Chapter 7. General Instructions

This chapter includes all the unprivileged instructions described in this publication other than the decimal and floating-point instructions.

Data Format

The general instructions treat data as being of four types: signed binary integers, unsigned binary integers, unstructured logical data, and decimal data. Data is treated as decimal by the conversion, packing, and unpacking instructions. Decimal data is described in Chapter 8, “Decimal Instructions.”

The general instructions manipulate data which resides in general registers or in storage or is introduced from the instruction stream. Some general instructions operate on data which resides in the PSW or the TOD clock.

In a storage-and-storage operation the operand fields may be defined in such a way that they overlap. The effect of this overlap depends upon the operation. When the operands remain unchanged, as in COMPARE or TRANSLATE AND TEST, overlapping does not affect the execution of the operation. For instructions such as MOVE and TRANSLATE, one operand is replaced by new data, and the execution of the operation may be affected by the amount of overlap and the manner in which data is fetched or stored. For purposes of evaluating the effect of overlapped operands, data is considered to be handled one eight-bit byte at a time. Special rules apply to the operands of MOVE LONG and MOVE INVERSE. See “Interlocks within a Single Instruction” on page 5-81 for how overlap is detected in the access-register mode.

Binary-Integer Representation

Binary integers are treated as signed or unsigned.

In an unsigned binary integer, all bits are used to express the absolute value of the number. When two unsigned binary integers of different lengths are added, the shorter number is considered to be extended on the left with zeros.

In some operations, the result is achieved by the use of the one's complement of the number. The one's complement of a number is obtained by inverting each bit of the number, including the sign.

For signed binary integers, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two's-complement binary notation with a one in the sign-bit position.

Specifically, a negative number is represented by the two's complement of the positive number of the same absolute value. The two's complement of a number is obtained by forming the one's complement of the number, adding a value of one in the rightmost bit position, allowing a carry into the sign position, and ignoring any carry out of the sign position.

This number representation can be considered the rightmost portion of an infinitely long representation of the number. When the number is positive, all bits to the left of the most significant bit of the number are zeros. When the number is negative, these bits are ones. Therefore, when a signed operand must be extended with bits on the left, the extension is achieved by setting these bits equal to the sign bit of the operand.

The notation for signed binary integers does not include a negative zero. It has a number range in which, for a given length, the set of negative nonzero numbers is one larger than the set of positive nonzero numbers. The maximum positive number consists of a sign bit of zero followed by all ones, whereas the maximum negative number (the negative number with the greatest absolute value) consists of a sign bit of one followed by all zeros.

A signed binary integer of either sign, except for zero and the maximum negative number, can be changed to a number of the same magnitude but opposite sign by forming its two's complement. Forming the two's complement of a number is equivalent to subtracting the number from zero. The two's complement of zero is zero.

The two's complement of the maximum negative number cannot be represented in the same number of bits. When an operation, such as LOAD COMPLEMENT, attempts to produce the two's complement of the maximum negative number, the result is the maximum negative number, and a fixed-point-overflow exception is recognized. An overflow does not result, however, when the maximum negative number is complemented as an intermediate result but the final result is within the representable range. An example of this case is a subtraction of the maximum negative number from -1. The product of two maximum negative numbers of a given length is representable as a positive number of double that length.

In discussions of signed binary integers in this publication, a signed binary integer includes the sign bit. Thus, the expression “32-bit signed binary integer” denotes an integer with 31 numeric bits and a sign bit, and the expression “64-bit signed binary integer” denotes an integer with 63 numeric bits and a sign bit.

In an arithmetic operation, a carry out of the numeric field of a signed binary integer is carried into the sign bit. However, in algebraic left-shifting, the sign bit does not change even if significant numeric bits are shifted out.

Programming Notes:

1. An alternate way of forming the two's complement of a signed binary integer is to invert all bits to the left of the rightmost one bit, leaving the rightmost one bit and all zero bits to the right of it unchanged.
2. The numeric bits of a signed binary integer may be considered to represent a positive value, with the sign representing a value of either zero or the maximum negative number.

Binary Arithmetic

Many of the instructions that perform a register-and-storage or register-and-register binary-arithmetic operation are provided in sets of three instructions corresponding to three different combinations of operand lengths. These three instructions have the same name but different operation codes and mnemonics. For example, ADD (A) operates on 32-bit operands and produces a 32-bit result, ADD (AG) operates on 64-bit operands and produces a 64-bit result, and ADD (AGF) operates on a 64-bit operand and a 32-bit operand and produces a 64-bit result. The letter “G” alone in the mnemonic indicates a completely 64-bit operation, and the letters “GF” indicate a 32-to-64-bit operation.

In a 32-to-64-bit operation, the intermediate result is 64 bits. LOAD COMPLEMENT (LCGFR) forms the two's complement of the maximum negative 32-bit number without recognizing overflow.

A 32-bit operand in a general register is in bit positions 32-63 of the register. In an operation on the operand, such as by ADD (A), bits 0-31 of the register are unused and remain unchanged. A 64-bit operand in a general register is in bit positions 0-63 of the register, and all of the bits participate in an operation on the operand, such as by ADD (AG). However, some instructions, which do not have “G” in their mnemonics, use a 64-bit operand of which the leftmost 32 bits are in bit positions 32-63 of the even register of an even-odd general-register pair, and the rightmost 32 bits are in bit positions 32-63 of the odd register of the pair.

The bits of a 32-bit operand in storage are numbered 0-31. When the operand is in bit positions 32-63 of a general register, the bits are numbered 32-63.

Signed Binary Arithmetic

Addition and Subtraction

Addition of signed binary integers is performed by adding all bits of each operand, including the sign bits. When one of the operands is shorter, the shorter operand is considered to be extended on the left to the length of the longer operand by propagating the sign-bit value.

For a 32-bit signed binary integer in a general register, the sign bit is bit 32 of the register. For a 64-bit signed binary integer in a general register, the sign bit is bit 0 of the register.

Subtraction is performed by adding the one's complement of the second operand and a value of one to the first operand.

Fixed-Point Overflow

A fixed-point-overflow condition exists for signed binary addition or subtraction when the carry out of the sign-bit position and the carry out of the leftmost numeric bit position disagree. Detection of an overflow does not affect the result produced by the addition. In mathematical terms, signed addition and subtraction produce a fixed-point overflow when the result is outside the range of representation for signed binary integers. Specifically, for ADD (A) and SUBTRACT (S), which operate on 32-bit signed binary integers, there is an overflow when the proper result would be greater than or equal to $+2^{31}$ or less than -2^{31}. The actual result placed in the general register after an overflow differs from the proper result by 2^{32}. A fixed-point overflow causes a program interruption if allowed by the program mask. Similarly, for ADD (AG) and SUBTRACT (SG), which operate on 64-bit signed binary integers, there is an overflow when the proper result would be greater than or equal to $+2^{63}$ or less than -2^{63}, and the actual result placed in the general register after an overflow differs from the proper result by 2^{64}. ADD (AGF) and SUBTRACT (SGF) have the same 64-bit result and overflow rules as ADD (AG) and SUBTRACT (SG).

The instructions SHIFT LEFT SINGLE and SHIFT LEFT DOUBLE produce an overflow when the result is outside the range of representation for signed binary integers. The actual result differs from that for addition and subtraction in that the sign of the result remains the same as the original sign.

Unsigned Binary Arithmetic

Addition of unsigned binary integers is performed by adding all bits of each operand. Subtraction is performed by adding the one's complement of the second operand (the subtractor) and a value of one to the first operand (the subtrahend). In any case, when one of the operands is shorter, the shorter operand is considered to be extended on the left with zeros. During subtraction, this extension applies before an operand is complemented, and it applies to the value of one.

Unsigned binary arithmetic is used in address arithmetic for adding the X, B, and D fields. (See “Address Generation” on page 5-7.) It is also used to obtain the addresses of the function bytes in TRANSLATE and TRANSLATE AND TEST. Furthermore, unsigned binary arithmetic is used on 32-bit or 64-bit unsigned binary integers by ADD LOGICAL, ADD LOGICAL WITH CARRY, DIVIDE LOGICAL, MULTIPLY LOGICAL, SUBTRACT LOGICAL, and SUBTRACT LOGICAL WITH BORROW.

Given the same length operands, ADD (A, AG, AGF) and ADD LOGICAL (AL, ALG, ALGF) produce the same 32-bit or 64-bit result. The instructions differ only in the interpretation of this result. ADD interprets the result as a signed binary integer and inspects it for sign, magnitude, and overflow to set the condition code accordingly. ADD LOGICAL interprets the result as an unsigned binary integer and sets the condition code according to whether the result is zero and whether there was a carry out of bit position 32, for a 32-bit integer, or out of bit position 0 for a 64-bit integer. Such a carry is not considered an overflow, and no program interruption for overflow can occur for ADD LOGICAL.

SUBTRACT LOGICAL differs from ADD LOGICAL in that the one's complement of the second operand and a value of one are added to the first operand.

For ADD LOGICAL WITH CARRY, a carry from a previous operation 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. For SUBTRACT LOGICAL WITH BORROW, a borrow from a previous operation is represented by a zero value of bit 18. A borrow is equivalent to the absence of a carry.

Programming Notes:

1. Logical addition and subtraction may be used to perform arithmetic on multiple-precision binary-integer operands. Thus, for multiple-precision addition, ADD LOGICAL can be used to add the lowest-order corresponding parts of the operands, and ADD LOGICAL WITH CARRY can be used to add the other corresponding parts of the operands, moving from right to left in the operands. If the multiple-precision operands are signed, ADD should be used on the highest-order parts. The condition code then indicates any overflow or the proper sign and magnitude of the entire result; an overflow is also indicated by a program interruption for fixed-point overflow if allowed by the program mask. When ADD is used, a value of one must be added to the sum of the highest-order parts if the condition code indicated there was a carry from the addition of the next-lower parts.
2. Another use for ADD LOGICAL is to increment values representing binary counters, which are allowed to wrap around from all ones to all zeros without indicating overflow.

Signed and Logical Comparison

Comparison operations determine whether two operands are equal or not and, for most operations, which of two unequal operands is the greater (high). Signed-binary-comparison operations are provided which treat the operands as signed binary integers, and logical-comparison operations are provided which treat the operands as unsigned binary integers or as unstructured data.

COMPARE (C, CG, CGF) and COMPARE HALFWORD are signed-binary-comparison operations. These instructions are equivalent to SUBTRACT (S, SG, SGF) and SUBTRACT HALFWORD without replacing either operand, the resulting difference being used only to set the condition code. The operations permit comparison of numbers of opposite sign which differ by 2^{63} or more. Thus, unlike SUBTRACT, COMPARE cannot cause overflow.

Logical comparison of two operands is performed byte by byte, in a left-to-right sequence. The operands are equal when all their bytes are equal. When the operands are unequal, the comparison result is determined by a left-to-right comparison of corresponding bit positions in the first unequal pair of bytes: the zero bit in the first unequal pair of bits indicates the low operand, and the one bit the high operand. Since the remaining bit and byte positions do not change the comparison, it is not necessary to continue comparing unequal operands beyond the first unequal bit pair.

Instructions

The general instructions and their mnemonics, formats, and operation codes are listed in Figure 7-1 on page 7-9. The figure also indicates which instructions are new in z/Architecture as compared to ESA/390, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

The instructions that are new in z/Architecture are indicated in Figure 7-1 by “N.” A few of the instructions that are new in z/Architecture have also been added to ESA/390, and these are indicated by “N3.”

When the operands of an instruction are 32-bit operands, the mnemonic for the instruction does not include a letter indicating the operand length. If there is an instruction with the same name but with 64-bit operands, its mnemonic includes the letter “G.” If there is an instruction with the same name but with a 64-bit first operand and a 32-bit second operand, its mnemonic includes the letters “GF.” In Figure 7-1, when there is an instruction with 32-bit operands and other instructions with the same name but with “G” or “GF” added in their mnemonics, the first instruction has “(32)” after its name, and the other instructions have “(64)” or “(64<32),” respectively, after their names. Some 32-bit operand-length instructions do not have 64-bit operand-length counterparts, and they do not have “(32)” after their names. However, all instructions for multiplication or division are marked to show, or approximately show, operand lengths.

