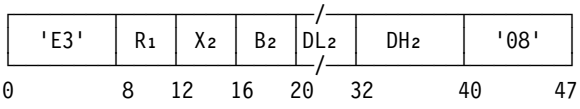
A R1,D2(X2,B2) [RX]



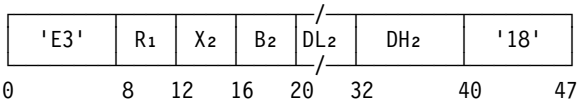
AY R1,D2(X2,B2) [RXY]



AG R1,D2(X2,B2) [RXY]



AGF R1,D2(X2,B2) [RXY]



The second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD (AR, A, and AY), the operands and the sum are treated as 32-bit signed binary integers. For ADD (AGR, AG), they are treated as 64-bit signed binary integers. For ADD (AGFR, AGF), the second operand is treated as a 32-bit signed binary integer, and the first operand and the sum are treated as 64-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for A is treated as a 12-bit unsigned binary integer. The displacement for AY, AG, and AGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

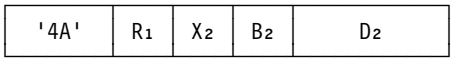
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:

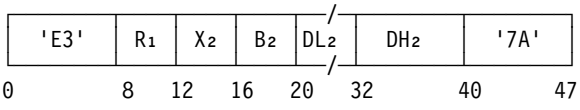
- Access (fetch, operand 2 of A, AY, AG, and AGF only)
- Fixed-point overflow
- Operation (AY, if the long-displacement facility is not installed)

ADD HALFWORD

AH R1,D2(X2,B2) [RX]

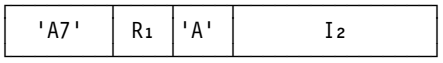


AHY R1,D2(X2,B2) [RXY]

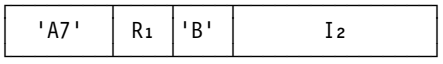


ADD HALFWORD IMMEDIATE

AHI R1,I2 [RI]



AGHI R1,I2 [RI]



The second operand is added to the first operand, and the sum is placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For ADD HALFWORD (AH, AHY) and ADD HALFWORD IMMEDIATE (AHI), the first operand and the sum are treated as 32-bit signed binary integers. For ADD HALFWORD IMMEDIATE (AGHI), they are treated as 64-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for AH is treated as a 12-bit unsigned binary integer. The displacement for AHY is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

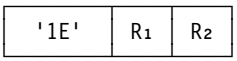
Program Exceptions:

- Access (fetch, operand 2 of AH, AHY)
- Fixed-point overflow
- Operation (AHY, if the long-displacement facility is not installed)

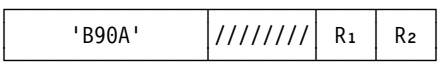
Programming Note: An example of the use of the ADD HALFWORD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

ADD LOGICAL

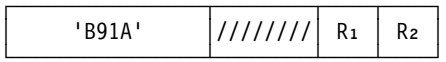
ALR R1,R2 [RR]



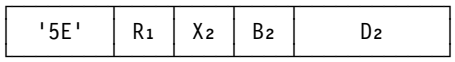
ALGR R1,R2 [RRE]



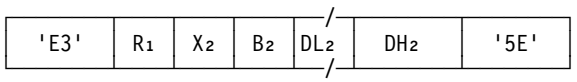
ALGFR R1,R2 [RRE]



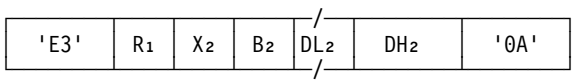
AL R1,D2(X2,B2) [RX]



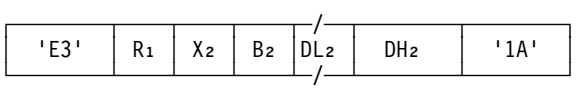
ALY R1,D2(X2,B2) [RXY]



ALG R1,D2(X2,B2) [RXY]



ALGF R1,D2(X2,B2) [RXY]



The second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD LOGICAL (ALR, AL, ALY), the operands and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL (ALGR, ALG), they are treated as 64-bit unsigned binary integers. For ADD LOGICAL (ALGFR, ALGF) the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the sum are treated as 64-bit unsigned binary integers.

The displacement for AL is treated as a 12-bit unsigned binary integer. The displacement for ALY, ALG, and ALGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

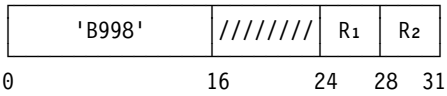
0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

Program Exceptions:

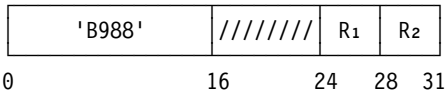
- Access (fetch, operand 2 of AL, ALY, ALG, and ALGF only)
- Operation (ALY, if the long-displacement facility is not installed)

ADD LOGICAL WITH CARRY

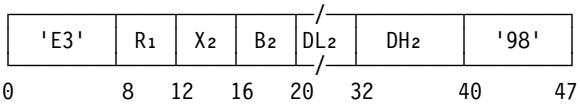
ALCR R1,R2 [RRE]



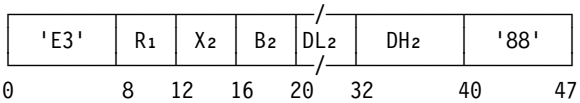
ALCGR R1,R2 [RRE]



ALC R1,D2(X2,B2) [RXY]



ALCG R1,D2(X2,B2) [RXY]



The second operand and the carry are added to the first operand, and the sum is placed at the first-operand location. For ADD LOGICAL WITH CARRY (ALCR, ALC), the operands, the carry, and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL WITH CARRY (ALCGR, ALCG), they are treated as 64-bit unsigned binary integers.

Resulting Condition Code:

0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

Program Exceptions:

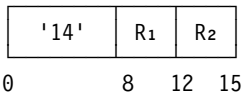
- Access (fetch, operand 2 of ALC and ALCG only)

Programming Notes:

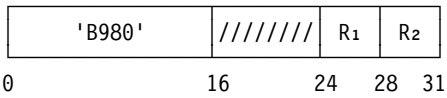
1. A carry is represented by a one value of bit 18 of the current PSW. Bit 18 is the leftmost bit of the two-bit condition code in the PSW. Bit 18 is set to one by an execution of an ADD LOGICAL or ADD LOGICAL WITH CARRY instruction that produces a carry out of bit position 0 of the result.
2. ADD and ADD LOGICAL may provide better performance than ADD LOGICAL WITH CARRY, depending on the model.

AND

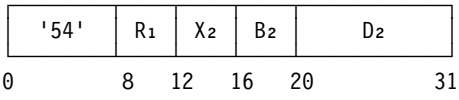
NR R1,R2 [RR]



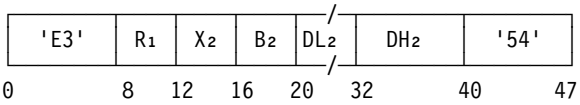
NGR R1,R2 [RRE]



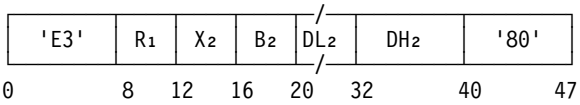
N R1,D2(X2,B2) [RX]



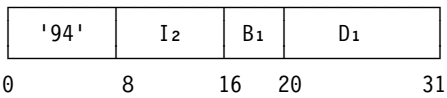
NY R1,D2(X2,B2) [RXY]



NG R1,D2(X2,B2) [RXY]



NI D1(B1),I2 [SI]



- Specification

```
1.-6.  Exceptions with the same priority as the priority of program-
       interruption conditions for the general case.

7.A    Access exceptions for second instruction halfword.

7.B    Operation exception.

8.     Specification exception due to invalid function code
       or invalid register number.

9.     Specification exception due to invalid operand length.

10.    Condition code 0 due to second-operand length originally zero.

11.    Access exceptions for an access to the parameter block, first,
       or second operand.

12.    Condition code 0 due to normal completion (second-operand
       length originally nonzero, but stepped to zero).

13.    Condition code 3 due to partial completion (second-operand
       length still nonzero).
```

Figure 7-28. Priority of Execution: KM and KMC

Programming Notes:

1. When condition code 3 is set, the general registers containing the operand addresses and length, and, for CIPHER MESSAGE WITH CHAINING, the chaining value in the parameter block, are usually updated such that the program can simply branch back to the instruction to continue the operation.

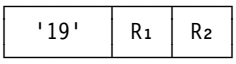
   For unusual situations, the CPU protects against endless reoccurrence of the no-progress case and also protects against setting condition code 3 when the portion of the first and second operands to be reprocessed overlap in storage. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop and no exposure to incorrectly retrying the instruction.

2. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3. For CIPHER MESSAGE WITH CHAINING, the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

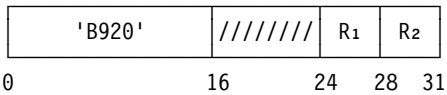
3. To save storage, the first and second operands may overlap exactly or the starting point of the first operand may be to the left of the starting point of the second operand. In either case, the overlap is not destructive.

COMPARE

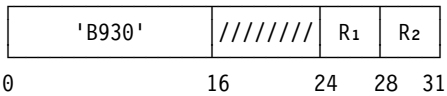
CR R1,R2 [RR]



CGR R1,R2 [RRE]



CGFR R1,R2 [RRE]



```
C       R1,D2(X2,B2)      [RX]

 +------+----+----+----+------------+
 | '59' | R1 | X2 | B2 |     D2     |
 +------+----+----+----+------------+
 0      8    12   16   20           31

CY      R1,D2(X2,B2)      [RXY]

 +------+----+----+----+-----+-----+------+
 | 'E3' | R1 | X2 | B2 | DL2 | DH2 | '59' |
 +------+----+----+----+-----+-----+------+
 0      8    12   16   20    32    40     47

CG      R1,D2(X2,B2)      [RXY]

 +------+----+----+----+-----+-----+------+
 | 'E3' | R1 | X2 | B2 | DL2 | DH2 | '20' |
 +------+----+----+----+-----+-----+------+
 0      8    12   16   20    32    40     47

CGF     R1,D2(X2,B2)      [RXY]

 +------+----+----+----+-----+-----+------+
 | 'E3' | R1 | X2 | B2 | DL2 | DH2 | '30' |
 +------+----+----+----+-----+-----+------+
 0      8    12   16   20    32    40     47
```

The first operand is compared with the second operand, and the result is indicated in the condition code. For COMPARE (CR, C, CY), the operands are treated as 32-bit signed binary integers. For COMPARE (CGR, CG), they are treated as 64-bit signed binary integers For COMPARE (CGFR, CGF), the second operand is treated as a 32-bit signed binary integer, and the first operand is treated as a 64-bit signed binary integer.

The displacement for C is treated as a 12-bit unsigned binary integer. The displacement for CY, CG, and CGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Operands equal
1 First operand low
2 First operand high
3 --

Program Exceptions:

- Access (fetch, operand 2 of C, CY, CG, and CGF only)
- Operation (CY, if the long-displacement facility is not installed)

COMPARE AND FORM CODEWORD

```
CFC     D2(B2)            [S]

 +--------+----+-----------+
 | 'B21A' | B2 |     D2    |
 +--------+----+-----------+
 0                16   20          31
```

General register 2 contains an index, which is used along with contents of general registers 1 and 3 to designate the starting addresses of two fields in storage, called the first and third operands. The first and third operands are logically compared, and a codeword is formed for use in sort/merge algorithms.

The second-operand address is not used to address data. Bits 49-62 of the second-operand address, with one rightmost and one leftmost zero appended, are used as a 16-bit index limit. Bit 63 of the second-operand address is the operand-control bit. When bit 63 is zero, the codeword is formed from the high operand; when bit 63 is one, the codeword is formed from the low operand. The remainder of the second-operand address is ignored.

General registers 1 and 3 contain the base addresses of the first and third operands. Bits 48-63 of general register 2 are used as an index for addressing both the first and third operands. General registers 1, 2, and 3 must all initially contain even values; otherwise, a specification exception is recognized.

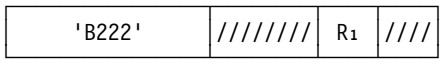
In the access-register mode, access register 1 specifies the address space containing the first and third operands.

The size of the units by which the first and third operands are compared, the size of the resulting codeword, and the participation of bits 0-31 of general registers 1, 2, and 3 in the operation depend on the addressing mode. In the 24-bit or 31-bit addressing mode, the comparison unit is two bytes, the codeword is four bytes, and bits 0-31 are ignored and remain unchanged. In the 64-bit addressing mode, the comparison unit is six bytes, the codeword is eight bytes, and bits 0-31 are used in and may be changed by the operation.

Operation in the 24-Bit or 31-Bit Addressing Mode

INSERT PROGRAM MASK

```
IPM    R1                [RRE]
```



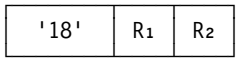
The condition code and program mask from the current PSW are inserted into bit positions 34 and 35 and 36-39, respectively, of general register R_1. Bits 32 and 33 of the register are set to zeros; bits 0-31 and 40-63 are left unchanged.

Condition Code: The code remains unchanged.

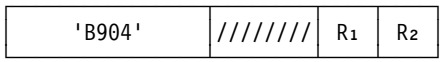
Program Exceptions: None.

LOAD

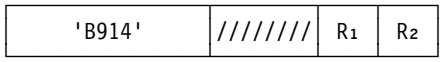
```
LR     R1,R2      [RR]
```



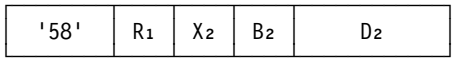

```
LGR    R1,R2      [RRE]
```



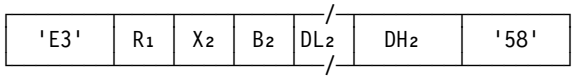

```
LGFR   R1,R2      [RRE]
```



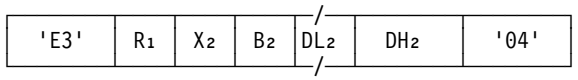

```
L      R1,D2(X2,B2)      [RX]
```



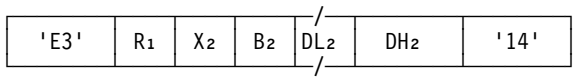

```
| LY     R1,D2(X2,B2)      [RXY]
```




```
LG     R1,D2(X2,B2)      [RXY]
```




```
LGF    R1,D2(X2,B2)      [RXY]
```



The second operand is placed unchanged at the first-operand location, except that, for LOAD (LGFR, LGF), it is sign extended.

For LOAD (LR, L, LY), the operands are 32 bits, and, for LOAD (LGR, LG), the operands are 64 bits. For LOAD (LGFR, LGF), the second operand is treated as a 32-bit signed binary integer, and the first operand is treated as a 64-bit signed binary integer.

The displacement for L is treated as a 12-bit unsigned binary integer. The displacement for LY, LG, and LGF is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

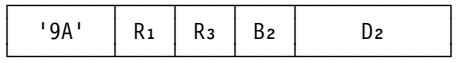
Program Exceptions:

- Access (fetch, operand 2 of L, LY, LG, and LGF only)
- Operation (LY, if the long-displacement facility is not installed)

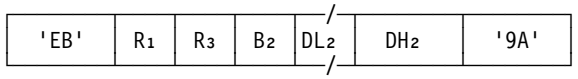
Programming Note: An example of the use of the LOAD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

LOAD ACCESS MULTIPLE

```
LAM    R1,R3,D2(B2)      [RS]
```




```
| LAMY   R1,R3,D2(B2)      [RSY]
```



The set of access registers starting with access register R_1 and ending with access register R_3 is loaded from the locations designated by the second-operand address.

The storage area from which the contents of the access registers are obtained starts at the location designated by the second-operand address and continues through as many storage words as the number of access registers specified. The access registers are loaded in ascending order of their register numbers, starting with access register R_1 and continuing up to and including access register R_3, with access register 0 following access register 15.

The displacement for LAM is treated as a 12-bit unsigned binary integer. The displacement for LAMY is treated as a 20-bit signed binary integer.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

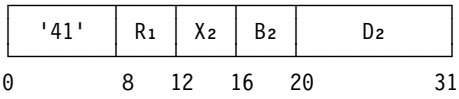
Condition Code: The code remains unchanged.

Program Exceptions:

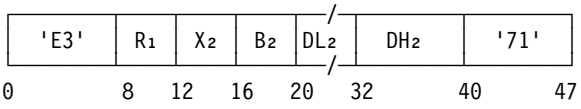
- Access (fetch, operand 2)
- Operation (LAMY, if the long-displacement facility is not installed)
- Specification

LOAD ADDRESS

LA R1,D2(X2,B2) [RX]



LAY R1,D2(X2,B2) [RXY]



The address specified by the X_2, B_2, and D_2 fields is placed in general register R_1. The address computation follows the rules for address arithmetic.

In the 24-bit addressing mode, the address is placed in bit positions 40-63, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The displacement for LA is treated as a 12-bit unsigned binary integer. The displacement for LAY is treated as a 20-bit signed binary integer.

No storage references for operands take place, and the address is not inspected for access exceptions.

Condition Code: The code remains unchanged.

Program Exceptions:

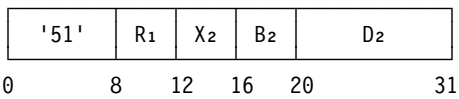
- Operation (LAY if the long-displacement facility is not installed)

Programming Notes:

1. An example of the use of the LOAD ADDRESS instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. LOAD ADDRESS may be used to increment the rightmost bits of a general register, other than register 0, by the contents of the D_2 field of the instruction. LOAD ADDRESS (LAY) may also be used to decrement the rightmost bits of a register, other than register 0. The register to be incremented should be designated by R_1 and by either X_2 (with B_2 set to zero) or B_2 (with X_2 set to zero). The instruction updates 24 bits in the 24-bit addressing mode, 31 bits in the 31-bit addressing mode, and 64 bits in the 64-bit addressing mode.

LOAD ADDRESS EXTENDED

LAE R1,D2(X2,B2) [RX]



The address specified by the X_2, B_2, and D_2 fields is placed in general register R_1. Access register R_1 is loaded with a value that depends on the current value of the address-space-control bits, bits 16 and 17 of the PSW. If the address-space-control bits are 01 binary, the value placed

in the access register also depends on whether the B_2 field is zero or nonzero.

The address computation follows the rules for address arithmetic. In the 24-bit addressing mode, the address is placed in bit positions 40-63 of general register R_1, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The value placed in access register R_1 is as shown in the following table:

PSW Bits 16 and 17	Value Placed in Access Register R_1
00	00000000 hex (zeros in bit positions 0-31)
10	00000001 hex (zeros in bit positions 0-30 and one in bit position 31)
01	If B_2 field is zero: 00000000 hex (zeros in bit positions 0-31) If B_2 field is nonzero: Contents of access register B_2
11	00000002 hex (zeros in bit positions 0-29 and 31, and one in bit position 30)

However, when PSW bits 16 and 17 are 01 binary and the B_2 field is nonzero, bit positions 0-6 of access register B_2 must contain all zeros; otherwise, the results in general register R_1 and access register R_1 are unpredictable.

No storage references for operands take place, and the address is not inspected for access exceptions.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

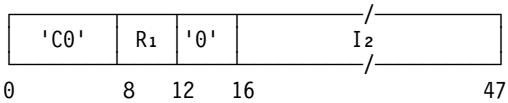
1. When DAT is on, the different values of the address-space-control bits correspond to translation modes as follows:

PSW Bits 16 and 17	Translation Mode
00	Primary-space mode
10	Secondary-space mode
01	Access-register mode
11	Home-space mode

2. In the access-register mode, the value 00000000 hex in an access register designates the primary address space, and the value 00000001 hex designates the secondary address space. The value 00000002 hex designates the home address space if the control program assigns entry 2 of the dispatchable-unit access list as designating the home address space and places a zero access-list-entry sequence number (ALESN) in that entry.

LOAD ADDRESS RELATIVE LONG

```
LARL     R1,I2     [RIL]
```



The address specified by the I_2 field is placed in general register R_1. The address computation follows the rules for the branch address of BRANCH RELATIVE ON CONDITION LONG and BRANCH RELATIVE AND SAVE LONG.

In the 24-bit addressing mode, the address is placed in bit positions 40-63, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The contents of the I_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the computed address.

No storage references for operands take place, and the address is not inspected for access exceptions.

Condition Code: The code remains unchanged.

Program Exceptions: None.

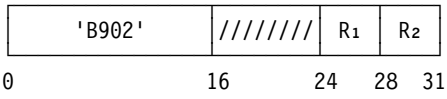
Programming Notes:

1. Only even addresses (halfword addresses) can be generated. If an odd address is desired, LOAD ADDRESS can be used to add one to an address formed by LOAD ADDRESS RELATIVE LONG.
2. When LOAD ADDRESS RELATIVE LONG is the target of EXECUTE, the address produced is relative to the location of the LOAD ADDRESS RELATIVE LONG instruction, not of the EXECUTE instruction. This is consistent with the operation of the relative-branch instructions.

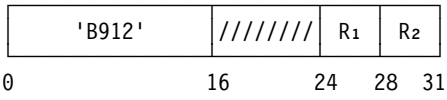
LOAD AND TEST

LTR R1,R2 [RR]

LTGR R1,R2 [RRE]



LTGFR R1,R2 [RRE]



The second operand is placed unchanged at the first-operand location, except that, for LOAD AND TEST (LTGFR), it is sign extended. The sign and magnitude of the second operand, treated as a signed binary integer, are indicated in the condition code.