A detailed definition of instruction formats, operand designation and length, and address generation is contained in “Instructions” on page 5-2. Exceptions to the general rules stated in that section are explicitly identified in the individual instruction descriptions.

Note: In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designations for the assembler language are shown with each instruction. For LOAD AND TEST with 32-bit operands, for example, LTR is the mnemonic and R_1,R_2 the operand designation.

Programming Notes:

1. Trimodal addressing affects the general instructions only in the manner in which logical storage addresses are handled, except as follows.

   The instructions BRANCH AND LINK (BAL, BALR), BRANCH AND SAVE (BAS, BASR), BRANCH AND SAVE AND SET MODE, BRANCH AND SET MODE, BRANCH RELATIVE AND SAVE, and BRANCH RELATIVE AND SAVE LONG place information in bit positions 32-39 of general register R_1 as in ESA/390 in the 24-bit or 31-bit addressing mode or place address bits in those bit positions in the 64-bit addressing mode.

   The instruction BRANCH AND SAVE AND SET MODE places a zero in bit position 63 of general register R_1 in the 24-bit or 31-bit addressing mode or places a one in that bit position in the 64-bit addressing mode.

   The instruction BRANCH AND SET MODE leaves the contents of bit position 63 of general register R_1 unchanged in the 24-bit or 31-bit addressing mode or places a one in that bit position in the 64-bit addressing mode.

   The following instructions leave bits 0-31 of a general register unchanged in the 24-bit or 31-bit addressing mode but place or update address or length information in them in the 64-bit addressing mode. Also, the leftmost byte of the results in registers may be handled differently depending on whether the addressing mode is the 24-bit or the 31-bit mode.

   - BRANCH AND LINK (BAL, BALR)
   - BRANCH AND SAVE (BAS, BASR)
   - BRANCH AND SAVE AND SET MODE
   - BRANCH RELATIVE AND SAVE
   - BRANCH RELATIVE AND SAVE LONG
   - CHECKSUM
   - COMPARE AND FORM CODEWORD
   - COMPARE LOGICAL LONG
   - COMPARE LOGICAL LONG EXTENDED
   - COMPARE LOGICAL LONG UNICODE
   - COMPARE LOGICAL STRING
   - COMPARE UNTIL SUBSTRING EQUAL
   - COMPRESSION CALL
   - CONVERT UNICODE TO UTF-8

- CONVERT UTF-8 TO UNICODE
- LOAD ADDRESS
- LOAD ADDRESS EXTENDED
- LOAD ADDRESS RELATIVE LONG
- MOVE LONG
- MOVE LONG EXTENDED
- MOVE LONG UNICODE
- MOVE STRING
- SEARCH STRING
- TRANSLATE EXTENDED
- TRANSLATE AND TEST
- TRANSLATE ONE TO ONE
- TRANSLATE ONE TO TWO
- TRANSLATE TWO TO ONE
- TRANSLATE TWO TO TWO
- UPDATE TREE

The instructions in the preceding list are sometimes called modal instructions.

2. Bits 0-31 of general registers are changed by two types of instructions. The first type is a modal instruction, as listed in the preceding note, when the instruction is executed in the 64-bit addressing mode. The second type is an instruction having, independent of the addressing mode, either a 64-bit result operand in a single general register or a 128-bit result operand in an even-odd general-register pair.

   Most of the instructions of the second type are indicated by a “G,” either alone or in “GF,” in their mnemonics. The other instructions that change or may change bits 0-31 of a general register regardless of the current addressing mode are:

   - AND IMMEDIATE (NIHH, NIHL only)
   - INSERT CHARACTERS UNDER MASK (ICMH only)
   - INSERT IMMEDIATE (IIHH, IIHL only)
   - LOAD LOGICAL IMMEDIATE (LLIHH, LLIHL only)
   - LOAD MULTIPLE DISJOINT
   - LOAD MULTIPLE HIGH
   - LOAD PAIR FROM QUADWORD
   - OR IMMEDIATE (OIHH, OIHL only)

   All of the instructions of the second type are sometimes referred to as “G-type” instructions.

   If a program is not executed in the 64-bit addressing mode and does not contain a G-type instruction, it cannot change bits 0-31 of any general register.

3. It is not intended or expected that old programs not containing G-type instructions will be able to be executed successfully in the 64-bit addressing mode. However, this may be possible, particularly if, by programming convention, bits 0-31 of the general registers are always all zeros when an old program is given control.

4. The following additional general instructions are available when the extended-translation facility 2 is installed:

   - COMPARE LOGICAL LONG UNICODE
   - MOVE LONG UNICODE
   - PACK ASCII
   - PACK UNICODE
   - TRANSLATE ONE TO ONE
   - TRANSLATE ONE TO TWO
   - TRANSLATE TWO TO ONE
   - TRANSLATE TWO TO TWO
   - UNPACK ASCII
   - UNPACK UNICODE

5. The long-displacement facilty uses new instruction formats, named RSY, RXY, and SIY, to provide 20-bit signed displacements. In connection with the long-displacement facility, all previously existing general instructions of the RSE or RXE format are changed to be of format RSY or RXY, respectively, where the new formats differ from the old by using a previously unused byte, now named DH, in the instructions. When the long-displacement facility is installed, the displacement for an instruction operand address is formed by appending DH on the left of the previous displacement field, now named DL, of the instruction. When the long-displacement facility is not installed, eight zero bits are appended on the left of DL, and DH is ignored.

   The following additional general instruction is available when the long-displacement facility is installed.

   - LOAD BYTE

   The following additional RSY-format versions of general instructions are available when the long-displacement facility is installed.

   - COMPARE AND SWAP
   - COMPARE DOUBLE AND SWAP
   - COMPARE LOGICAL CHARACTERS UNDER MASK

- LOAD ACCESS MULTIPLE
- LOAD MULTIPLE
- STORE ACCESS MULTIPLE
- STORE CHARACTERS UNDER MASK
- STORE MULTIPLE

The following additional RXY-format versions of general instructions are available when the long-displacement facility is installed.

- ADD
- ADD HALFWORD
- ADD LOGICAL
- AND
- COMPARE
- COMPARE HALFWORD
- COMPARE LOGICAL
- CONVERT TO BINARY
- CONVERT TO DECIMAL
- EXCLUSIVE OR
- INSERT CHARACTER
- INSERT CHARACTER UNDER MASK
- LOAD
- LOAD ADDRESS
- LOAD HALFWORD
- MULTIPLY SINGLE
- OR
- STORE
- STORE CHARACTER
- STORE HALFWORD
- SUBTRACT
- SUBTRACT HALFWORD
- SUBTRACT LOGICAL

The following additional SIY-format versions of general instructions are available when the long-displacement facility is installed.

- AND
- COMPARE LOGICAL
- EXCLUSIVE OR
- MOVE
- OR
- TEST UNDER MASK

6. The following additional general instructions are available when the message-security assist is installed:

   - CIPHER MESSAGE
   - CIPHER MESSAGE WITH CHAINING
   - COMPUTE INTERMEDIATE MESSAGE DIGEST
   - COMPUTE LAST MESSAGE DIGEST
   - COMPUTE MESSAGE AUTHENTICATION CODE

Name	Mne-monic	Characteristics							Op Code
ADD (32)	AR	RR	C			IF			1A
ADD (64)	AGR	RRE	C	N		IF			B908
ADD (64<32)	AGFR	RRE	C	N		IF			B918
ADD (32)	A	RX	C		A	IF		B_2	5A
ADD (32)	AY	RXY	C	LD	A	IF		B_2	E35A
ADD (64)	AG	RXY	C	N	A	IF		B_2	E308
ADD (64<32)	AGF	RXY	C	N	A	IF		B_2	E318
ADD HALFWORD	AH	RX	C		A	IF		B_2	4A
ADD HALFWORD	AHY	RXY	C	LD	A	IF		B_2	E37A
ADD HALFWORD IMMEDIATE (32)	AHI	RI	C			IF			A7A
ADD HALFWORD IMMEDIATE (64)	AGHI	RI	C	N		IF			A7B
ADD LOGICAL (32)	ALR	RR	C						1E
ADD LOGICAL (64)	ALGR	RRE	C	N					B90A
ADD LOGICAL (64<32)	ALGFR	RRE	C	N					B91A
ADD LOGICAL (32)	AL	RX	C		A			B_2	5E
ADD LOGICAL (32)	ALY	RXY	C	LD	A			B_2	E35E
ADD LOGICAL (64)	ALG	RXY	C	N	A			B_2	E30A
ADD LOGICAL (64<32)	ALGF	RXY	C	N	A			B_2	E31A
ADD LOGICAL WITH CARRY (32)	ALCR	RRE	C	N3					B998
ADD LOGICAL WITH CARRY (64)	ALCGR	RRE	C	N					B988
ADD LOGICAL WITH CARRY (32)	ALC	RXY	C	N3	A			B_2	E398
ADD LOGICAL WITH CARRY (64)	ALCG	RXY	C	N	A			B_2	E388
AND (32)	NR	RR	C						14
AND (64)	NGR	RRE	C	N					B980
AND (32)	N	RX	C		A			B_2	54
AND (32)	NY	RXY	C	LD	A			B_2	E354
AND (64)	NG	RXY	C	N	A			B_2	E380
AND (character)	NC	SS	C		A		ST	B_1 B_2	D4
AND (immediate)	NI	SI	C		A		ST	B_1	94
AND (immediate)	NIY	SIY	C	LD	A		ST	B_1	EB54
AND IMMEDIATE (high high)	NIHH	RI	C	N					A54
AND IMMEDIATE (high low)	NIHL	RI	C	N					A55
AND IMMEDIATE (low high)	NILH	RI	C	N					A56
AND IMMEDIATE (low low)	NILL	RI	C	N					A57
BRANCH AND LINK	BALR	RR				T	B		05
BRANCH AND LINK	BAL	RX					B		45
BRANCH AND SAVE	BASR	RR				T	B		0D
BRANCH AND SAVE	BAS	RX					B		4D
BRANCH AND SAVE AND SET MODE	BASSM	RR				T	B		0C
BRANCH AND SET MODE	BSM	RR				T	B		0B
BRANCH ON CONDITION	BCR	RR				¢[1]	B		07
BRANCH ON CONDITION	BC	RX					B		47
BRANCH ON COUNT (32)	BCTR	RR					B		06
BRANCH ON COUNT (64)	BCTGR	RRE		N			B		B946
BRANCH ON COUNT (32)	BCT	RX					B		46