For LOAD AND TEST (LTR), the operands are 32 bits, and, for LOAD AND TEST (LTGR), the operands are 64 bits. For LOAD AND TEST (LTGFR), the second operand is 32 bits, and the first operand is treated as a 64-bit signed binary integer.

Resulting Condition Code:

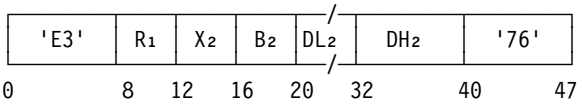
0 Result zero
1 Result less than zero
2 Result greater than zero
3 --

Program Exceptions: None.

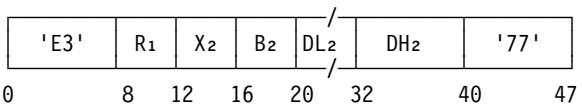
Programming Note: For LOAD AND TEST (LTR and LTGR) when the R_1 and R_2 fields designate the same register, the operation is equivalent to a test without data movement.

LOAD BYTE

LB R1,D2(X2,B2) [RXY]



LGB R1,D2(X2,B2) [RXY]



The second operand is sign extended and placed at the first-operand location. The second operand is one byte in length and is treated as an eight-bit signed binary integer. For LOAD BYTE (LB), the first operand is treated as a 32-bit signed binary integer. For LOAD BYTE (LGB), the first operand is treated as a 64-bit signed binary integer.

The displacement is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

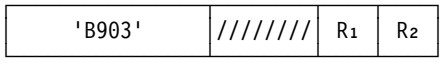
Program Exceptions:

- Access (fetch, operand 2)
- Operation (if the long-displacement facility is not installed)

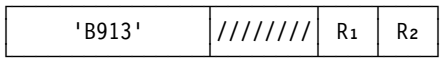
LOAD COMPLEMENT

LCR R1,R2 [RR]

```
LCGR     R1,R2     [RRE]
```




```
LCGFR    R1,R2     [RRE]
```



The two's complement of the second operand is placed at the first-operand location. For LOAD COMPLEMENT (LCR), the second operand and result are treated as 32-bit signed binary integers. For LOAD COMPLEMENT (LCGR), they are treated as 64-bit signed binary integers. For LOAD COMPLEMENT (LCGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

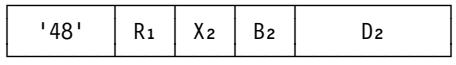
Program Exceptions:

- Fixed-point overflow

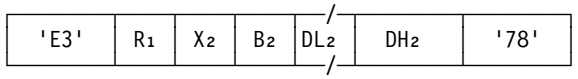
Programming Note: The operation complements all numbers. Zero remains unchanged. For LCR or LCGR, the maximum negative 32-bit number or 64-bit number, respectively, remains unchanged, and an overflow condition occurs when the number is complemented. LCGFR complements the maximum negative 32-bit number without recognizing overflow.

LOAD HALFWORD

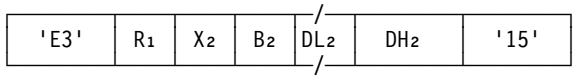
```
LH       R1,D2(X2,B2)      [RX]
```




```
LHY      R1,D2(X2,B2)      [RXY]
```

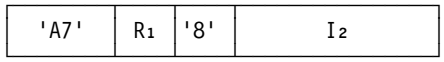



```
LGH      R1,D2(X2,B2)      [RXY]
```

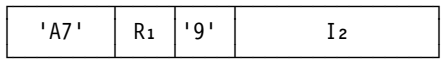


LOAD HALFWORD IMMEDIATE

```
LHI      R1,I2     [RI]
```




```
LGHI     R1,I2     [RI]
```



The second operand is sign extended and placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For LOAD HALFWORD (LH, LHY) and LOAD HALFWORD IMMEDIATE (LHI), the first operand is treated as a 32-bit signed binary integer. For LOAD HALFWORD (LGH) and LOAD HALFWORD IMMEDIATE (LGHI), the first operand is treated as a 64-bit signed binary integer.

The displacement for LH is treated as a 12-bit unsigned binary integer. The displacement for LHY and LGH is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LH, LHY, and LGH)

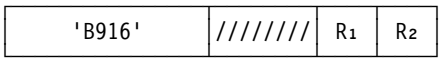
Program Exceptions:

- Access (fetch, operand 2 of LH and LHY)
- Operation (LHY, if the long-displacement facility is not installed)

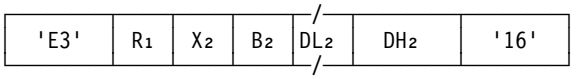
Programming Note: An example of the use of the LOAD HALFWORD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

LOAD LOGICAL

LLGFR R1,R2 [RRE]



LLGF R1,D2(X2,B2) [RXY]



The four-byte second operand is placed in bit positions 32-63 of general register R_1, and zeros are placed in bit positions 0-31 of general register R_1.

For LOAD LOGICAL (LLGFR), the second operand is in bit positions 32-63 of general register R_2.

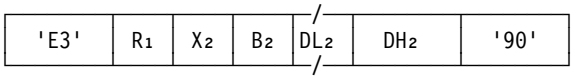
Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LLGF only)

LOAD LOGICAL CHARACTER

LLGC R1,D2(X2,B2) [RXY]



The one-byte second operand is placed in bit positions 56-63 of general register R_1, and zeros are placed in bit positions 0-55 of general register R_1.

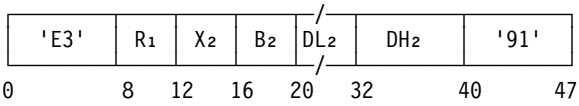
Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)

LOAD LOGICAL HALFWORD

LLGH R1,D2(X2,B2) [RXY]



The two-byte second operand is placed in bit positions 48-63 of general register R_1, and zeros are placed in bit positions 0-47 of general register R_1.

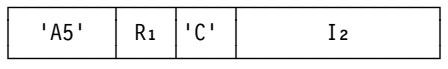
Condition Code: The code remains unchanged.

Program Exceptions:

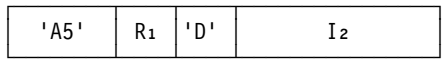
- Access (fetch, operand 2)

LOAD LOGICAL IMMEDIATE

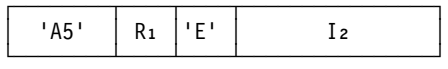
LLIHH R1,I2 [RI]



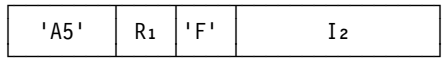
LLIHL R1,I2 [RI]



LLILH R1,I2 [RI]



LLILL R1,I2 [RI]



The second operand is placed in bit positions of the first operand. The remainder of the first operand is set to zeros.

For each instruction, the bit positions of the first operand that are loaded with the second operand are as follows:

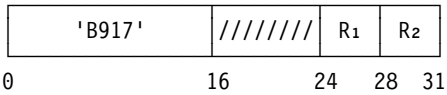
Instruction	Bit Positions Loaded
LLIHH	0-15
LLIHL	16-31
LLILH	32-47
LLILL	48-63

Condition Code: The code remains unchanged.

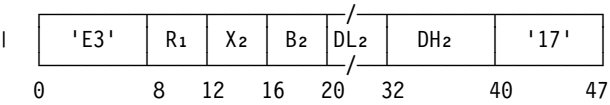
Program Exceptions: None.

LOAD LOGICAL THIRTY ONE BITS

LLGTR R1,R2 [RRE]



LLGT R1,D2(X2,B2) [RXY]



For LLGTR, bits 33-63 of general register R_2, with 33 zeros appended on the left, are placed in general register R_1. For LLGT, bits 1-31 of the four bytes at the second-operand location, with 33 zeros appended on the left, are placed in general register R_1.

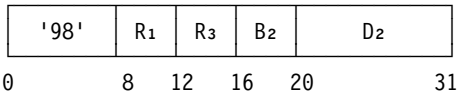
Condition Code: The code remains unchanged.

Program Exceptions:

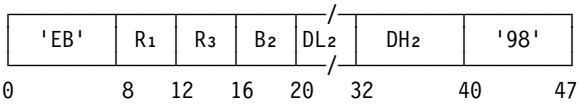
- Access (fetch, operand 2 of LLGT only)

LOAD MULTIPLE

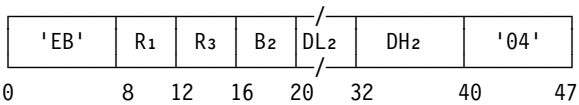
LM R1,R3,D2(B2) [RS]



LMY R1,R3,D2(B2) [RSY]



LMG R1,R3,D2(B2) [RSY]



Bit positions of the set of general registers starting with general register R_1 and ending with general register R_3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed.

For LOAD MULTIPLE (LM, LMY), bit positions 32-63 of the general registers are loaded from successive four-byte fields beginning at the second-operand address, and bits 0-31 of the registers remain unchanged. For LOAD MULTIPLE (LMG), bit positions 0-63 of the general registers are loaded from successive eight-byte fields beginning at the second-operand address.

The general registers are loaded in the ascending order of their register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15.

The displacement for LM is treated as a 12-bit unsigned binary integer. The displacement for LMY and LMG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

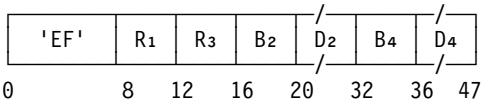
Program Exceptions:

- Access (fetch, operand 2)
- Operation (LMY, if the long-displacement facility is not installed)

Programming Note: All combinations of register numbers specified by R_1 and R_3 are valid. When the register numbers are equal, only four bytes, for LM or LMY or eight bytes, for LMG, are transmitted. When the number specified by R_3 is less than the number specified by R_1, the register numbers wrap around from 15 to 0.

LOAD MULTIPLE DISJOINT

LMD R1,R3,D2(B2),D4(B4) [SS]



Bit positions 0-31 of the set of general registers starting with general register R_1 and ending with general register R_3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed. Bit positions 32-63 of the same registers are similarly loaded from storage beginning at the location designated by the fourth-operand address.

The general registers are loaded in the ascending order of their register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15.

Condition Code: The code remains unchanged.

Program Exceptions:

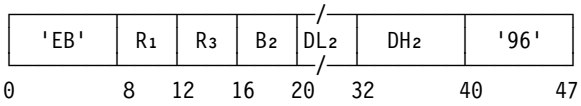
- Access (fetch, operands 2 and 4)

Programming Notes:

1. All combinations of register numbers specified by R_1 and R_3 are valid. When the register numbers are equal, only eight bytes are transmitted. When the number specified by R_3 is less than the number specified by R_1, the register numbers wrap around from 15 to 0.
2. The second-operand and fourth-operand addresses are computed before the contents of any register are changed.
3. The combination of a LOAD MULTIPLE instruction and a LOAD MULTIPLE HIGH instruction provides equal or better performance than a LOAD MULTIPLE DISJOINT instruction for the same register range. LOAD MULTIPLE DISJOINT is for use when the second or fourth operand must be addressed by means of one of the registers loaded.

LOAD MULTIPLE HIGH

LMH R1,R3,D2(B2) [RSY]



The high-order halves, bit positions 0-31, of the set of general registers starting with general register R_1 and ending with general register R_3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed, that is, bit positions 0-31 are loaded from successive four-byte fields beginning at the second-operand address. Bits 32-63 of the registers remain unchanged.

The general registers are loaded in the ascending order of their register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15.

Condition Code: The code remains unchanged.

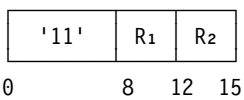
Program Exceptions:

- Access (fetch, operand 2)

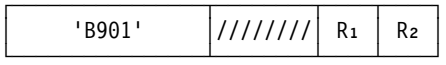
Programming Note: All combinations of register numbers specified by R_1 and R_3 are valid. When the register numbers are equal, only four bytes are transmitted. When the number specified by R_3 is less than the number specified by R_1, the register numbers wrap around from 15 to 0.

LOAD NEGATIVE

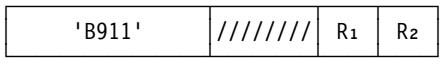
LNR R1,R2 [RR]



LNGR R1,R2 [RRE]



LNGFR R1,R2 [RRE]



The two's complement of the absolute value of the second operand is placed at the first-operand location. For LOAD NEGATIVE (LNR), the second operand and result are treated as 32-bit signed binary integers, and, for LOAD NEGATIVE (LNGR), they are treated as 64-bit signed binary integers. For LOAD NEGATIVE (LNGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

Resulting Condition Code:

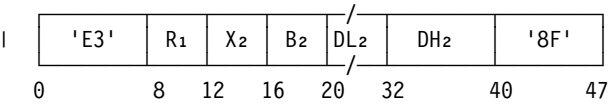
0 Result zero
1 Result less than zero
2 --
3 --

Program Exceptions: None.

Programming Note: The operation complements positive numbers; negative numbers remain unchanged. The number zero remains unchanged.

LOAD PAIR FROM QUADWORD

LPQ R1,D2(X2,B2) [RXY]



The quadword second operand is loaded into the first-operand location. The second operand appears to be fetched with quadword concurrency as observed by other CPUs. The left doubleword of the quadword is loaded into general register R1, and the right doubleword is loaded into general register R1 + 1.

The R1 field designates an even-odd pair of general registers and must designate an even-numbered register. The second operand must be designated on a quadword boundary. Otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Specification

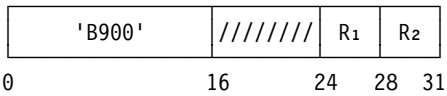
Programming Notes:

1. The LOAD MULTIPLE (LM or LMG) instruction does not necessarily provide quadword-concurrent access.
2. The performance of LOAD PAIR FROM QUADWORD on some models may be significantly slower than that of LOAD MULTIPLE (LMG). Unless quadword consistency is required, LMG should be used instead of LPQ.

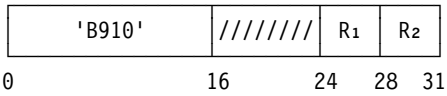
LOAD POSITIVE

LPR R1,R2 [RR]

LPGR R1,R2 [RRE]



LPGFR R1,R2 [RRE]



The absolute value of the second operand is placed at the first-operand location. For LOAD POSITIVE (LPR), the second operand and result are treated as 32-bit signed binary integers, and, for LOAD POSITIVE (LPGR), they are treated as 64-bit signed binary integers. For LOAD POSITIVE (LPGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and

ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

0 Result zero; no overflow
1 --
2 Result greater than zero; no overflow
3 Overflow

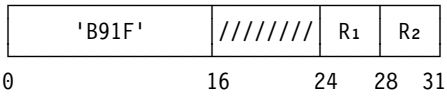
Program Exceptions:

- Fixed-point overflow

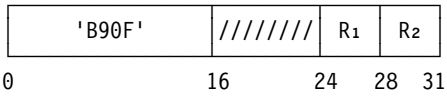
Programming Note: The operation complements negative numbers; positive numbers and zero remain unchanged. For LPR or LPGR, an overflow condition occurs when the maximum negative 32-bit number or 64-bit number, respectively, is complemented; the number remains unchanged. LPGFR complements the maximum negative 32-bit number without recognizing overflow.

LOAD REVERSED

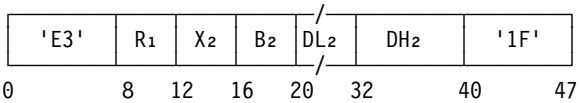
LRVR R1,R2 [RRE]



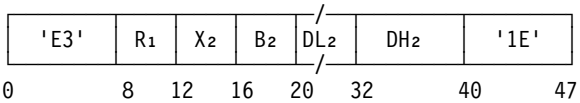
LRVGR R1,R2 [RRE]



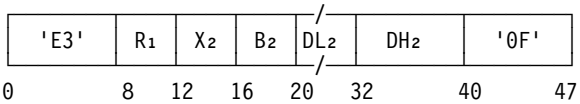
LRVH R1,D2(X2,B2) [RXY]



LRV R1,D2(X2,B2) [RXY]



LRVG R1,D2(X2,B2) [RXY]



The second operand is placed at the first-operand location with the left-to-right sequence of the bytes reversed.

For LOAD REVERSED (LRVH), the second operand is two bytes, the result is placed in bit positions 48-63 of general register R_1, and bits 0-47 of the register remain unchanged.

For LOAD REVERSED (LRVR, LRV), the second operand is four bytes, the result is placed in bit positions 32-63 of general register R_1, and bits 0-31 of the register remain unchanged. For LOAD REVERSED (LRVR), the second operand is in bit positions 32-63 of general register R_2.

For LOAD REVERSED (LRVGR, LRVG), the second operand is eight bytes.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LRVH, LRV, LRVG only)

Programming Notes:

1. The instruction can be used to convert two, four, or eight bytes from a “little-endian” format to a “big-endian” format, or vice versa. In the big-endian format, the bytes in a left-to-right sequence are in the order most significant to least significant. In the little-endian format, the bytes are in the order least significant to most significant. For example, the bytes ABCD in the big-endian format are DCBA in the little-endian format.
2. LOAD REVERSED (LRVR) can be used with a two-byte value already in a register as shown in the following example. In the example, the two bytes of interest are in bit positions 48-63 of the R1 register.

```
LRVR   R1,R1
SRA    R1,16
```

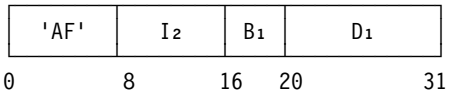
   The LOAD REVERSED instruction places the two bytes of interest in bit positions 32-47 of the register, with the order of the bytes reversed. The SHIFT RIGHT SINGLE (SRA) instruction shifts the two bytes to bit positions 48-63 of the register and extends them on their left, in bit positions 32-47, with their sign bit. The instruction SHIFT RIGHT SINGLE

LOGICAL (SRL) should be used, instead, if the two bytes of interest are unsigned.

3. The storage-operand references of LOAD REVERSED may be multiple-access references. (See "Storage-Operand Consistency" on page 5-87.)

MONITOR CALL

```
MC      D1(B1),I2         [SI]
```



A program interruption is caused if the appropriate monitor-mask bit in control register 8 is one.

The monitor-mask bits are in bit positions 48-63 of control register 8, which correspond to monitor classes 0-15, respectively.

Bit positions 12-15 in the I_2 field contain a binary number specifying one of 16 monitoring classes. When the monitor-mask bit corresponding to the class specified by the I_2 field is one, a monitor-event program interruption occurs. The contents of the I_2 field are stored at location 149, with zeros stored at location 148. Bit 9 of the program-interruption code is set to one.

The first-operand address is not used to address data; instead, the address specified by the B_1 and D_1 fields forms the monitor code, which is placed in the doubleword at location 176. Address computation follows the rules of address arithmetic; in the 24-bit addressing mode, bits 0-39 are set to zeros; in the 31-bit addressing mode, bits 0-32 are set to zeros.

When the monitor-mask bit corresponding to the class specified by bits 12-15 of the instruction is zero, no interruption occurs, and the instruction is executed as a no-operation.

Bit positions 8-11 of the instruction must contain zeros; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

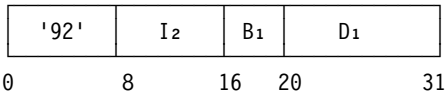
- Monitor event
- Specification

Programming Notes:

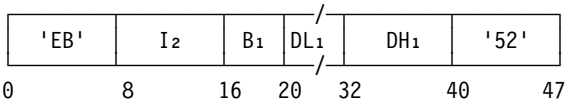
1. MONITOR CALL provides the capability for passing control to a monitoring program when selected points are reached in the monitored program. This is accomplished by implanting MONITOR CALL instructions at the desired points in the monitored program. This function may be useful in performing various measurement functions; specifically, tracing information can be generated indicating which programs were executed, counting information can be generated indicating how often particular programs were used, and timing information can be generated indicating the amount of time a particular program required for execution.
2. The monitor masks provide a means of disallowing all monitor-event program interruptions or allowing monitor-event program interruptions for all or selected classes.
3. The monitor code provides a means of associating descriptive information, in addition to the class number, with each MONITOR CALL. Without the use of a base register, up to 4,096 distinct monitor codes can be associated with a monitoring interruption. With the base register designated by a nonzero value in the B_1 field, each monitoring interruption can be identified by a 24-bit, 31-bit, or 64-bit code, depending on the addressing mode.

MOVE

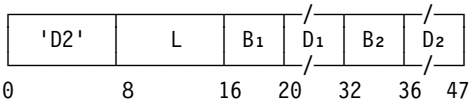
```
MVI     D1(B1),I2         [SI]
```




```
MVIY    D1(B1),I2         [SIY]
```



```
MVC    D1(L,B1),D2(B2)          [SS]
```



The second operand is placed at the first-operand location.

For MOVE (MVC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand byte.

For MOVE (MVI, MVIY), the first operand is one byte in length, and only one byte is stored.

The displacements for MVI and both operands of MVC are treated as 12-bit unsigned binary integers. The displacement for MVIY is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

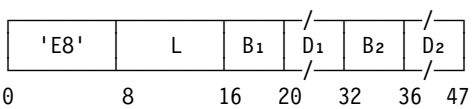
- Access (fetch, operand 2 of MVC; store, operand 1, MVI, MVIY, and MVC)
- Operation (MVIY, if the long-displacement facility is not installed)

Programming Notes:

1. Examples of the use of the MOVE instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. It is possible to propagate one byte through an entire field by having the first operand start one byte to the right of the second operand.

MOVE INVERSE

```
MVCIN  D1(L,B1),D2(B2)          [SS]
```



The second operand is placed at the first-operand location with the left-to-right sequence of the bytes inverted.