Figure 7-1 (Part 1 of 9). Summary of General Instructions

	Name	Mne- monic	Characteristics					Op Code
	BRANCH ON COUNT (64)	BCTG	RXY N			B		E346
	BRANCH ON INDEX HIGH (32)	BXH	RS			B		86
	BRANCH ON INDEX HIGH (64)	BXHG	RSY N			B		EB44
	BRANCH ON INDEX LOW OR EQUAL (32)	BXLE	RS			B		87
	BRANCH ON INDEX LOW OR EQUAL (64)	BXLEG	RSY N			B		EB45
	BRANCH RELATIVE AND SAVE	BRAS	RI			B		A75
	BRANCH RELATIVE AND SAVE LONG	BRASL	RIL N3			B		C05
	BRANCH RELATIVE ON CONDITION	BRC	RI			B		A74
	BRANCH RELATIVE ON CONDITION LONG	BRCL	RIL N3			B		C04
	BRANCH RELATIVE ON COUNT (32)	BRCT	RI			B		A76
	BRANCH RELATIVE ON COUNT (64)	BRCTG	RI N			B		A77
	BRANCH RELATIVE ON INDEX HIGH (32)	BRXH	RSI			B		84
	BRANCH RELATIVE ON INDEX HIGH (64)	BRXHG	RIE N			B		EC44
	BRANCH RELATIVE ON INDEX L OR E (32)	BRXLE	RSI			B		85
	BRANCH RELATIVE ON INDEX L OR E (64)	BRXLG	RIE N			B		EC45
	CHECKSUM	CKSM	RRE C	A SP			R_2	B241
|	CIPHER MESSAGE	KM	RRE C MS	A SP	GM I1	ST	R_1 R_2	B92E
|	CIPHER MESSAGE WITH CHAINING	KMC	RRE C MS	A SP	GM I1	ST	R_1 R_2	B92F
	COMPARE (32)	CR	RR C					19
	COMPARE (64)	CGR	RRE C N					B920
	COMPARE (64<32)	CGFR	RRE C N					B930
	COMPARE (32)	C	RX C	A			B_2	59
|	COMPARE (32)	CY	RXY C LD	A			B_2	E359
	COMPARE (64)	CG	RXY C N	A			B_2	E320
	COMPARE (64<32)	CGF	RXY C N	A			B_2	E330
	COMPARE AND FORM CODEWORD	CFC	S C	A SP	II GM		I1	B21A
	COMPARE AND SWAP (32)	CS	RS C	A SP	$	ST	B_2	BA
|	COMPARE AND SWAP (32)	CSY	RSY C LD	A SP	$	ST	B_2	EB14
	COMPARE AND SWAP (64)	CSG	RSY C N	A SP	$	ST	B_2	EB30
	COMPARE DOUBLE AND SWAP (32)	CDS	RS C	A SP	$	ST	B_2	BB
|	COMPARE DOUBLE AND SWAP (32)	CDSY	RSY C LD	A SP	$	ST	B_2	EB31
	COMPARE DOUBLE AND SWAP (64)	CDSG	RSY C N	A SP	$	ST	B_2	EB3E
	COMPARE HALFWORD	CH	RX C	A			B_2	49
|	COMPARE HALFWORD	CHY	RXY C LD	A			B_2	E379
	COMPARE HALFWORD IMMEDIATE (32)	CHI	RI C					A7E
	COMPARE HALFWORD IMMEDIATE (64)	CGHI	RI C N					A7F
	COMPARE LOGICAL (32)	CLR	RR C					15
|	COMPARE LOGICAL (32)	CLY	RXY C LD	A			B_2	E355
	COMPARE LOGICAL (64)	CLGR	RRE C N					B921
	COMPARE LOGICAL (64<32)	CLGFR	RRE C N					B931
	COMPARE LOGICAL (32)	CL	RX C	A			B_2	55
	COMPARE LOGICAL (64)	CLG	RXY C N	A			B_2	E321
	COMPARE LOGICAL (64<32)	CLGF	RXY C N	A			B_2	E331
	COMPARE LOGICAL (character)	CLC	SS C	A			B_1 B_2	D5
	COMPARE LOGICAL (immediate)	CLI	SI C	A			B_1	95

Figure 7-1 (Part 2 of 9). Summary of General Instructions

	Name	Mne-monic	Characteristics													Op Code
|	COMPARE LOGICAL (immediate)	CLIY	SIY	C	LD	A								B_1		EB55
	COMPARE LOGICAL C. UNDER MASK (high)	CLMH	RSY	C	N	A									B_2	EB20
	COMPARE LOGICAL C. UNDER MASK (low)	CLM	RS	C		A									B_2	BD
|	COMPARE LOGICAL C. UNDER MASK (low)	CLMY	RSY	C	LD	A									B_2	EB21
	COMPARE LOGICAL LONG	CLCL	RR	C		A	SP	II						R_1	R_2	0F
	COMPARE LOGICAL LONG EXTENDED	CLCLE	RS	C		A	SP							R_1	R_3	A9
	COMPARE LOGICAL LONG UNICODE	CLCLU	RSY	C	E2	A	SP							R_1	R_2	EB8F
	COMPARE LOGICAL STRING	CLST	RRE	C		A	SP				G0			R_1	R_2	B25D
	COMPARE UNTIL SUBSTRING EQUAL	CUSE	RRE	C		A	SP	II			GM			R_1	R_2	B257
	COMPRESSION CALL	CMPSC	RRE	C		A	SP	II	Dd		GM		ST	R_1	R_2	B263
|	COMPUTE INTERMEDIATE MESSAGE DIGEST	KIMD	RRE	C	MS	A	SP				GM	I1	ST		R_2	B93E
|	COMPUTE LAST MESSAGE DIGEST	KLMD	RRE	C	MS	A	SP				GM	I1	ST		R_2	B93F
|	COMPUTE MESSAGE AUTHENTICATION CODE	KMAC	RRE	C	MS	A	SP				GM	I1	ST		R_2	B91E
	CONVERT TO BINARY (32)	CVB	RX			A			Dd	IK					B_2	4F
|	CONVERT TO BINARY (32)	CVBY	RXY		LD	A			Dd	IK					B_2	E306
	CONVERT TO BINARY (64)	CVBG	RXY		N	A			Dd	IK					B_2	E30E
	CONVERT TO DECIMAL (32)	CVD	RX			A							ST		B_2	4E
|	CONVERT TO DECIMAL (32)	CVDY	RXY		LD	A							ST		B_2	E326
	CONVERT TO DECIMAL (64)	CVDG	RXY		N	A							ST		B_2	E32E
	CONVERT UNICODE TO UTF-8	CUUTF	RRE	C		A	SP						ST	R_1	R_2	B2A6
	CONVERT UTF-8 TO UNICODE	CUTFU	RRE	C		A	SP						ST	R_1	R_2	B2A7
	COPY ACCESS	CPYA	RRE											U_1	U_2	B24D
	DIVIDE (32<64)	DR	RR				SP			IK						1D
	DIVIDE (32<64)	D	RX			A	SP			IK					B_2	5D
	DIVIDE LOGICAL (32<64)	DLR	RRE		N3		SP			IK						B997
	DIVIDE LOGICAL (64<128)	DLGR	RRE		N		SP			IK						B987
	DIVIDE LOGICAL (32<64)	DL	RXY		N3	A	SP			IK					B_2	E397
	DIVIDE LOGICAL (64<128)	DLG	RXY		N	A	SP			IK					B_2	E387
	DIVIDE SINGLE (64)	DSGR	RRE		N		SP			IK						B90D
	DIVIDE SINGLE (64<32)	DSGFR	RRE		N		SP			IK						B91D
	DIVIDE SINGLE (64)	DSG	RXY		N	A	SP			IK					B_2	E30D
	DIVIDE SINGLE (64<32)	DSGF	RXY		N	A	SP			IK					B_2	E31D
	EXCLUSIVE OR (32)	XR	RR	C												17
	EXCLUSIVE OR (64)	XGR	RRE	C	N											B982
	EXCLUSIVE OR (32)	X	RX	C		A									B_2	57
|	EXCLUSIVE OR (32)	XY	RXY	C	LD	A									B_2	E357
	EXCLUSIVE OR (64)	XG	RXY	C	N	A									B_2	E382
	EXCLUSIVE OR (character)	XC	SS	C		A							ST	B_1	B_2	D7
	EXCLUSIVE OR (immediate)	XI	SI	C		A							ST	B_1		97
|	EXCLUSIVE OR (immediate)	XIY	SIY	C	LD	A							ST	B_1		EB57
	EXECUTE	EX	RX			AI	SP				EX					44
	EXTRACT ACCESS	EAR	RRE												U_2	B24F
	EXTRACT PSW	EPSW	RRE		N3											B98D
	INSERT CHARACTER	IC	RX			A									B_2	43
|	INSERT CHARACTER	ICY	RXY		LD	A									B_2	E373

Figure 7-1 (Part 3 of 9). Summary of General Instructions

Name	Mne- monic	Characteristics					Op Code
INSERT CHARACTERS UNDER MASK (high)	ICMH	RSY C N	A			B_2	EB80
INSERT CHARACTERS UNDER MASK (low)	ICM	RS C	A			B_2	BF
INSERT CHARACTERS UNDER MASK (low)	ICMY	RSY C LD	A			B_2	EB81
INSERT IMMEDIATE (high high)	IIHH	RI N					A50
INSERT IMMEDIATE (high low)	IIHL	RI N					A51
INSERT IMMEDIATE (low high)	IILH	RI N					A52
INSERT IMMEDIATE (low low)	IILL	RI N					A53
INSERT PROGRAM MASK	IPM	RRE					B222
LOAD (32)	LR	RR					18
LOAD (64)	LGR	RRE N					B904
LOAD (64<32)	LGFR	RRE N					B914
LOAD (32)	L	RX	A			B_2	58
LOAD (32)	LY	RXY LD	A			B_2	E358
LOAD (64)	LG	RXY N	A			B_2	E304
LOAD (64<32)	LGF	RXY N	A			B_2	E314
LOAD ACCESS MULTIPLE	LAM	RS	A SP			UB	9A
LOAD ACCESS MULTIPLE	LAMY	RSY LD	A SP			UB	EB9A
LOAD ADDRESS	LA	RX					41
LOAD ADDRESS	LAY	RXY LD					E371
LOAD ADDRESS EXTENDED	LAE	RX				U_1 BP	51
LOAD ADDRESS RELATIVE LONG	LARL	RIL N3					C00
LOAD AND TEST (32)	LTR	RR C					12
LOAD AND TEST (64)	LTGR	RRE C N					B902
LOAD AND TEST (64<32)	LTGFR	RRE C N					B912
LOAD BYTE (32)	LB	RXY LD	A				E376
LOAD BYTE (64)	LGB	RXY LD	A				E377
LOAD COMPLEMENT (32)	LCR	RR C		IF			13
LOAD COMPLEMENT (64)	LCGR	RRE C N		IF			B903
LOAD COMPLEMENT (64<32)	LCGFR	RRE C N		IF			B913
LOAD HALFWORD (32)	LH	RX	A			B_2	48
LOAD HALFWORD (32)	LHY	RXY LD	A			B_2	E378
LOAD HALFWORD (64)	LGH	RXY N	A			B_2	E315
LOAD HALFWORD IMMEDIATE (32)	LHI	RI					A78
LOAD HALFWORD IMMEDIATE (64)	LGHI	RI N					A79
LOAD LOGICAL (64<32)	LLGFR	RRE N					B916
LOAD LOGICAL (64<32)	LLGF	RXY N	A			B_2	E316
LOAD LOGICAL CHARACTER	LLGC	RXY N	A			B_2	E390
LOAD LOGICAL HALFWORD	LLGH	RXY N	A			B_2	E391
LOAD LOGICAL IMMEDIATE (high high)	LLIHH	RI N					A5C
LOAD LOGICAL IMMEDIATE (high low)	LLIHL	RI N					A5D
LOAD LOGICAL IMMEDIATE (low high)	LLILH	RI N					A5E
LOAD LOGICAL IMMEDIATE (low low)	LLILL	RI N					A5F
LOAD LOGICAL THIRTY ONE BITS	LLGTR	RRE N					B917
LOAD LOGICAL THIRTY ONE BITS	LLGT	RXY N	A			B_2	E317
LOAD LOGICAL IMMEDIATE (low high)	LLILH	RI N					A5E

Figure 7-1 (Part 4 of 9). Summary of General Instructions

Name	Mne- monic	Characteristics					Op Code
LOAD LOGICAL IMMEDIATE (low low)	LLILL	RI N					A5F
LOAD LOGICAL THIRTY ONE BITS	LLGTR	RRE N					B917
LOAD LOGICAL THIRTY ONE BITS	LLGT	RXY N	A			B_2	E317
LOAD MULTIPLE (32)	LM	RS	A			B_2	98
LOAD MULTIPLE (32)	LMY	RSY LD	A			B_2	EB98
LOAD MULTIPLE (64)	LMG	RSY N	A			B_2	EB04
LOAD MULTIPLE DISJOINT	LMD	SS N	A			B_2 B_4	EF
LOAD MULTIPLE HIGH	LMH	RSY N	A			B_2	EB96
LOAD NEGATIVE (32)	LNR	RR C					11
LOAD NEGATIVE (64)	LNGR	RRE C N					B901
LOAD NEGATIVE (64<32)	LNGFR	RRE C N					B911
LOAD PAIR FROM QUADWORD	LPQ	RXY N	A SP			B_2	E38F
LOAD POSITIVE (32)	LPR	RR C		IF			10
LOAD POSITIVE (64)	LPGR	RRE C N		IF			B900
LOAD POSITIVE (64<32)	LPGFR	RRE C N		IF			B910
LOAD REVERSED (32)	LRVR	RRE N3					B91F
LOAD REVERSED (64)	LRVGR	RRE N					B90F
LOAD REVERSED (16)	LRVH	RXY N3	A			B_2	E31F
LOAD REVERSED (32)	LRV	RXY N3	A			B_2	E31E
LOAD REVERSED (64)	LRVG	RXY N	A			B_2	E30F
MONITOR CALL	MC	SI	SP	MO			AF
MOVE (character)	MVC	SS	A		ST	B_1 B_2	D2
MOVE (immediate)	MVI	SI	A		ST	B_1	92
MOVE (immediate)	MVIY	SIY LD	A		ST	B_1	EB52
MOVE INVERSE	MVCIN	SS	A		ST	B_1 B_2	E8
MOVE LONG	MVCL	RR C	A SP	II	ST	R_1 R_2	0E
MOVE LONG EXTENDED	MVCLE	RS C	A SP		ST	R_1 R_3	A8
MOVE LONG UNICODE	MVCLU	RSY C E2	A SP		ST	R_1 R_2	EB8E
MOVE NUMERICS	MVN	SS	A		ST	B_1 B_2	D1
MOVE STRING	MVST	RRE C	A SP	G0	ST	R_1 R_2	B255
MOVE WITH OFFSET	MVO	SS	A		ST	B_1 B_2	F1
MOVE ZONES	MVZ	SS	A		ST	B_1 B_2	D3
MULTIPLY (64<32)	MR	RR	SP				1C
MULTIPLY (64<32)	M	RX	A SP			B_2	5C
MULTIPLY HALFWORD (32)	MH	RX	A			B_2	4C
MULTIPLY HALFWORD IMMEDIATE (32)	MHI	RI					A7C
MULTIPLY HALFWORD IMMEDIATE (64)	MGHI	RI N					A7D
MULTIPLY LOGICAL (64<32)	MLR	RRE N3	SP				B996
MULTIPLY LOGICAL (128<64)	MLGR	RRE N	SP				B986
MULTIPLY LOGICAL (64<32)	ML	RXY N3	A SP			B_2	E396
MULTIPLY LOGICAL (128<64)	MLG	RXY N	A SP			B_2	E386
MULTIPLY SINGLE (32)	MSR	RRE					B252
MULTIPLY SINGLE (64)	MSGR	RRE N					B90C
MULTIPLY SINGLE (64<32)	MSGFR	RRE N					B91C
MULTIPLY SINGLE (32)	MS	RX	A			B_2	71

Figure 7-1 (Part 5 of 9). Summary of General Instructions

	Name	Mne-monic	Characteristics					Op Code
|	MULTIPLY SINGLE (32)	MSY	RXY LD	A			B_2	E351
	MULTIPLY SINGLE (64)	MSG	RXY N	A			B_2	E30C
	MULTIPLY SINGLE (64<32)	MSGF	RXY N	A			B_2	E31C
	OR (32)	OR	RR C					16
	OR (64)	OGR	RRE C N					B981
	OR (32)	O	RX C	A			B_2	56
|	OR (32)	OY	RXY C LD	A			B_2	E356
	OR (64)	OG	RXY C N	A			B_2	E381
	OR (character)	OC	SS C	A		ST	B_1 B_2	D6
	OR (immediate)	OI	SI C	A		ST	B_1	96
|	OR (immediate)	OIY	SIY C LD	A		ST	B_1	EB56
	OR IMMEDIATE (high high)	OIHH	RI C N					A58
	OR IMMEDIATE (high low)	OIHL	RI C N					A59
	OR IMMEDIATE (low high)	OILH	RI C N					A5A
	OR IMMEDIATE (low low)	OILL	RI C N					A5B
	PACK	PACK	SS	A		ST	B_1 B_2	F2
	PACK ASCII	PKA	SS E2	A SP		ST	B_1 B_2	E9
	PACK UNICODE	PKU	SS E2	A SP		ST	B_1 B_2	E1
	PERFORM LOCKED OPERATION	PLO	SS C	A SP	$ GM	ST	FC	EE
	ROTATE LEFT SINGLE LOGICAL (32)	RLL	RSY N3					EB1D
	ROTATE LEFT SINGLE LOGICAL (64)	RLLG	RSY N					EB1C
	SEARCH STRING	SRST	RRE C	A SP	G0		R_2	B25E
	SET ACCESS	SAR	RRE				U_1	B24E
	SET ADDRESSING MODE (24)	SAM24	E N3	SP	T			010C
	SET ADDRESSING MODE (31)	SAM31	E N3	SP	T			010D
	SET ADDRESSING MODE (64)	SAM64	E N		T			010E
	SET PROGRAM MASK	SPM	RR L					04
	SHIFT LEFT DOUBLE	SLDA	RS C	SP	IF			8F
	SHIFT LEFT DOUBLE LOGICAL	SLDL	RS	SP				8D
	SHIFT LEFT SINGLE (32)	SLA	RS C		IF			8B
	SHIFT LEFT SINGLE (64)	SLAG	RSY C N		IF			EB0B
	SHIFT LEFT SINGLE LOGICAL (32)	SLL	RS					89
	SHIFT LEFT SINGLE LOGICAL (64)	SLLG	RSY N					EB0D
	SHIFT RIGHT DOUBLE	SRDA	RS C	SP				8E
	SHIFT RIGHT DOUBLE LOGICAL	SRDL	RS	SP				8C
	SHIFT RIGHT SINGLE (32)	SRA	RS C					8A
	SHIFT RIGHT SINGLE (64)	SRAG	RSY C N					EB0A
	SHIFT RIGHT SINGLE LOGICAL (32)	SRL	RS					88
	SHIFT RIGHT SINGLE LOGICAL (64)	SRLG	RSY N					EB0C
	STORE (32)	ST	RX	A		ST	B_2	50
|	STORE (32)	STY	RXY LD	A		ST	B_2	E350
	STORE (64)	STG	RXY N	A		ST	B_2	E324
	STORE ACCESS MULTIPLE	STAM	RS	A SP		ST	UB	9B
|	STORE ACCESS MULTIPLE	STAMY	RSY LD	A SP		ST	UB	EB9B
	STORE CHARACTER	STC	RX	A		ST	B_2	42

Figure 7-1 (Part 6 of 9). Summary of General Instructions

Name	Mne- monic	Characteristics								Op Code
STORE CHARACTER	STCY	RXY		LD	A			ST	B_2	E372
STORE CHARACTERS UNDER MASK (high)	STCMH	RSY		N	A			ST	B_2	EB2C
STORE CHARACTERS UNDER MASK (low)	STCM	RS			A			ST	B_2	BE
STORE CHARACTERS UNDER MASK (low)	STCMY	RSY		LD	A			ST	B_2	EB2D
STORE CLOCK	STCK	S	C		A		$	ST	B_2	B205
STORE CLOCK EXTENDED	STCKE	S	C		A		$	ST	B_2	B278
STORE HALFWORD	STH	RX			A			ST	B_2	40
STORE HALFWORD	STHY	RXY		LD	A			ST	B_2	E370
STORE MULTIPLE (32)	STM	RS			A			ST	B_2	90
STORE MULTIPLE (32)	STMY	RSY		LD	A			ST	B_2	EB90
STORE MULTIPLE (64)	STMG	RSY		N	A			ST	B_2	EB24
STORE MULTIPLE HIGH	STMH	RSY		N	A			ST	B_2	EB26
STORE PAIR TO QUADWORD	STPQ	RXY		N	A	SP		ST	B_2	E38E
STORE REVERSED (16)	STRVH	RXY		N3	A			ST	B_2	E33F
STORE REVERSED (32)	STRV	RXY		N3	A			ST	B_2	E33E
STORE REVERSED (64)	STRVG	RXY		N	A			ST	B_2	E32F
SUBTRACT (32)	SR	RR	C				IF			1B
SUBTRACT (64)	SGR	RRE	C	N			IF			B909
SUBTRACT (64<32)	SGFR	RRE	C	N			IF			B919
SUBTRACT (32)	S	RX	C		A		IF		B_2	5B
SUBTRACT (32)	SY	RXY	C	LD	A		IF		B_2	E35B
SUBTRACT (64)	SG	RXY	C	N	A		IF		B_2	E309
SUBTRACT (64<32)	SGF	RXY	C	N	A		IF		B_2	E319
SUBTRACT HALFWORD	SH	RX	C		A		IF		B_2	4B
SUBTRACT HALFWORD	SHY	RXY	C	LD	A		IF		B_2	E37B
SUBTRACT LOGICAL (32)	SLR	RR	C							1F
SUBTRACT LOGICAL (64)	SLGR	RRE	C	N						B90B
SUBTRACT LOGICAL (64<32)	SLGFR	RRE	C	N						B91B
SUBTRACT LOGICAL (32)	SL	RX	C		A				B_2	5F
SUBTRACT LOGICAL (32)	SLY	RXY	C	LD	A				B_2	E35F
SUBTRACT LOGICAL (64)	SLG	RXY	C	N	A				B_2	E30B
SUBTRACT LOGICAL (64<32)	SLGF	RXY	C	N	A				B_2	E31B
SUBTRACT LOGICAL WITH BORROW (32)	SLBR	RRE	C	N3						B999
SUBTRACT LOGICAL WITH BORROW (64)	SLBGR	RRE	C	N						B989
SUBTRACT LOGICAL WITH BORROW (32)	SLB	RXY	C	N3	A				B_2	E399
SUBTRACT LOGICAL WITH BORROW (64)	SLBG	RXY	C	N	A				B_2	E389
SUPERVISOR CALL	SVC	RR					¢			0A
TEST ADDRESSING MODE	TAM	E	C	N3						010B
TEST AND SET	TS	S	C		A		$	ST	B_2	93
TEST UNDER MASK	TM	SI	C		A				B_1	91
TEST UNDER MASK	TMY	SIY	C	LD	A				B_1	EB51
TEST UNDER MASK (high high)	TMHH	RI	C	N						A72
TEST UNDER MASK (high low)	TMHL	RI	C	N						A73
TEST UNDER MASK (low high)	TMLH	RI	C	N						A70
TEST UNDER MASK (low low)	TMLL	RI	C	N						A71

Figure 7-1 (Part 7 of 9). Summary of General Instructions

Name	Mne-monic	Characteristics					Op Code
TEST UNDER MASK HIGH	TMH	RI C					A70
TEST UNDER MASK LOW	TML	RI C					A71
TRANSLATE	TR	SS	A		ST	B_1 B_2	DC
TRANSLATE AND TEST	TRT	SS C	A	GM		B_1 B_2	DD
TRANSLATE EXTENDED	TRE	RRE C	A SP		ST	R_1 R_2	B2A5
TRANSLATE ONE TO ONE	TROO	RRE C E2	A SP	GM	ST	RM R_2	B993
TRANSLATE ONE TO TWO	TROT	RRE C E2	A SP	GM	ST	RM R_2	B992
TRANSLATE TWO TO ONE	TRTO	RRE C E2	A SP	GM	ST	RM R_2	B991
TRANSLATE TWO TO TWO	TRTT	RRE C E2	A SP	GM	ST	RM R_2	B990
UNPACK	UNPK	SS	A		ST	B_1 B_2	F3
UNPACK ASCII	UNPKA	SS C E2	A SP		ST	B_1 B_2	EA
UNPACK UNICODE	UNPKU	SS C E2	A SP		ST	B_1 B_2	E2
UPDATE TREE	UPT	E C	A SP	II GM	ST	I4	0102

Explanation:

¢ Causes serialization and checkpoint synchronization.
¢^{1} Causes serialization and checkpoint synchronization when the M_1 and R_2 fields contain all ones and all zeros, respectively.
$ Causes serialization.
A Access exceptions for logical addresses.
A^1 Access exceptions; not all access exceptions may occur; see instruction description for details.
AI Access exceptions for instruction address.
B PER branch event.
B_1 B_1 field designates an access register in the access-register mode.
B_2 B_2 field designates an access register in the access-register mode.
BP B_2 field designates an access register when PSW bits 16 and 17 have the value 01 binary.
C Condition code is set.
Dd Decimal-operand data exception.
E E instruction format.
E2 Extended-translation facility 2.
EX Execute exception.
FC Designation of access registers depends on the function code of the instruction.
G0 Instruction execution includes the implied use of general register 0.
GM Instruction execution includes the implied use of multiple general registers:
General registers 1, 2, and 3 for COMPARE AND FORM CODEWORD.
General registers 0 and 1 for COMPARE UNTIL SUBSTRING EQUAL and PERFORM LOCKED OPERATION.
General registers 0 and 1 for COMPRESSION CALL, TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO
General registers 1 and 2 for TRANSLATE AND TEST.
General registers 0-5 for UPDATE TREE.