The first-operand address designates the leftmost byte of the first operand. The second-operand address designates the rightmost byte of the second operand. Both operands have the same length.

The result is obtained as if the second operand were processed from right to left and the first operand from left to right. The second operand may wrap around from location 0 to location $2^{24} - 1$ in the 24-bit addressing mode, to location $2^{31} - 1$ in the 31-bit addressing mode, or to location $2^{64} - 1$ in the 64-bit addressing mode. The first operand may wrap around from location $2^{24} - 1$ to location 0 in the 24-bit addressing mode, from location $2^{31} - 1$ to location 0 in the 31-bit addressing mode, or from location $2^{64} - 1$ to location 0 in the 64-bit addressing mode.

When the operands overlap by more than one byte, the contents of the overlapped portion of the result field are unpredictable.

Condition Code: The code remains unchanged.

Program Exceptions:

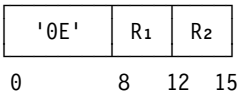
- Access (fetch, operand 2; store, operand 1)

Programming Notes:

1. An example of the use of the MOVE INVERSE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The contents of each byte moved remain unchanged.
3. MOVE INVERSE is the only SS-format instruction for which the second-operand address designates the rightmost, instead of the leftmost, byte of the second operand.
4. The storage-operand references for MOVE INVERSE may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

MOVE LONG

```
MVCL   R1,R2      [RR]
```



The second operand is placed at the first-operand location, provided overlapping of operand locations would not affect the final contents of the first-operand location. The remaining rightmost byte positions, if any, of the first-operand location are filled with padding bytes.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R_1 and R_2, respectively. The number of bytes in the first-operand and second-operand locations is specified by unsigned binary integers in bit positions 40-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively. Bit positions 32-39 of general register $R_2 + 1$ contain the padding byte. The contents of bit positions 0-39 of general register $R_1 + 1$ and of bit positions 0-31 of general register $R_2 + 1$ are ignored.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-68 on page 7-124.

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified by bits 40-63 of general register $R_1 + 1$ have been moved into the first-operand location. If the second operand is shorter than the first operand, the remaining rightmost bytes of the first-operand location are filled with the padding byte.

As part of the execution of the instruction, the values of the two length fields are compared for the setting of the condition code, and a check is made for destructive overlap of the operands. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it, assuming the inspection for overlap is performed by the use of logical operand addresses. When the operands overlap destructively, no movement takes place, and condition code 3 is set.

Operands do not overlap destructively, and movement is performed, if the leftmost byte of the first operand does not coincide with any of the second-operand bytes participating in the operation other than the leftmost byte of the second operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand bytes in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of bytes in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.

In the access-register mode, the contents of access register R_1 and access register R_2 are compared. If the R_1 or R_2 field is zero, 32 zeros are used rather than the contents of access register 0. If all 32 bits of the compared values are equal, then the destructive overlap test is made. If all 32 bits of the compared values are not equal, destructive overlap is declared not to exist. If, for this case, the operands actually overlap in real storage, it is unpredictable whether the result reflects the overlap condition.

When the length specified by bits 40-63 of general register $R_1 + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

The execution of the instruction is interruptible. When an interruption occurs, other than one that

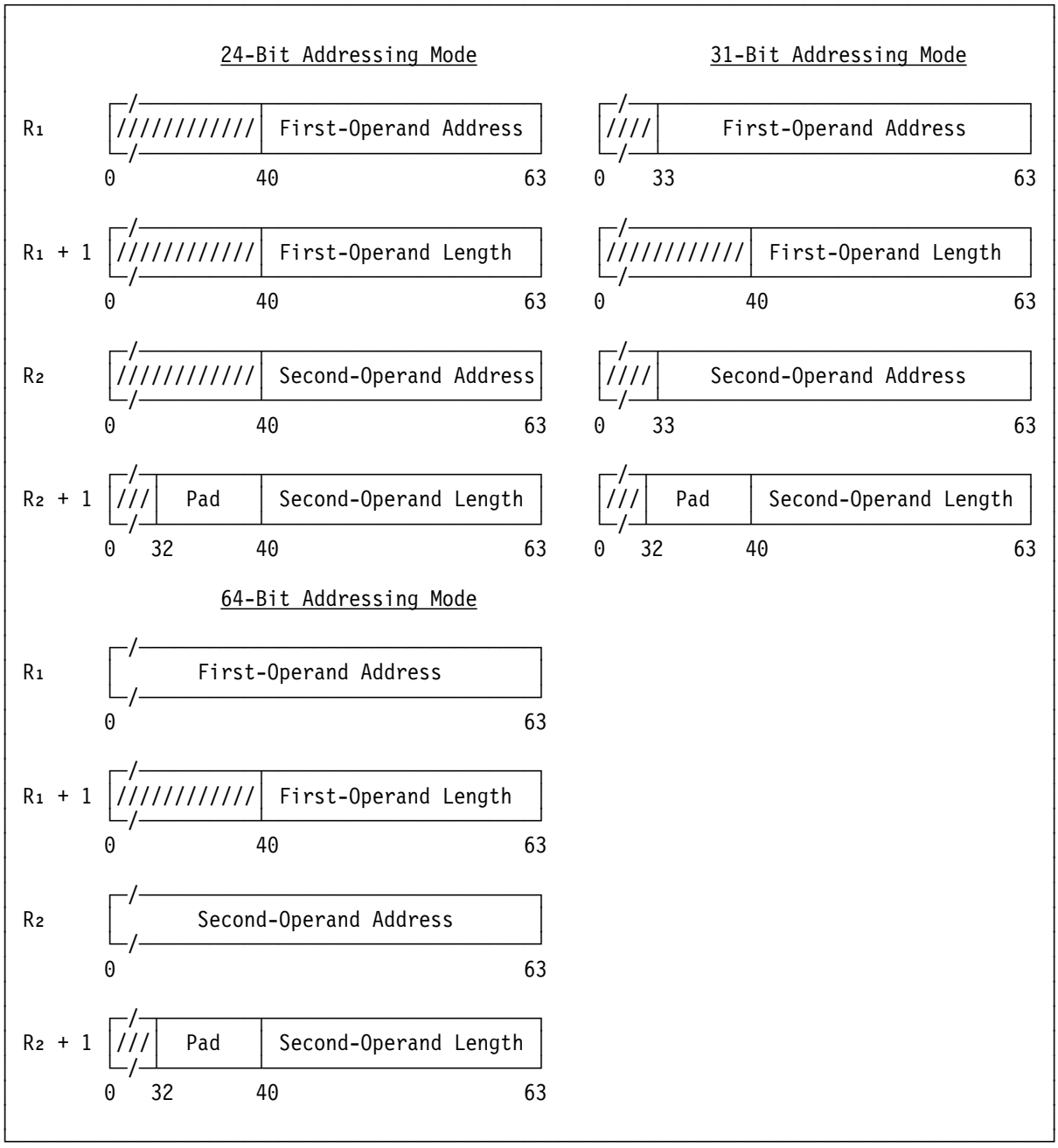


Figure 7-68. Register Contents for MOVE LONG

follows termination, the lengths in general registers $R_1 + 1$ and $R_2 + 1$ are decremented by the number of bytes moved, and the addresses in general registers R_1 and R_2 are incremented by the same number, so that the instruction, when reexecuted, resumes at the point of interruption. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_2 are set to zeros, and the contents of bit positions 0-31 remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged; and the condition code is unpredictable. If the operation is interrupted during padding, the length field in general register $R_2 + 1$ is 0, the address in general register R_2 is incremented by the original length in general register $R_2 + 1$, and general registers R_1 and $R_1 + 1$ reflect the extent of the padding operation.

When the first-operand location includes the location of the instruction or of EXECUTE, the instruction may be refetched from storage and

reinterpreted even in the absence of an interruption during execution. The exact point in the execution at which such a refetch occurs is unpredictable.

Padding byte values of B0 hex and B8 hex may be used during the nonpadding part of the operation by some models, in certain cases, as an indication of whether the movement should be performed bypassing the cache or using the cache, respectively. Thus, a padding byte of B0 hex indicates no intention to reference the destination area after the move, and a padding byte of B8 hex indicates an intention to reference the destination area.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG are single-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs, unless the padding byte is B1 hex. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by channel programs, that portion of the first operand which is filled with the padding byte is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. The length in general register $R_2 + 1$ is decremented by the number of bytes moved out of the second-operand location, and the address in general register R_2 is incremented by the same amount. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_2 are set to zeros, even when one or both of the original length values are zeros or when condition code 3 is set. The contents of bit positions 0-31 of the registers remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged.

When condition code 3 is set, no exceptions associated with operand access are recognized. When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the second operand is longer than the first operand, access exceptions are not recognized for the part of the second-operand field that is in excess of the first-operand field. For operands longer than 2K bytes, access exceptions are not recognized for locations more than 2K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R_1 field is odd, PER storage-alteration events are not recognized, and no change bits are set.

Resulting Condition Code:

0 Operand lengths equal; no destructive overlap
1 First-operand length low; no destructive overlap
2 First-operand length high; no destructive overlap
3 No movement performed because of destructive overlap

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Specification

Programming Notes:

1. An example of the use of the MOVE LONG instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. MOVE LONG may be used for clearing storage by setting the padding byte to zero and the second-operand length to zero. On most models, this is the fastest instruction for clearing storage areas in excess of 256 bytes. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-87.
3. The program should avoid specification of a length for either operand which would result in an addressing exception. Addressing (and also protection) exceptions may result in ter-

mination of the entire operation, not just the current unit of operation. The termination may be such that the contents of all result fields are unpredictable; in the case of MOVE LONG, this includes the condition code and the two even-odd general-register pairs, as well as the first-operand location in main storage. The following are situations that have actually occurred on one or more models:

a. When a protection exception occurs on a 4K-byte block of a first operand which is several blocks in length, stores to the protected block are suppressed. However, the move continues into the subsequent blocks of the first operand, which are not protected. Similarly, an addressing exception on a block does not necessarily suppress processing of subsequent blocks which are available.

b. Some models may update the general registers only when an external, I/O, repressible machine-check, or restart interruption occurs, or when a program interruption occurs for which it is required to nullify or suppress a unit of operation. Thus, if, after a move into several blocks of the first operand, an addressing or protection exception occurs, the general registers may remain unchanged.

4. When the first-operand length is zero, the operation consists in setting the condition code and, in the 24-bit or 31-bit addressing mode, of setting the leftmost bits in bit positions 32-63 of general registers R_1 and R_2 to zero.

5. When the contents of the R_1 and R_2 fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by twice the number of bytes moved. Condition code 0 is set.

6. The following is a detailed description of those cases in which movement takes place, that is, where destructive overlap does not exist.

   In the access-register mode, the contents of the access registers used are called the effective space designations. When the effective space designations are not equal, destructive overlap is declared not to exist and movement occurs. When the effective space designations are the same or when not in the access-register mode, then the following cases apply.

   Depending on whether the second operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$, depending on the addressing mode) to location 0, movement takes place in the following cases:

   a. When the second operand does not wrap around, movement is performed if the leftmost byte of the first operand coincides with or is to the left of the leftmost byte of the second operand, *or* if the leftmost byte of the first operand is to the right of the rightmost second-operand byte participating in the operation.

   b. When the second operand wraps around, movement is performed if the leftmost byte of the first operand coincides with or is to the left of the leftmost byte of the second operand, *and* if the leftmost byte of the first operand is to the right of the rightmost second-operand byte participating in the operation.

   The rightmost second-operand byte is determined by using the smaller of the first-operand and second-operand lengths.

   When the second-operand length is one or zero, destructive overlap cannot exist.

7. Special precautions should be taken if MOVE LONG is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

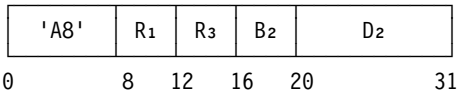
8. Since the execution of MOVE LONG is interruptible, the instruction cannot be used for situations where the program must rely on uninterrupted execution of the instruction. Similarly, the program should normally not let the first operand of MOVE LONG include the location of the instruction or of EXECUTE because the new contents of the location may be interpreted for a resumption after an interruption, or the instruction may be refetched without an interruption.

9. Further programming notes concerning interruptible instructions are included in “Interruptible Instructions” in Chapter 5, “Program Execution.”

10. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

MOVE LONG EXTENDED

```
MVCLE   R1,R3,D2(B2)      [RS]
```



All or part of the third operand is placed at the first-operand location. The remaining rightmost byte positions, if any, of the first-operand location are filled with padding bytes. The operation proceeds until the end of the first-operand location is reached or a CPU-determined number of bytes have been placed at the first-operand location, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and third operand is designated by the contents of general registers R_1 and R_3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The second-operand address is not used to address data; instead, the rightmost eight bits of the second-operand address, bits 56-63, are the padding byte. Bits 0-55 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-69 on page 7-128.

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified in general register $R_1 + 1$ have been placed at the first-operand location or when a CPU-determined number of bytes have been placed, whichever occurs first. If the third operand is shorter than the first operand, the remaining rightmost bytes of the first-operand location are filled with the padding byte.

When the operation is completed because the end of the first operand has been reached, the condition code is set to 0 if the two operand lengths are equal, it is set to 1 if the first-operand length is less than the third-operand length, or it is set to 2 if the first-operand length is greater than the third-operand length. When the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, condition code 3 is set.

No test is made for destructive overlap, and the results in the first-operand location are unpredictable when destructive overlap exists. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it.

Operands do not overlap destructively if the leftmost byte of the first operand does not coincide with any of the third-operand bytes participating in the operation other than the leftmost byte of the third operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand bytes in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of bytes in locations from 0 up.

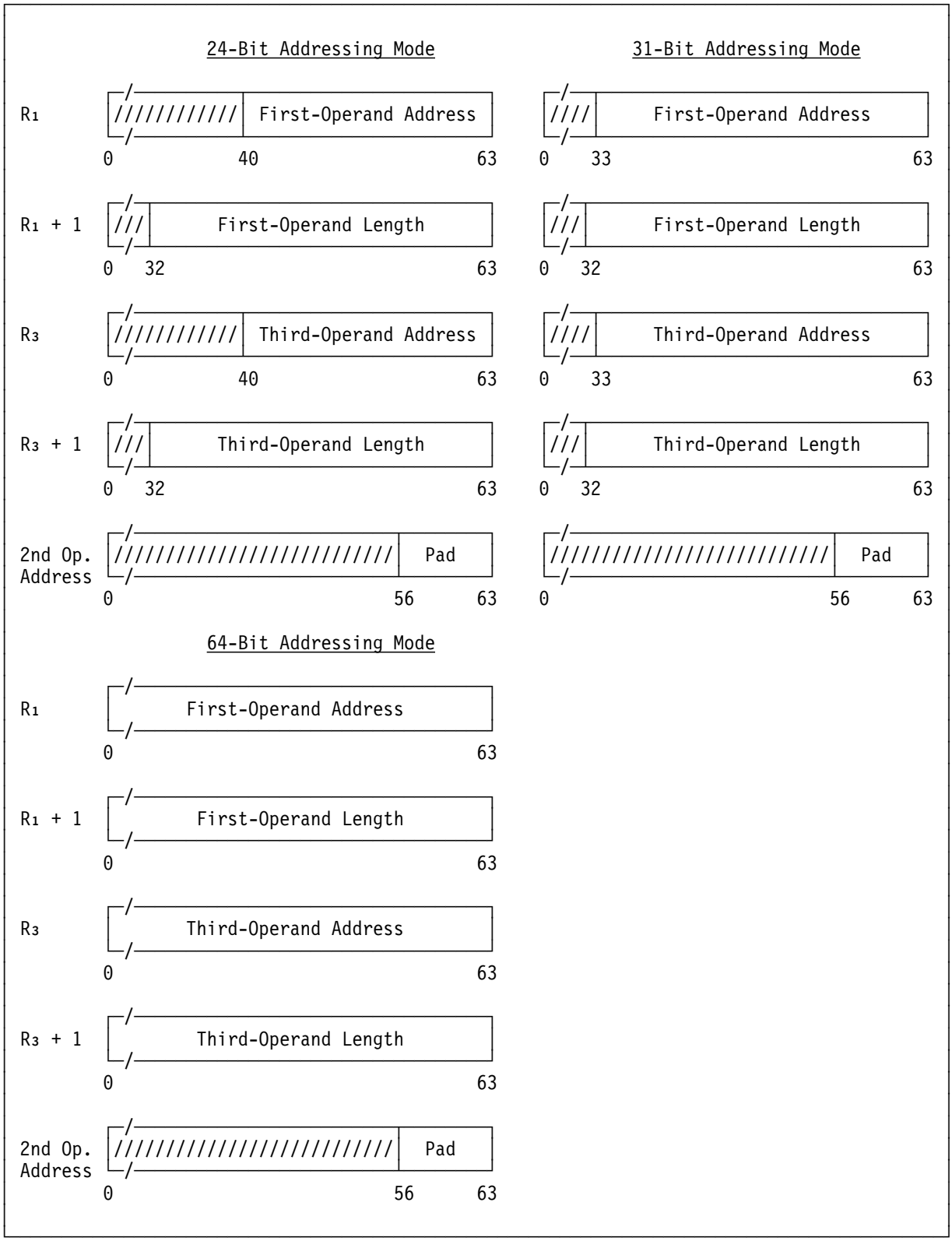


Figure 7-69. Register Contents and Second-Operand Address for MOVE LONG EXTENDED

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; and, in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.

When the length specified in general register $R_1 + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

Padding byte values of B0 hex and B8 hex may be used during the nonpadding part of the operation by some models, in certain cases, as an indication of whether the movement should be performed bypassing the cache or using the cache, respectively. Thus, a padding byte of B0 hex indicates no intention to reference the destination area after the move, and a padding byte of B8 hex indicates an intention to reference the destination area.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG are single-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs, unless the padding byte is B1 hex. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by channel programs, that portion of the first operand which is filled with the padding byte is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. The length in general register $R_3 + 1$ is decremented by the number of bytes moved out of the third-operand location, and the address in general register R_3 is incremented by the same amount.

If the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes moved, and the addresses in general registers R_1 and R_3 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next byte to be moved. If the operation is completed during padding, the length field in general register $R_3 + 1$ is zero, the address in general register R_3 is incremented by the original length in general register $R_3 + 1$, and general registers R_1 and $R_1 + 1$ reflect the extent of the padding operation.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_3, and $R_3 + 1$, always remain unchanged.

The padding byte may be formed from $D_2(B_2)$ multiple times during the execution of the instruction, and the registers designated by R_1 and R_3 may be updated multiple times. Therefore, if B_2 equals R_1, $R_1 + 1$, R_3, or $R_3 + 1$ and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The maximum amount is approximately 4K bytes of either operand.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_3 may be set to zeros or may remain unchanged from their original values, even when one or both of the original length values are zeros.

When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the third operand is longer than the first operand, access exceptions are not recognized for the part of the third-operand field that is in excess of the first-operand field. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R_1 field is odd, PER storage-alteration events are not recognized, and no change bits are set.

Resulting Condition Code:

0 All bytes moved, operand lengths equal
1 All bytes moved, first-operand length low
2 All bytes moved, first-operand length high

3 CPU-determined number of bytes moved without reaching end of first operand

Program Exceptions:

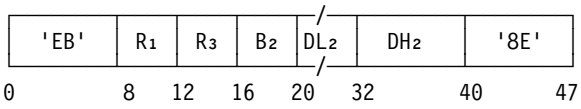
- Access (fetch, operand 3; store, operand 1)
- Specification

Programming Notes:

1. MOVE LONG EXTENDED is intended for use in place of MOVE LONG when the operand lengths are specified as 32-bit or 64-bit binary integers and a test for destructive overlap is not required. MOVE LONG EXTENDED sets condition code 3 in cases in which MOVE LONG would be interrupted.
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the movement. The program need not determine the number of bytes that were moved.
3. The function of not processing more than approximately 4K bytes of either operand is intended to permit software polling of a flag that may be set by a program on another CPU during long operations.
4. MOVE LONG EXTENDED may be used for clearing storage by setting the padding byte to zero and the third-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see "Storage-Operand Consistency" on page 5-87.
5. When the contents of the R_1 and R_3 fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by twice the number of bytes moved. The condition code is finally set to 0 after possible settings to 3.
6. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

MOVE LONG UNICODE

```
MVCLU    R1,R3,D2(B2)      [RSY]
```



All or part of the third operand is placed at the first-operand location. The remaining rightmost two-byte character positions, if any, of the first-operand location are filled with two-byte padding characters. The operation proceeds until the end of the first-operand location is reached or a CPU-determined number of characters have been placed at the first-operand location, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost character of the first operand and third operand is designated by the contents of general registers R_1 and R_3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 0-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The contents of general registers $R_1 + 1$ and $R_3 + 1$ must specify an even number of bytes; otherwise, a specification exception is recognized.