Figure 7-1 (Part 8 of 9). Summary of General Instructions

Explanation (Continued):

I1	Access register 1 is implicitly designated in the access-register mode.
I2	Access register 2 is implicitly designated in the access-register mode.
I4	Access register 4 is implicitly designated in the access-register mode.
IF	Fixed-point-overflow exception.
II	Interruptible instruction.
IK	Fixed-point-divide exception.
L	New condition code is loaded.
LD	Long-displacement facility.
MO	Monitor event.
MS	Message-security assist.
N	Instruction is new in z/Architecture as compared to ESA/390.
N3	Instruction is new in z/Architecture and has been added to ESA/390. Any RSY or RXY instructions still use the RSE or RXE format and 12-bit displacements in ESA/390.
R_1	R_1 field designates an access register in the access-register mode.
R_2	R_2 field designates an access register in the access-register mode.
R_3	R_3 field designates an access register in the access-register mode.
RI	RI instruction format.
RIE	RIE instruction format.
RIL	RIL instruction format.
RR	RR instruction format.
RRE	RRE instruction format.
RS	RS instruction format.
RSI	RSI instruction format.
RXY	RXY instruction format.
RSY	RSY instruction format.
RX	RX instruction format.
S	S instruction format.
SI	SI instruction format.
SIY	SIY instruction format.
SP	Specification exception.
SS	SS instruction format.
ST	PER storage-alteration event.
T	Trace exceptions (includes trace table, addressing, and low-address protection).
U_1	R_1 field designates an access register unconditionally.
U_2	R_2 field designates an access register unconditionally.
UB	R_1 and R_3 fields designate access registers unconditionally, and B_2 field designates an access register in the access-register mode.

Figure 7-1 (Part 9 of 9). Summary of General Instructions

ADD

```
AR     R1,R2     [RR]

 ┌─────────┬────┬────┐
 │  '1A'   │ R1 │ R2 │
 └─────────┴────┴────┘
 0         8    12   15
```

```
AGR    R1,R2     [RRE]

 ┌─────────────────┬─────────┬────┬────┐
 │     'B908'      │//////// │ R1 │ R2 │
 └─────────────────┴─────────┴────┴────┘
 0                 16        24   28   31
```

```
AGFR   R1,R2     [RRE]

 ┌─────────────────┬─────────┬────┬────┐
 │     'B918'      │//////// │ R1 │ R2 │
 └─────────────────┴─────────┴────┴────┘
 0                 16        24   28   31
```

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

 ┌─────────┬────┬────┬────┬─────────────┐
 │  '5A'   │ R1 │ X2 │ B2 │     D2      │
 └─────────┴────┴────┴────┴─────────────┘
 0         8    12   16   20            31
```

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

 ┌─────────┬────┬────┬────┬─/──┬─────────┬─────────┐
 │  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '5A'   │
 └─────────┴────┴────┴────┴─/──┴─────────┴─────────┘
 0         8    12   16   20   32        40        47
```

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

 ┌─────────┬────┬────┬────┬─/──┬─────────┬─────────┐
 │  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '08'   │
 └─────────┴────┴────┴────┴─/──┴─────────┴─────────┘
 0         8    12   16   20   32        40        47
```

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

 ┌─────────┬────┬────┬────┬─/──┬─────────┬─────────┐
 │  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '18'   │
 └─────────┴────┴────┴────┴─/──┴─────────┴─────────┘
 0         8    12   16   20   32        40        47
```

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]

 ┌─────────┬────┬────┬────┬─────────────┐
 │  '4A'   │ R1 │ X2 │ B2 │     D2      │
 └─────────┴────┴────┴────┴─────────────┘
 0         8    12   16   20            31
```

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

 ┌─────────┬────┬────┬────┬─/──┬─────────┬─────────┐
 │  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '7A'   │
 └─────────┴────┴────┴────┴─/──┴─────────┴─────────┘
 0         8    12   16   20   32        40        47
```

ADD HALFWORD IMMEDIATE

```
AHI    R1,I2     [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A7'   │ R1 │'A' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0         8    12   16               31
```

```
AGHI   R1,I2     [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A7'   │ R1 │'B' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0         8    12   16               31
```

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]

+---------+----+----+
|  '1E'   | R1 | R2 |
+---------+----+----+
0         8    12  15
```

```
ALGR     R1,R2     [RRE]

+----------------+--------+----+----+
|     'B90A'     |////////| R1 | R2 |
+----------------+--------+----+----+
0                16       24   28  31
```

```
ALGFR    R1,R2     [RRE]

+----------------+--------+----+----+
|     'B91A'     |////////| R1 | R2 |
+----------------+--------+----+----+
0                16       24   28  31
```

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

+---------+----+----+----+------------+
|  '5E'   | R1 | X2 | B2 |     D2     |
+---------+----+----+----+------------+
0         8    12   16   20          31
```

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

+---------+----+----+----+-/--+--------+--------+
|  'E3'   | R1 | X2 | B2 |DL2 |  DH2   |  '5E'  |
+---------+----+----+----+-/--+--------+--------+
0         8    12   16   20   32       40      47
```

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

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

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

+---------+----+----+----+-/--+--------+--------+
|  'E3'   | R1 | X2 | B2 |DL2 |  DH2   |  '1A'  |
+---------+----+----+----+-/--+--------+--------+
0         8    12   16   20   32       40      47
```

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]

 ┌────────────────┬────────┬────┬────┐
 │     'B998'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31

ALCGR    R1,R2     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B988'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '98'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '88'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

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)

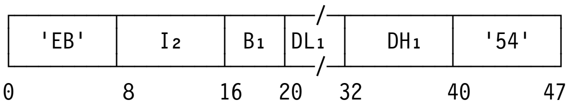
Programming Notes:

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.

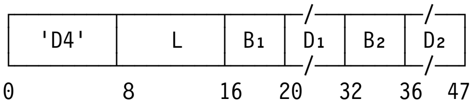
AND

```
NR     R1,R2     [RR]

 ┌────────┬────┬────┐
 │  '14'  │ R1 │ R2 │
 └────────┴────┴────┘
 0        8    12  15

NGR     R1,R2     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B980'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31

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

 ┌────────┬────┬────┬────┬────────────┐
 │  '54'  │ R1 │ X2 │ B2 │     D2     │
 └────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '54'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '80'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47

NI     D1(B1),I2        [SI]

 ┌────────┬────────┬────┬────────────┐
 │  '94'  │   I2   │ B1 │     D1     │
 └────────┴────────┴────┴────────────┘
 0        8        16   20          31
```

NIY $D_1(B_1),I_2$ [SIY]



NC $D_1(L,B_1),D_2(B_2)$ [SS]



The AND of the first and second operands is placed at the first-operand location.

The connective AND 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 positions in both operands contain ones; otherwise, the result bit is set to zero.

For AND (NC), 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.

For AND (NI, NIY), the first operand is one byte in length, and only one byte is stored.

For AND (NR, N, NY), the operands are 32 bits, and for AND (NGR, NG), they are 64 bits.

The displacements for N, NI, and both operands of NC are treated as 12-bit unsigned binary integers. The displacement for NY, NIY, and NG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

Program Exceptions:

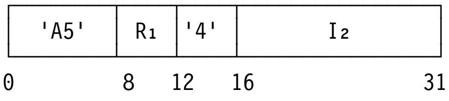
- Access (fetch, operand 2, N, NY, NG, and NC; fetch and store, operand 1, NI, NIY, and NC)
- Operation (NY and NIY, if the long-displacement facility is not installed)

Programming Notes:

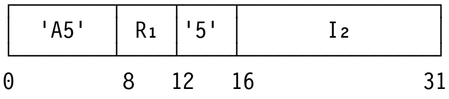
1. An example of the use of the AND instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The AND instruction may be used to set a bit to zero.
3. Accesses to the first operand of AND (NI) and AND (NC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, the instruction AND 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” on page A-43.

AND IMMEDIATE

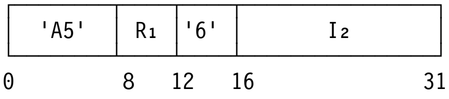
NIHH R_1,I_2 [RI]



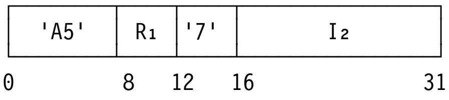
NIHL R_1,I_2 [RI]



NILH R_1,I_2 [RI]



NILL R_1,I_2 [RI]



The second operand is ANDed 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 ANDed with the second operand and then replaced are as follows:

Instruction	Bits ANDed and Replaced
NIHH	0-15
NIHL	16-31
NILH	32-47
NILL	48-63

The connective AND 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 positions in both operands contain ones; 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.

BRANCH AND LINK

```
BALR   R₁,R₂      [RR]

 ┌─────────┬────┬────┐
 │  '05'   │ R₁ │ R₂ │
 └─────────┴────┴────┘
0          8   12  15
```

```
BAL    R₁,D₂(X₂,B₂)     [RX]

 ┌─────────┬────┬────┬────┬────────────┐
 │  '45'   │ R₁ │ X₂ │ B₂ │     D₂     │
 └─────────┴────┴────┴────┴────────────┘
0          8   12   16   20           31
```

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

The link information in the 24-bit addressing mode consists of the instruction-length code (ILC), the condition code (CC), the program-mask bits, and the rightmost 24 bits of the updated instruction address, arranged in bit positions 32-63 of the first-operand location in the following format:

```
┌──┬──┬────┬─────────────────────┐
│IL│  │Prog│                     │
│C │CC│Mask│ Instruction Address │
└──┴──┴────┴─────────────────────┘
32 34 36    40                  63
```

The instruction-length code is 1 or 2.

The link information in the 31-bit addressing mode consists of bit 32 of the PSW, the basic-addressing-mode bit (always a one) and the rightmost 31 bits of the updated instruction address, arranged in bit positions 32-63 of the first-operand location in the following format:

```
┌─┬──────────────────────────────┐
│1│     Instruction Address      │
└─┴──────────────────────────────┘
32                              63
```

In the 24-bit or 31-bit addressing mode, bits 0-31 of the first-operand location remain unchanged.

The link information in the 64-bit addressing mode consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R₂ are used to generate the branch address; however, when the R₂ field is zero, the operation is performed without branching. The branch address is computed before general register R₁ is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace (R₂ field nonzero, BALR only)

Programming Notes:

1. An example of the use of the BRANCH AND LINK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When the R₂ field in the RR format is zero, the link information is loaded without branching.
3. The BRANCH AND LINK instruction (BAL and BALR) is provided for compatibility purposes. It is recommended that, where possible, the BRANCH AND SAVE instruction (BAS and

BASR), BRANCH RELATIVE AND SAVE, or BRANCH RELATIVE AND SAVE LONG be used and BRANCH AND LINK avoided, since the latter places nonzero information in bit positions 32-39 of the link register in the 24-bit addressing mode, which may lead to problems. Additionally, in the 24-bit addressing mode, BRANCH AND LINK may be slower than the other instructions because BRANCH AND LINK must construct the ILC, condition code, and program mask to be placed in bit positions 32-39 of the link register.

4. The condition-code and program-mask information, which is provided in the leftmost byte of the link information only in the 24-bit addressing mode, can be obtained in any addressing mode by means of the INSERT PROGRAM MASK instruction.