The handling of the addresses in general registers R_1 and R_3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In

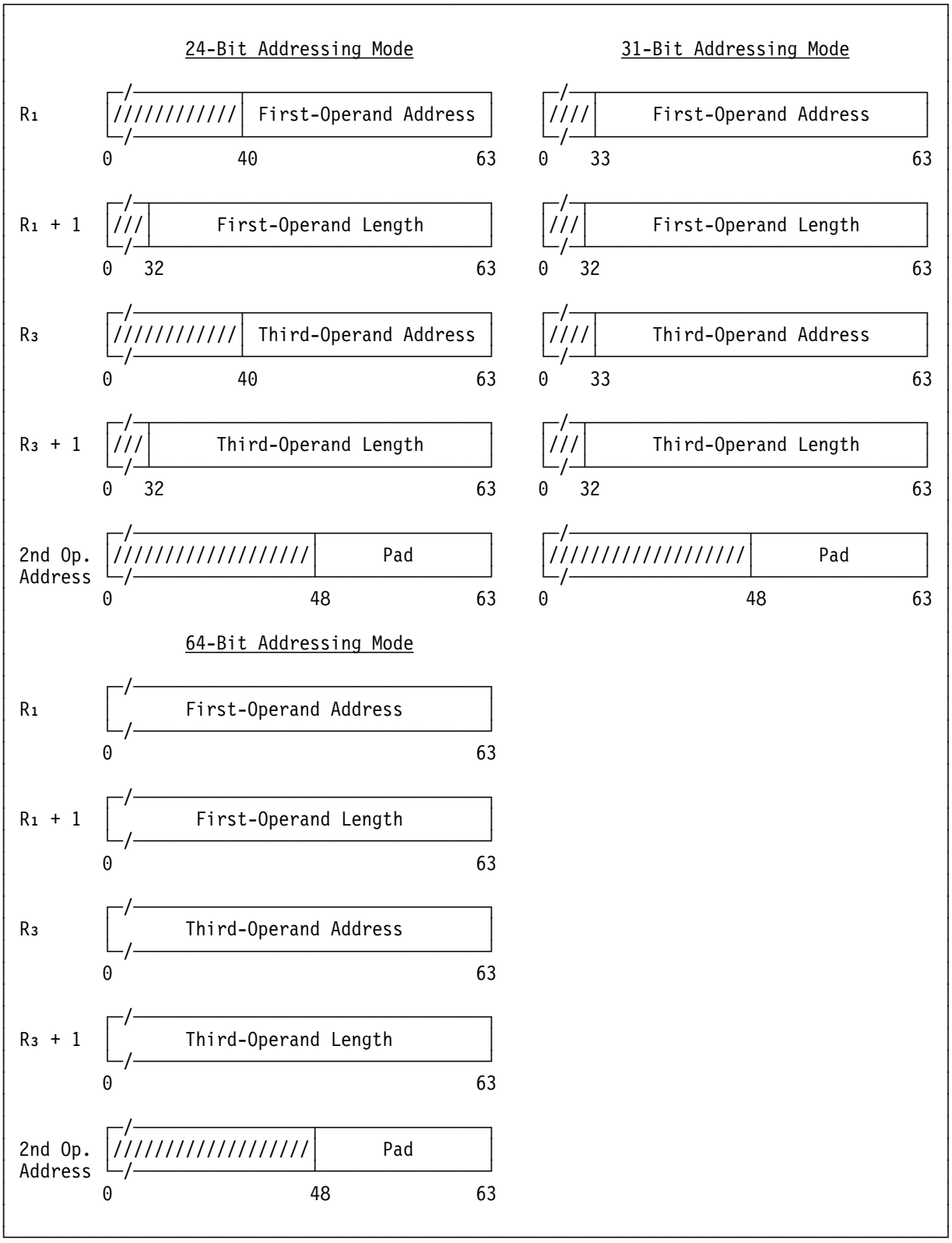


Figure 7-70. Register Contents and Second-Operand Address for MOVE LONG UNICODE

the 64-bit addressing mode, the contents of bit positions 0-63 of the registers constitute the address.

The second-operand address is not used to address data; instead, the rightmost 16 bits of the

second-operand address, bits 48-63, are the two-byte padding character. Bits 0-47 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-70 on page 7-131.

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, character by character. The operation is ended when the number of characters specified by the contents of general register $R_1 + 1$ have been placed at the first-operand location or when a CPU-determined number of characters have been placed, whichever occurs first. If the third operand is shorter than the first operand, the remaining rightmost character positions of the first-operand location are filled with the two-byte padding character.

When the operation is completed because the end of the first operand has been reached, the condition code is set to 0 if the two operand lengths are equal, it is set to 1 if the first-operand length is less than the third-operand length, or it is set to 2 if the first-operand length is greater than the third-operand length. When the operation is completed because a CPU-determined number of characters have been moved without reaching the end of the first operand, condition code 3 is set.

No test is made for destructive overlap, and the results in the first-operand location are unpredictable when destructive overlap exists. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it.

Operands do not overlap destructively if the leftmost character of the first operand does not coincide with any of the third-operand characters participating in the operation other than the leftmost character of the third operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand characters in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of characters in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; and, in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.

When the length specified in general register $R_1 + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG UNICODE are single-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by channel programs, that portion of the first operand which is filled with the two-byte padding character is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by 2 times the number of characters stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. The length in general register $R_3 + 1$ is decremented by 2 times the number of characters moved out of the third-operand location, and the address in general register R_3 is incremented by the same amount.

If the operation is completed because a CPU-determined number of characters have been moved without reaching the end of the first operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by 2 times the number of characters moved, and the addresses in general registers R_1 and R_3 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next character to be moved. If the operation is completed during padding, the length field in general register $R_3 + 1$ is zero, the address in general register R_3 is incremented by 2 times the number of characters moved from operand 3, and general registers R_1 and $R_1 + 1$ reflect the extent of the padding operation.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

The two-byte padding character may be formed from $D_2(B_2)$ multiple times during the execution of the instruction, and the registers designated by R_1 and R_3 may be updated multiple times. Therefore, if B_2 equals R_1, $R_1 + 1$, R_3, or $R_3 + 1$ and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_3 may be set to zeros or may remain unchanged from their original values, including the case when one or both of the original length values are zeros.

When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the third operand is longer than the first operand, access exceptions are not recognized for the part of the third-operand field that is in excess of the first-operand field. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field or length associated with that operand is odd. Also, when the R_1 field or length is odd, PER storage-alteration events are not recognized, and no change bits are set.

Resulting Condition Code:

0 All characters moved, operand lengths equal
1 All characters moved, first-operand length low
2 All characters moved, first-operand length high
3 CPU-determined number of characters moved without reaching end of first operand

Program Exceptions:

- Access (fetch, operand 3; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

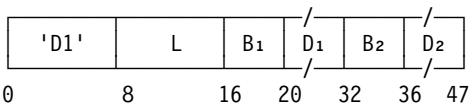
Programming Notes:

1. MOVE LONG UNICODE is intended for use in place of MOVE LONG or MOVE LONG EXTENDED when the padding character is two bytes. The character may be a Unicode character or any other double-byte character. MOVE LONG UNICODE sets condition code 3 in cases in which MOVE LONG would be interrupted.
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the movement. The program need not determine the number of characters that were moved.
3. MOVE LONG UNICODE may be used for filling storage with padding characters by placing the padding character in the second-operand address and setting the third-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-87.
4. When the contents of the R_1 and R_3 fields are the same, the contents of the designated registers are incremented or decremented only by 2 times the number of characters moved, not by 4 times the number of characters moved. The condition code is finally set to 0 after possible settings to 3.
5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.
6. If padding with a Unicode space character is required (or any character whose representation is less than or equal to FFF hex), the character may be represented in the displacement field of the instruction, for example:

```
MVCLU   6,8,X'020'
```

MOVE NUMERICS

MVN D1(L,B1),D2(B2) [SS]



The rightmost four bits of each byte in the second operand are placed in the rightmost bit positions of the corresponding bytes in the first operand. The leftmost four bits of each byte in the first operand remain unchanged.

Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

Condition Code: The code remains unchanged.

Program Exceptions:

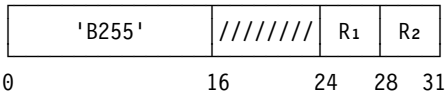
- Access (fetch, operand 2; fetch and store, operand 1)

Programming Notes:

1. An example of the use of the MOVE NUMERICS instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. MOVE NUMERICS moves the numeric portion of a decimal-data field that is in the zoned format. The zoned-decimal format is described in Chapter 8, "Decimal Instructions." The operands are not checked for valid sign and digit codes.
3. Accesses to the first operand of MOVE NUMERICS consist in fetching the rightmost four bits of each byte in the first operand and subsequently storing the updated value of the byte. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

MOVE STRING

MVST R1,R2 [RRE]



All or part of the second operand is placed in the first-operand location. The operation proceeds until the end of the second operand is reached or a CPU-determined number of bytes have been moved, whichever occurs first. The CPU-determined number is at least one. The result is indicated in the condition code.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R_1 and R_2, respectively.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The end of the second operand is indicated by an ending character in the last byte position of the operand. The ending character to be used to determine the end of the second operand is specified in bit positions 56-63 of general register 0. Bit positions 32-55 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right and ends as soon as the second-operand ending character has been moved or a CPU-determined number of second-operand bytes have been moved, whichever occurs first. The CPU-determined number is at least one. When the ending character is in the first byte position of the second operand, only the ending character is moved. When the ending character has been moved, condition code 1 is set. When a CPU-determined number of second-operand bytes not including an ending character

have been moved, condition code 3 is set. Destructive overlap is not recognized. If the second operand is used as a source after it has been used as a destination, the results are unpredictable to the extent that an ending character in the second operand may not be recognized.

When condition code 1 is set, the address of the ending character in the first operand is placed in general register R_1, and the contents of general register R_2 remain unchanged. When condition code 3 is set, the address of the next byte to be processed in the first and second operands is placed in general registers R_1 and R_2, respectively. Whenever an address is placed in a general register, bits 32-39 of the register, in the 24-bit addressing mode, or bit 32, in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the R_1 and R_2 registers always remain unchanged in the 24-bit or 31-bit mode.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the first and second operands are recognized only for that portion of the operand that is necessarily used in the operation.

The storage-operand-consistency rules are the same as for the MOVE (MVC) instruction, except that destructive overlap is not recognized.

Resulting Condition Code:

0 --
1 Entire second operand moved; general register R_1 updated with address of ending character in first operand; general register R_2 unchanged
2 --
3 CPU-determined number of bytes moved; general registers R_1 and R_2 updated with addresses of next bytes

Program Exceptions:

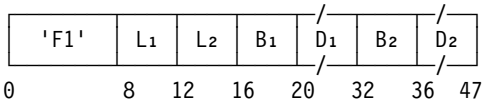
- Access (fetch, operand 2; store, operand 1)
- Specification

Programming Notes:

1. An example of the use of the MOVE STRING instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the data movement. The program need not determine the number of bytes that were moved.
3. R_1 or R_2 may be zero, in which case general register 0 is treated as containing an address and also the ending character.
4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

MOVE WITH OFFSET

MVO $D_1(L_1,B_1),D_2(L_2,B_2)$ [SS]



The second operand is placed to the left of and adjacent to the rightmost four bits of the first operand.