BRANCH AND SAVE

```
BASR   R1,R2      [RR]

┌─────────┬────┬────┐
│  '0D'   │ R1 │ R2 │
└─────────┴────┴────┘
0          8   12  15
```

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

┌─────────┬────┬────┬────┬────────────┐
│  '4D'   │ R1 │ X2 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0          8   12   16   20          31
```

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged.

In the 64-bit addressing mode, the link information consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching. The branch address is computed before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace (R_2 field nonzero, BASR only)

Programming Notes:

1. An example of the use of the BRANCH AND SAVE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The BRANCH AND SAVE instruction (BAS and BASR) is intended to be used for linkage to programs known to be in the same addressing mode as the caller. This instruction should be used in place of the BRANCH AND LINK instruction (BAL and BALR). See the programming notes on pages 5-12 and 5-18 in the section “Subroutine Linkage without the Linkage Stack” for a detailed discussion of this and other linkage instructions. See also the programming note under BRANCH AND LINK for a discussion of the advantages of the BRANCH AND SAVE instruction.

BRANCH AND SAVE AND SET MODE

```
BASSM  R1,R2      [RR]

┌─────────┬────┬────┐
│  '0C'   │ R1 │ R2 │
└─────────┴────┴────┘
0          8   12  15
```

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, if the R_2 field is nonzero, the addressing-mode bits and instruction address in the PSW are replaced as specified by the second operand.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged. In the 64-bit addressing mode, the link information is bits 64-126 of the PSW with a one appended on the right, placed in bit positions 0-63 of the first-operand location.

The contents of general register R_2 specify the new addressing mode and designate the branch address; however, when the R_2 field is zero, the operation is performed without branching and without setting either addressing-mode bit.

When the contents of general register R_2 are used and bit 63 of the register is zero, bit 31 of the current PSW, the extended-addressing-mode bit, is set to zero, bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The branch address replaces the instruction address in the PSW.

When the contents of general register R_2 are used and bit 63 of the register is one, the following occurs. Bits 31 and 32 of the current PSW are set to one, the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new extended addressing mode, and the branch address replaces the instruction address in the PSW. Bit 63 of the register remains one. However, if R_2 is the same as R_1, the results in the designated general register are as specified for the R_1 register.

The new value for the PSW is computed before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace (R_2 field nonzero)

Programming Notes:

1. An example of the use of the BRANCH AND SAVE AND SET MODE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. BRANCH AND SAVE AND SET MODE is intended to be the principal calling instruction to subroutines which may operate in a different addressing mode from that of the caller. See the programming notes on pages 5-12 and 5-18 in the section “Subroutine Linkage without the Linkage Stack” for a detailed discussion of this and other linkage instructions.
3. An old 24-bit or 31-bit program can use BRANCH AND SAVE AND SET MODE to call a new 64-bit program without any change, provided that bits 0-31 of general register R_2 are all zeros. The old program can load into bit positions 32-63 of general register R_2 a four-byte address constant, which is provided from outside the program, in which bit 63 in the register (bit 31 of the constant in storage) either is or is not one. If the addressing mode is not changed to the 64-bit mode by the execution of the BRANCH AND SAVE AND SET MODE instruction, or even if it is, the called program can set the 64-bit mode by issuing a SET ADDRESSING MODE (SAM64) instruction.
4. See the programming notes on page 5-12 (under “Simple Branch Instructions”).

BRANCH AND SET MODE

```
BSM    R1,R2      [RR]

┌─────────┬────┬────┐
│  '0B'   │ R1 │ R2 │
└─────────┴────┴────┘
0         8   12   15
```

In the 24-bit or 31-bit addressing mode, bit 32 of the current PSW, the basic-addressing-mode bit, is inserted into bit position 32 of the first operand, and bits 0-31 and 33-63 of the operand remain unchanged. In the 64-bit addressing mode, a one is inserted into bit position 63 of the first operand, and bits 0-62 of the operand remain unchanged. Subsequently, the addressing-mode bits and instruction address in the PSW are replaced as specified by the second operand. The action associated with an operand is not performed if the associated R field is zero.

The contents of general register R_2 specify the new addressing mode and designate the branch address; however, when the R_2 field is zero, the operation is performed without branching and without setting either addressing-mode bit.

When the contents of general register R_2 are used and bit 63 of the register is zero, bit 31 of the current PSW, the extended-addressing-mode bit, is set to zero, bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The branch address replaces the instruction address in the PSW.

When the contents of general register R_2 are used and bit 63 of the register is one, the following occurs. Bits 31 and 32 of the current PSW are set to one, the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new extended addressing mode, and the branch address replaces the instruction address in the PSW. Bit 63 of the register remains one. However, if R_2 is the same as R_1, the results in the designated general register are as specified for the R_1 register.

The new value for the PSW is computed before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace

Programming Notes:

1. An example of the use of the BRANCH AND SET MODE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. BRANCH AND SET MODE with an R_1 field of zero is intended to be the standard return instruction in a program entered by means of BRANCH AND SAVE AND SET MODE. It can also be the return instruction in a program entered in the 24-bit or 31-bit addressing mode by means of BRANCH AND SAVE, BRANCH RELATIVE AND SAVE, or BRANCH RELATIVE AND SAVE LONG. BRANCH AND SET MODE with a nonzero R_1 field is intended to be used in a “glue module” to connect either old 24-bit programs and newer programs that are executed in the 31-bit addressing mode or old 24-bit or 31-bit programs and new programs that are executed in the 64-bit addressing mode. See the programming notes on pages 5-12 and 5-18 in the section “Subroutine Linkage without the Linkage Stack” for a detailed discussion of this and other linkage instructions.

BRANCH ON CONDITION

```
BCR    M1,R2     [RR]

┌────────┬────┬────┐
│  '07'  │ M1 │ R2 │
└────────┴────┴────┘
0         8   12  15

BC     M1,D2(X2,B2)     [RX]

┌────────┬────┬────┬────┬────────────┐
│  '47'  │ M1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0         8   12   16   20           31
```

The instruction address in the current PSW is replaced by the branch address if the condition code has one of the values specified by M_1; otherwise, normal instruction sequencing proceeds with the updated instruction address.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching.

The M_1 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	Instruction Bit No. of Mask	Mask Position Value
0	8	8
1	9	4
2	10	2
3	11	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the branch is successful. If the mask bit selected is zero, normal instruction sequencing proceeds with the next sequential instruction.

When the M_1 and R_2 fields of BRANCH ON CONDITION (BCR) are all ones and all zeros, respectively, a serialization and checkpoint-synchronization function is performed.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

1. An example of the use of the BRANCH ON CONDITION instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When a branch is to depend on more than one condition, the pertinent condition codes are specified in the mask as the sum of their mask position values. A mask of 12, for example, specifies that a branch is to be made when the condition code is 0 or 1.
3. When all four mask bits are zeros or when the R_2 field in the RR format contains zero, the branch instruction is equivalent to a no-operation. When all four mask bits are ones, that is, the mask value is 15, the branch is unconditional unless the R_2 field in the RR format is zero.
4. Execution of BCR 15,0 (that is, an instruction with a value of 07F0 hex) may result in significant performance degradation. To ensure optimum performance, the program should avoid use of BCR 15,0 except in cases when the serialization or checkpoint-synchronization function is actually required.
5. Note that the relation between the RR and RX formats in branch-address specification is not the same as in operand-address specification. For branch instructions in the RX format, the branch address is the address specified by X_2, B_2, and D_2; in the RR format, the branch address is contained in the register designated by R_2. For operands, the address specified by X_2, B_2, and D_2 is the operand address, but the register designated by R_2 contains the operand, not the operand address.

BRANCH ON COUNT

```
BCTR   R1,R2     [RR]

┌────────┬────┬────┐
│  '06'  │ R1 │ R2 │
└────────┴────┴────┘
0         8   12  15
```

```
BCTGR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B946'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                 16       24   28  31
```

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

┌────────┬────┬────┬────┬────────────┐
│  '46'  │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0         8   12   16   20           31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '46'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0         8   12   16   20   32       40       47
```

A one is subtracted from the first operand, and the result is placed at the first-operand location. For BRANCH ON COUNT (BCT, BCTR), the first operand and result are treated as 32-bit binary integers, with overflow ignored. For BRANCH ON COUNT (BCTG, BCTGR), the first operand and result are treated as 64-bit binary integers, with overflow ignored. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address in the current PSW is replaced by the branch address.

In the RX or RXY format, the second-operand address is used as the branch address. In the RR or RRE format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching. The branch address is generated before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

1. An example of the use of the BRANCH ON COUNT instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. The first operand and result can be considered as either signed or unsigned binary integers since the result of a binary subtraction is the same in both cases.
3. An initial count of one results in zero, and no branching takes place; an initial count of zero results in -1 and causes branching to be performed; an initial count of -1 results in -2 and causes branching to be performed; and so on. In a loop, branching takes place each time the instruction is executed until the result is again zero. Note that for BCT or BCTR, because of the number range, an initial count of -2^{31} results in a positive value of $2^{31} - 1$, or, for BCTG or BCTGR, an initial count of -2^{63} results in a positive value of $2^{63} - 1$.
4. Counting is performed without branching when the R_2 field in the RR or RRE format contains zero.

BRANCH ON INDEX HIGH

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

┌────────┬────┬────┬────┬────────────┐
│  '86'  │ R1 │ R3 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20          31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'  │ R1 │ R3 │ B2 │DL2 │  DH2   │  '44'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47
```

BRANCH ON INDEX LOW OR EQUAL

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

┌────────┬────┬────┬────┬────────────┐
│  '87'  │ R1 │ R3 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20          31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'  │ R1 │ R3 │ B2 │DL2 │  DH2   │  '45'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47
```

An increment is added to the first operand, and the sum is compared with a compare value. The result of the comparison determines whether branching occurs. Subsequently, the sum is placed at the first-operand location. The second-operand address is used as a branch address. The R_3 field designates registers containing the increment and the compare value.

For BRANCH ON INDEX HIGH, when the sum is high, the instruction address in the current PSW is replaced by the branch address. When the sum is low or equal, normal instruction sequencing proceeds with the updated instruction address.

For BRANCH ON INDEX LOW OR EQUAL, when the sum is low or equal, the instruction address in the current PSW is replaced by the branch address. When the sum is high, normal instruction sequencing proceeds with the updated instruction address.

When the R_3 field is even, it designates a pair of registers; the contents of the even and odd registers of the pair are used as the increment and the compare value, respectively. When the R_3 field is odd, it designates a single register, the contents of which are used as both the increment and the compare value.

For purposes of the addition and comparison, all operands and results are treated as 32-bit signed binary integers for BXH and BXLE or as 64-bit signed binary integers for BXHG and BXLEG. Overflow caused by the addition is ignored.

The original contents of the compare-value register are used as the compare value even when that register is also specified to be the first-operand location. The branch address is generated before general register R_1 is changed.

The sum is placed at the first-operand location, regardless of whether the branch is taken.

Condition Code: The code remains unchanged.

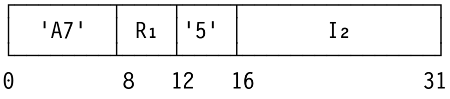
Program Exceptions: None.

Programming Notes:

1. Several examples of the use of the BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL instructions are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The word “index” in the names of these instructions indicates that one of the major purposes is the incrementing and testing of an index value. The increment, being a signed binary integer, may be used to increase or decrease the value in general register R_1 by an arbitrary amount, subject to the limit of the integer size.
3. Care must be taken in the 31-bit addressing mode when a data area in storage is at the rightmost end of a 31-bit address space and a BRANCH ON INDEX HIGH (BXH) or BRANCH ON INDEX LOW OR EQUAL (BXLE) instruction is used to step upward through the data. Since the addition and comparison operations performed during the execution of these instructions treat the operands as 32-bit signed binary integers, the value following 2^{31} - 1 is not 2^{31}, which cannot be represented in that format, but -2^{31}. The instruction does not provide an indication of such overflow. Consequently, some common looping techniques based on the use of these instructions do not work when a data area ends at address 2^{31} - 1. This problem is illustrated in a BRANCH ON INDEX LOW OR EQUAL example in Appendix A, “Number Representation and Instruction-Use Examples.” A similar caution applies in the 64-bit addressing mode when data is at the end of a 64-bit address space and BRANCH ON INDEX HIGH (BXHG) or BRANCH ON INDEX LOW OR EQUAL (BXLEG) is used.

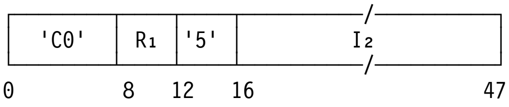
BRANCH RELATIVE AND SAVE

BRAS R_1,I_2 [RI]



BRANCH RELATIVE AND SAVE LONG

BRASL R_1,I_2 [RIL]



Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged.

In the 64-bit addressing mode, the link information consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

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 branch address.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

1. The operation is the same as that of the BRANCH AND SAVE (BAS) instruction except for the means of specifying the branch address. An example of the use of BRANCH AND SAVE is given in Appendix A.
2. The BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG instructions, like the BRANCH AND SAVE instruction, are intended to be used for linkage to programs known to be in the same addressing mode as the caller. These instructions should be used in place of the BRANCH AND LINK instruction (BAL and BALR). See the programming notes on pages 5-12 and 5-18 in the section“Subroutine

Linkage without the Linkage Stack" for a detailed discussion of these and other linkage instructions. See also the programming note under BRANCH AND LINK for a discussion of the advantages of the BRANCH RELATIVE AND SAVE, BRANCH RELATIVE AND SAVE LONG, and BRANCH AND SAVE instructions.

3. When the instruction is the target of EXECUTE, the branch is relative to the target address; see "Branch-Address Generation" on page 5-9.

BRANCH RELATIVE ON CONDITION

```
BRC     M1,I2      [RI]

 ┌────────┬────┬────┬────────────────┐
 │  'A7'  │ M1 │'4' │       I2       │
 └────────┴────┴────┴────────────────┘
 0         8    12   16              31
```

BRANCH RELATIVE ON CONDITION LONG

```
BRCL     M1,I2      [RIL]

 ┌────────┬────┬────┬─────────────/─────┐
 │  'C0'  │ M1 │'4' │            I2     │
 └────────┴────┴────┴─────────────/─────┘
 0         8    12   16                 47
```

The instruction address in the current PSW is replaced by the branch address if the condition code has one of the values specified by M_1; otherwise, normal instruction sequencing proceeds with the updated instruction address.

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 branch address.

The M_1 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	Instruction Bit No. of Mask	Mask Position Value
0	8	8
1	9	4
2	10	2
3	11	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the branch is successful. If the mask bit selected is zero, normal instruction sequencing proceeds with the next sequential instruction.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

1. The operation is the same as that of the BRANCH ON CONDITION instruction except for the means of specifying the branch address. An example of the use of BRANCH ON CONDITION is given in Appendix A.

2. When a branch is to depend on more than one condition, the pertinent condition codes are specified in the mask as the sum of their mask position values. A mask of 12, for example, specifies that a branch is to be made when the condition code is 0 or 1.

3. When all four mask bits are zeros, the branch instruction is equivalent to a no-operation. When all four mask bits are ones, that is, the mask value is 15, the branch is unconditional.

4. When the instruction is the target of EXECUTE, the branch is relative to the target address; see "Branch-Address Generation" on page 5-9.

BRANCH RELATIVE ON COUNT

```
BRCT     R1,I2      [RI]

 ┌────────┬────┬────┬────────────────┐
 │  'A7'  │ R1 │'6' │       I2       │
 └────────┴────┴────┴────────────────┘
 0         8    12   16              31
```

```
BRCTG     R1,I2     [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A7'   │ R1 │'7' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0          8   12   16              31
```

A one is subtracted from the first operand, and the result is placed at the first-operand location. For BRANCH RELATIVE ON COUNT (BRCT), the first operand and result are treated as 32-bit binary integers, with overflow ignored. For BRANCH RELATIVE ON COUNT (BRCTG), the first operand and result are treated as 64-bit binary integers, with overflow ignored. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address in the current PSW is replaced by the branch address.

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 branch address.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

1. The operation is the same as that of the BRANCH ON COUNT instruction except for the means of specifying the branch address. An example of the use of BRANCH ON COUNT is given in Appendix A.
2. The first operand and result can be considered as either signed or unsigned binary integers since the result of a binary subtraction is the same in both cases.
3. An initial count of one results in zero, and no branching takes place; an initial count of zero results in -1 and causes branching to be executed; an initial count of -1 results in -2 and causes branching to be executed; and so on. In a loop, branching takes place each time the instruction is executed until the result is again zero. Note that for BRCT, because of the number range, an initial count of -2^{31} results in a positive value of $2^{31} - 1$, or, for BRCTG, an initial count of -2^{63} results in a positive value of $2^{63} - 1$.
4. When the instruction is the target of EXECUTE, the branch is relative to the target address; see “Branch-Address Generation” on page 5-9.

BRANCH RELATIVE ON INDEX HIGH

```
BRXH    R1,R3,I2     [RSI]

 ┌─────────┬────┬────┬────────────────┐
 │  '84'   │ R1 │ R3 │       I2       │
 └─────────┴────┴────┴────────────────┘
 0          8   12   16              31

BRXHG    R1,R3,I2     [RIE]

 ┌─────────┬────┬────┬──/──┬────────┬────────┐
 │  'EC'   │ R1 │ R3 │  I2 │////////│  '44'  │
 └─────────┴────┴────┴──/──┴────────┴────────┘
 0          8   12   16    32       40      47
```

BRANCH RELATIVE ON INDEX LOW OR EQUAL

```
BRXLE    R1,R3,I2     [RSI]

 ┌─────────┬────┬────┬────────────────┐
 │  '85'   │ R1 │ R3 │       I2       │
 └─────────┴────┴────┴────────────────┘
 0          8   12   16              31

BRXLG    R1,R3,I2     [RIE]

 ┌─────────┬────┬────┬──/──┬────────┬────────┐
 │  'EC'   │ R1 │ R3 │  I2 │////////│  '45'  │
 └─────────┴────┴────┴──/──┴────────┴────────┘
 0          8   12   16    32       40      47
```

An increment is added to the first operand, and the sum is compared with a compare value. The result of the comparison determines whether branching occurs. Subsequently, the sum is placed at the first-operand location. The R_3 field designates registers containing the increment and the compare value.

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 branch address.

For BRANCH RELATIVE ON INDEX HIGH, when the sum is high, the instruction address in the current PSW is replaced by the branch address. When the sum is low or equal, normal instruction sequencing proceeds with the updated instruction address.

For BRANCH RELATIVE ON INDEX LOW OR EQUAL, when the sum is low or equal, the instruction address in the current PSW is replaced by the branch address. When the sum is high, normal instruction sequencing proceeds with the updated instruction address.

When the R_3 field is even, it designates a pair of registers; the contents of the even and odd registers of the pair are used as the increment and the compare value, respectively. When the R_3 field is odd, it designates a single register, the contents of which are used as both the increment and the compare value.

For purposes of the addition and comparison, all operands and results are treated as 32-bit signed binary integers for BRXH and BRXLE or as 64-bit signed binary integers for BRXHG and BRXLG. Overflow caused by the addition is ignored.

The original contents of the compare-value register are used as the compare value even when that register is also specified to be the first-operand location.

The sum is placed at the first-operand location, regardless of whether the branch is taken.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Programming Notes:

1. The operations are the same as those of the BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL instructions except for the means of specifying the branch address. Several examples of the use of BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL are given in Appendix A.

2. The word “index” in the names of these instructions indicates that one of the major purposes is the incrementing and testing of an index value. The increment, being a signed binary integer, may be used to increase or decrease the value in general register R_1 by an arbitrary amount.

3. Care must be taken in the 31-bit addressing mode when a data area in storage is at the rightmost end of an address space and a BRANCH RELATIVE ON INDEX HIGH (BRXH) or BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLE) instruction is used to step upward through the data. Since the addition and comparison operations performed during the execution of these instructions treat the operands as 32-bit signed binary integers, the value following $2^{31} - 1$ is not 2^{31}, which cannot be represented in that format, but -2^{31}. The instruction does not provide an indication of such overflow. Consequently, some common looping techniques based on the use of these instructions do not work when a data area ends at address $2^{31} - 1$. This problem is illustrated in a BRANCH ON INDEX LOW OR EQUAL example in Appendix A. A similar caution applies in the 64-bit addressing mode when data is at the end of a 64-bit address space and BRANCH RELATIVE ON INDEX HIGH (BRXHG) or BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLG) is used.

4. When the instruction is the target of EXECUTE, the branch is relative to the target address; see “Branch-Address Generation” on page 5-9.

CHECKSUM

```
CKSM     R₁,R₂     [RRE]

┌───────────────┬────────┬────┬────┐
│    'B241'     │////////│ R₁ │ R₂ │
└───────────────┴────────┴────┴────┘
0               16       24   28  31
```

Successive four-byte elements of the second operand are added to the first operand in bit positions 32-63 of general register R_1 to form a 32-bit checksum in those bit positions. The first operand and the four-byte elements are treated as 32-bit unsigned binary integers. After each addition of an element, a carry out of bit position 32 of the first operand is added to bit position 63 of the first operand. Bits 0-31 of general register R_1 always remain unchanged. If the second operand is not a multiple of four bytes, its last one, two, or three bytes are treated as appended on the right with the number of all-zeros bytes needed to form a four-byte element. The four-byte elements are added to the first operand until either the entire second operand or a CPU-determined amount of the second operand has been processed. The result is indicated in the condition code.

The R_2 field designates 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 second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified by the 32-bit or 64-bit unsigned binary integer in the $R_2 + 1$ general register.

The handling of the address in general register R_2 and the length in general register $R_2 + 1$ is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 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 register 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 of the register constitute the address. In the 24-bit or 31-bit addressing mode, the length is a 32-bit unsigned binary integer in bit positions 32-63 of general register $R_2 + 1$, and the contents of bit positions 0-31 are ignored. In the 64-bit addressing mode, the length is a 64-bit unsigned binary integer in the register.

The addition of second-operand four-byte elements to the first operand proceeds left to right, four-byte element by four-byte element, and ends as soon as (1) the entire second operand has been processed or (2) a lesser CPU-determined amount of the second operand has been processed. In either case, the result in bit positions 32-63 of general register R_1 is a 32-bit checksum for the part of the second operand that has been processed. When the second operand is not a multiple of four bytes, the final second-operand bytes in excess of a multiple of four are conceptually appended on the right with an appropriate number of all-zeros bytes to form the final four-byte element.

If the operation ends because the entire second operand has been processed, the condition code is set to 0. If the operation ends because a lesser CPU-determined amount of the second operand has been processed, the condition code is set to 3. When the operation is to end with a setting of condition code 3, any carry out of bit position 32 of the first operand is added to bit position 63 of the first operand before the operation ends.

At the completion of the operation, the 32-bit or 64-bit operand-length field in the $R_2 + 1$ register is decremented by the number of actual second-operand bytes added to the first operand (not including any conceptually appended all-zeros bytes), and the address in the R_2 register is incremented by the same number. Thus, the the 32-bit or 64-bit operand-length field contains a zero value if the condition code is set to 0, or it contains a nonzero value if the condition code is set to 3. In the 24-bit or 31-bit addressing mode, bits 0-31 of the $R_2 + 1$ register always remain unchanged.

When condition code 3 is set, the general registers used by the instruction have been set so that the remainder of the second operand can be processed by simply branching back to reexecute the instruction.

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 minimum amount is four bytes or the number of bytes specified in the $R_2 + 1$ general register, whichever is smaller.

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 register R_2 may be set to zeros or may remain unchanged, even when the initial length in register $R_2 + 1$ is zero. Bits 0-31 of general register R_2 remain unchanged.

When the R_1 register is the same register as the R_2 or $R_2 + 1$ register, the results are unpredictable.

Access exceptions for the portion of the second operand to the right of the last byte processed may or may not be recognized. For a second operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

Access exceptions are not recognized if the R_2 field is odd. When the length of the second operand is zero, no access exceptions are recognized.