The rightmost four bits of the first operand are attached as the rightmost bits to the second operand, the second-operand bits are offset by four bit positions, and the result is placed at the first-operand location.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first operand is too short to contain all of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if the operands were processed one byte at a time, as if each result byte were stored immediately after fetching the necessary operand bytes, and as if the left digit of each second-operand byte were to remain available for the next result byte and need not be refetched.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; fetch and store, operand 1)

Programming Notes:

1. An example of the use of the MOVE WITH OFFSET instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. MOVE WITH OFFSET may be used to shift packed decimal data by an odd number of digit positions. The packed-decimal format is described in Chapter 8, "Decimal Instructions." The operands are not checked for valid sign and digit codes. In many cases, however, SHIFT AND ROUND DECIMAL may be more convenient to use.
3. Access to the rightmost byte of the first operand of MOVE WITH OFFSET consists in fetching the rightmost four bits and subsequently storing the updated value of this byte. These fetch and store accesses to the rightmost byte of the first operand do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."
4. The storage-operand references for MOVE WITH OFFSET may be multiple-access references. (See "Storage-Operand Consistency" on page 5-87.)

MOVE ZONES

```
MVZ     D1(L,B1),D2(B2)          [SS]
```

The leftmost four bits of each byte in the second operand are placed in the leftmost four bit positions of the corresponding bytes in the first operand. The rightmost four bits of each byte in the first operand remain unchanged.

Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after the necessary operand byte is fetched.

Condition Code: The code remains unchanged.

Program Exceptions:

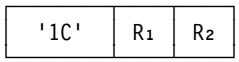
- Access (fetch, operand 2; fetch and store, operand 1)

Programming Notes:

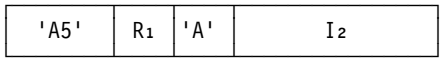
1. An example of the use of the MOVE ZONES instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. MOVE ZONES moves the zoned portion of a decimal field in the zoned format. The zoned format is described in Chapter 8, "Decimal Instructions." The operands are not checked for valid sign and digit codes.
3. Accesses to the first operand of MOVE ZONES consist in fetching the leftmost four bits of each byte in the first operand and subsequently storing the updated value of the byte. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for the OR (OI) instruction in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

MULTIPLY

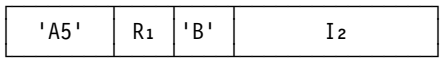
```
MR      R1,R2       [RR]
```



```
OILH     R1,I2     [RI]
```




```
OILL     R1,I2     [RI]
```



The second operand is ORed with bits of the first operand, and the result replaces those bits of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bits of the first operand that are ORed with the second operand and then replaced are as follows:

Instruction	Bits ORed and Replaced
OIHH	0-15
OIHL	16-31
OILH	32-47
OILL	48-63

The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero.

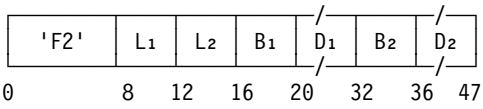
Resulting Condition Code:

0 Sixteen-bit result zero
1 Sixteen-bit result not zero
2 --
3 --

Program Exceptions: None.

PACK

```
PACK   D1(L1,B1),D2(L2,B2)     [SS]
```



The format of the second operand is changed from zoned to packed, and the result is placed at the first-operand location. The zoned and packed formats are described in Chapter 8, “Decimal Instructions.”

The second operand is treated as having the zoned format. The numeric bits of each byte are treated as a digit. The zone bits are ignored, except the zone bits in the rightmost byte, which are treated as a sign.

The sign and digits are moved unchanged to the first operand and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first operand is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the necessary operand bytes. Two second-operand bytes are needed for each result byte, except for the rightmost byte of the result field, which requires only the rightmost second-operand byte.

Condition Code: The code remains unchanged.

Program Exceptions:

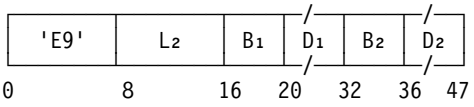
- Access (fetch, operand 2; store, operand 1)

Programming Notes:

1. An example of the use of the PACK instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. PACK may be used to interchange the two hexadecimal digits in one byte by specifying a zero in the L_1 and L_2 fields and the same address for both operands.
3. To remove the zone bits of all bytes of a field, including the rightmost byte, both operands should be extended on the right with a dummy byte, which subsequently should be ignored in the result field.
4. The storage-operand references for PACK may be multiple-access references. (See "Storage-Operand Consistency" on page 5-87.)

PACK ASCII

```
PKA      D1(B1),D2(L2,B2)   [SS]
```



The format of the second operand is changed from ASCII to packed, and the result is placed at the first-operand location. The packed format is described in Chapter 8, "Decimal Instructions."

The second-operand bytes are treated as containing decimal digits, having the binary encoding 0000-1001 for 0-9, in their rightmost four bit positions. The leftmost four bit positions of a byte are ignored. The second operand is considered to be positive.

The implied positive sign (1100 binary) and the source digits are placed at the first-operand location. The source digits are moved unchanged and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros.

The length of the first operand is 16 bytes.

The length of the second operand is designated by the contents of the L_2 field. The second-operand length must not exceed 32 bytes (L_2 must be less than or equal to 31); otherwise, a specification exception is recognized.

When the length of the second operand is 32 bytes, the leftmost byte is ignored.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first-operand location is not necessarily stored into in any particular order.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

Programming Notes:

1. Although PACK ASCII is primarily intended to change the format of ASCII decimal digits, its use is not restricted to ASCII since the leftmost four bits of each byte are ignored.
2. The following example illustrates the use of the instruction to pack ASCII digits:

```
ASDIGITS  DS     0CL31
          DC     X'3132333435'
          DC     X'3637383930'
          DC     X'3132333435'
          DC     X'3637383930'
          DC     X'3132333435'
          DC     X'3637383930'
          DC     X'31'
PKDIGITS  DS     PL16
          ...
          PKA    PKDIGITS,ASDIGITS(31)
```

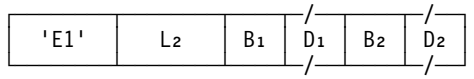
3. The instruction can also be used to pack EBCDIC digits, which is especially useful when the length of the second operand is greater than the 16-byte second-operand limit of PACK.

```
EBDIGITS  DS     0CL31
          DC     X'F1F2F3F4F5'
          DC     X'F6F7F8F9F0'
          DC     X'F1F2F3F4F5'
          DC     X'F6F7F8F9F0'
          DC     X'F1F2F3F4F5'
          DC     X'F6F7F8F9F0'
          DC     X'F1'
PKDIGITS  DS     PL16
          ...
          PKA    PKDIGITS,EBDIGITS(31)
```

4. The storage-operand references for PACK ASCII may be multiple-access references. (See "Storage-Operand Consistency" on page 5-87.)

PACK UNICODE

```
PKU      D1(B1),D2(L2,B2)    [SS]
```



The format of the second operand is changed from Unicode to packed, and the result is placed at the first-operand location. The packed format is described in Chapter 8, "Decimal Instructions."

The two-byte second-operand characters are treated as Unicode Basic Latin characters containing decimal digits, having the binary encoding 0000-1001 for 0-9, in their rightmost four bit positions. The leftmost 12 bit positions of a character are ignored. The second operand is considered to be positive.

The implied positive sign (1100 binary) and the source digits are placed at the first-operand location. The source digits are moved unchanged and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros.

The length of the first operand is 16 bytes.

The byte length of the second operand is designated by the contents of the L_2 field. The second-operand length must not exceed 32 characters or 64 bytes, and the byte length must be even (L_2 must be less than or equal to 63 and must be odd); otherwise, a specification exception is recognized.

When the length of the second operand is 32 characters (64 bytes), the leftmost character is ignored.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first-operand location is not necessarily stored into in any particular order.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

Programming Notes:

1. The following example illustrates the use of PACK UNICODE to pack European numbers:

```
UNDIGITS  DS     0CL62
          DC     X'00310032003300340035'
          DC     X'00360037003800390030'
          DC     X'00310032003300340035'
          DC     X'00360037003800390030'
          DC     X'00310032003300340035'
          DC     X'00360037003800390030'
          DC     X'0031'
PKDIGITS  DS     PL16
          ...
          PKU    PKDIGITS,UNDIGITS(62)
```

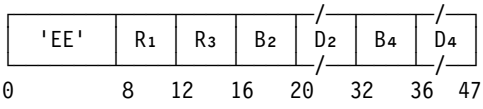
2. Because the leftmost 12 bits of each character are ignored, those Unicode decimal digits where the digit zero has four rightmost zero bits can also be packed by the instruction. For example, for Thai digits:

```
UNDIGITS  DS     0CL62
          DC     X'0E510E520E530E540E55'
          DC     X'0E560E570E580E590E50'
          DC     X'0E510E520E530E540E55'
          DC     X'0E560E570E580E590E50'
          DC     X'0E510E520E530E540E55'
          DC     X'0E560E570E580E590E50'
          DC     X'0E51'
PKDIGITS  DS     PL16
          ...
          PKU    PKDIGITS,UNDIGITS(62)
```

3. The storage-operand references for PACK UNICODE may be multiple-access references. (See "Storage-Operand Consistency" on page 5-87.)

PERFORM LOCKED OPERATION

```
PLO    R1,D2(B2),R3,D4(B4)      [SS]
```



After the lock specified in general register 1 has been obtained, the operation specified by the function code in general register 0 is performed, and then the lock is released. However, as observed by other CPUs: (1) storage operands, including fields in a parameter list that may be used, may be fetched, and may be tested for store-type access exceptions if a store at a tested location is possible, before the lock is obtained, and (2) operands may be stored in the parameter list after the lock has been released. If an operand not in the parameter list is fetched before the lock is obtained, it is fetched again after the lock has been obtained.

The function code can specify any of six operations: compare and load, compare and swap, double compare and swap, compare and swap and store, compare and swap and double store, or compare and swap and triple store.

A test bit in general register 0 specifies, when one, that a lock is not to be obtained and none of the six operations is to be performed but, instead, the validity of the function code is to be tested. This will be useful if additional function codes for additional operations are assigned in the future. This definition is written as if the test bit is zero except when stated otherwise.

If compare and load is specified, the first-operand comparison value and the second operand are compared. If they are equal, the fourth operand is placed in the third-operand location. If the comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value.

If compare and swap is specified, the first-operand comparison value and the second operand are compared. If they are equal, the first-operand replacement value is stored at the second-operand location. If the comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value.

If double compare and swap is specified, the first-operand comparison value and the second operand are compared. If they are equal, the third-operand comparison value and the fourth operand are compared. If both comparisons indicate equality, the first-operand and third-operand replacement values are stored at the second-operand location and fourth-operand location, respectively. If the first comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value. If the first comparison indicates equality but the second does not, the fourth operand is placed in the third-operand-comparison-value location as a new third-operand comparison value.

If compare and swap and store, double store, or triple store is specified, the first-operand comparison value and the second operand are compared. If they are equal, the first-operand replacement value is stored at the second-operand location, and the third operand is stored at the fourth-operand location. Then, if the operation is the double-store or triple-store operation, the fifth operand is stored at the sixth-operand location, and, if it is the triple-store operation, the seventh operand is stored at the eighth-operand location. If the first-operand comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value.

After any of the six operations, the result of the comparison or comparisons is indicated in the condition code.