Resulting Condition Code:

0 Entire second operand processed
1 --
2 --
3 CPU-determined amount of second operand processed

Program Exceptions:

- Access (fetch, operand 2)
- Specification

Programming Notes:

1. The initial contents of bit positions 32-63 of the R_1 general register contribute to the 32-bit checksum. The program normally should set those contents to all zeros before issuing the CHECKSUM instruction.

2. A 16-bit checksum is used in, for example, the TCP/IP application. The following program can be executed after the CHECKSUM instruction to produce in bit positions 32-63 of general register R_2 a 16-bit checksum from the 32-bit checksum in bit positions 32-63 of general register R_1. The program is annotated to show the contents of bit positions 32-63 of the R_2 and $R_2 + 1$ registers after the execution of each instruction. The contents of bit positions 32-63 of the R_1 register are represented as A,B, meaning the value A in bit positions 32-47 and the value B in bit positions 48-63. The value C is a carry from A + B. Note that bit positions 32-63 of register $R_2 + 1$ are known to contain all zeros when CHECKSUM has set condition code 0.

```
                       R2 Bits        R2+1 Bits
Program                32-63          32-63

LR     R2,R1           A,B            0,0
SRDL   R2,16           0,A            B,0
ALR    R2,R2+1         B,A            B,0
ALR    R2,R1           A+B+C,A+B      B,0
SRL    R2,16           0,A+B+C        B,0
```

3. The CHECKSUM instruction may be used in computing hash values as illustrated in the following programming example. The variable KEY contains a string to be mapped into a slot in a hash table. The variable SIZE is a prime number designating the size of the hash table. The value of SIZE is determined by (a) the number of strings to be hashed into the table divided by the acceptable number of hash collisions, and (b) a value that is not too close to a power of two. Following the DIVIDE (D) instruction, the remainder in register 0 represents the resulting hash value.

```
      SR    1,1     Zero accumulator
      LA    2,KEY   Point to string
      LA    3,L'KEY Load string length
LOOP  CKSM  1,2     Compute checksum
      BNZ   LOOP    Repeat if not done
      SR    0,0     Zero for divide
      D     0,SIZE  Compute hash value
      ...
KEY   DS    CL64    String to be hashed
SIZE  DS    F       Size of hash table
```

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

5. The storage-operand references of CHECKSUM may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

6. Figure 7-2 on page 7-34 contains a summary of the operation.

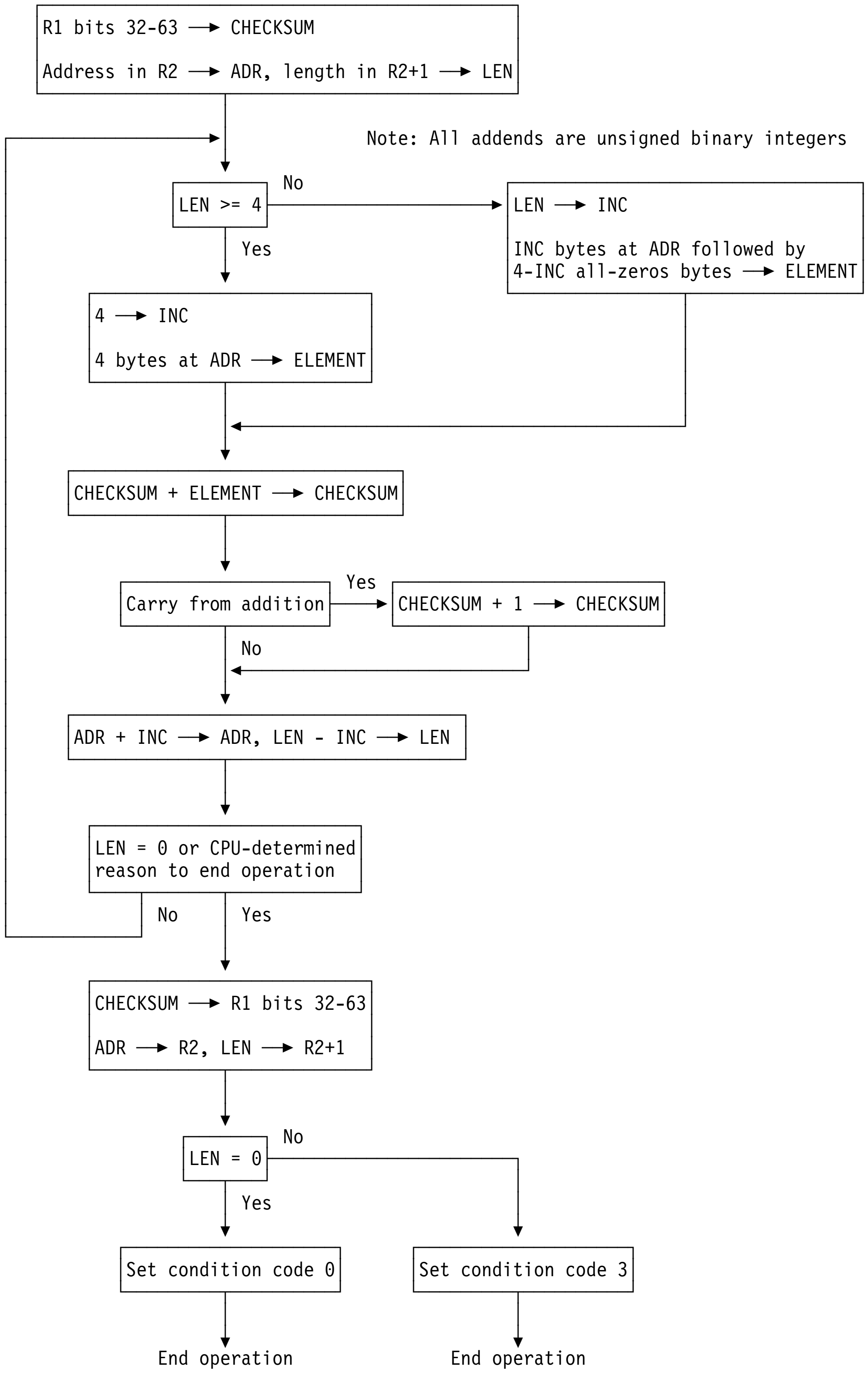


Figure 7-2. Execution of CHECKSUM

CIPHER MESSAGE (KM)

KM R_1,R_2 [RRE]

'B92E'	////////	R_1	R_2
0	16	24	28 31

CIPHER MESSAGE WITH CHAINING (KMC)

KMC R_1,R_2 [RRE]

'B92F'	////////	R_1	R_2
0	16	24	28 31

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figures 7-3 and 7-4 show the assigned function codes for CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING, respectively. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 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 general register 1 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 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE are as follows.

Figure 7-3. Function Codes for CIPHER MESSAGE

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KM-Query	16	—
1	KM-DEA	8	8
2	KM-TDEA-128	16	8
3	KM-TDEA-192	24	8

Explanation:

— Not applicable

The function codes for CIPHER MESSAGE WITH CHAINING are as follows.

Figure 7-4. Function Codes for CIPHER MESSAGE WITH CHAINING

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KMC-Query	16	—
1	KMC-DEA	16	8
2	KMC-TDEA-128	24	8
3	KMC-TDEA-192	32	8

Explanation:

— Not applicable

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, R_2, and $R_1 + 1$ are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key in the parameter block, and the result is placed in the first-operand location. For CIPHER MESSAGE WITH CHAINING, ciphering also uses an initial chaining value in the parameter block, and the chaining value is updated as part of the operation.

The R_1 field designates a general register and must designate an even-numbered register; otherwise, a specification exception is recognized.

The R_2 field designates 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 and second operands is specified by the contents of the R_1 and R_2 general registers, respectively. The number of bytes in the second-operand location is specified in general register $R_2 + 1$. The first operand is the same length as the second operand.

As part of the operation, the addresses in general registers R_1 and R_2 are incremented by the number of bytes processed, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the addresses and length 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 addresses of the first and second operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R_1 and R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R_1 and R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R_1 and R_2, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; and the updated value replaces the contents of general register $R_2 + 1$.

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

Figure 7-5 on page 7-37 shows the contents of the general registers just described.

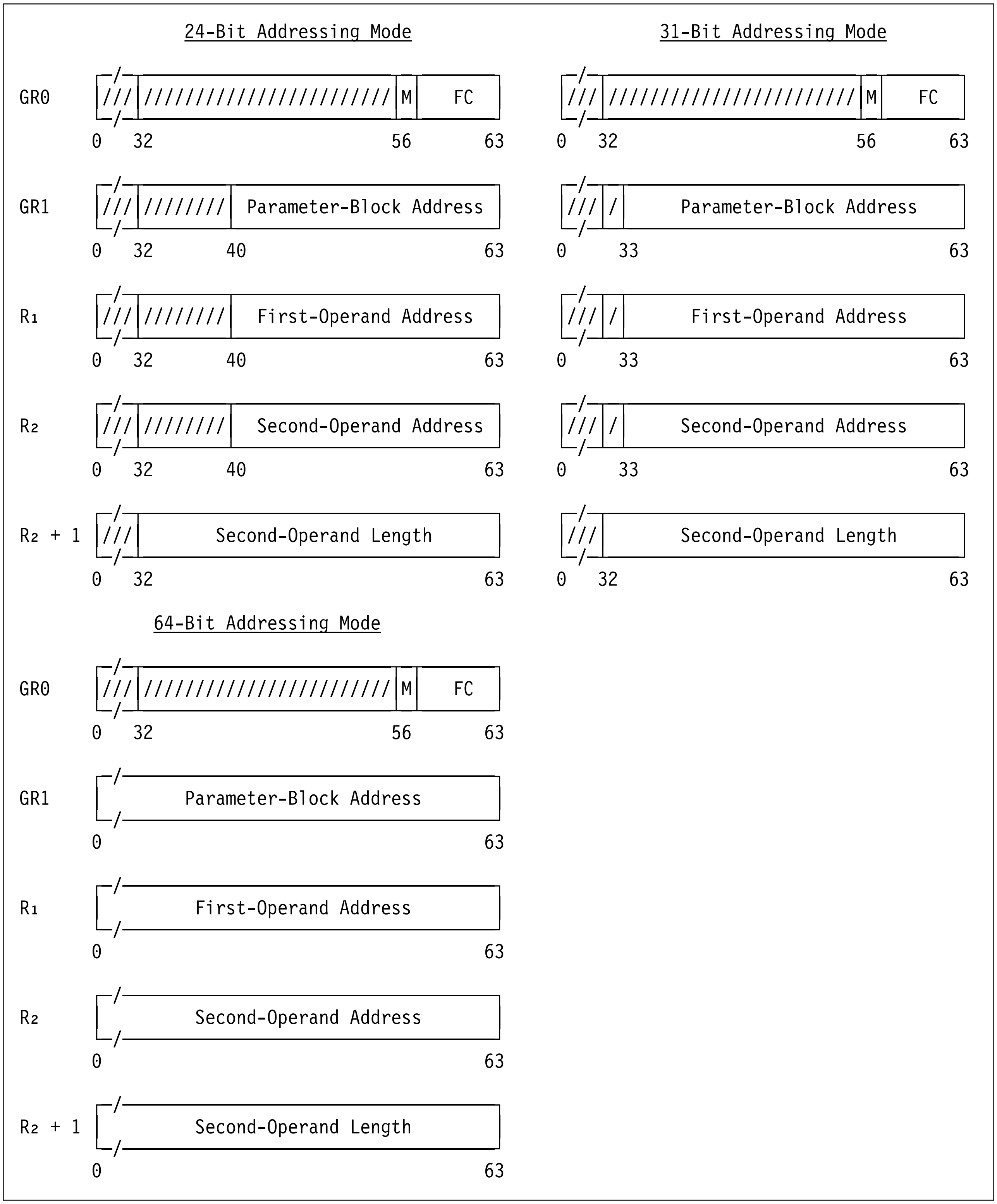


Figure 7-5. General Register Assignment for KM and KMC

In the access-register mode, access registers 1, R₁, and R₂ specify the address spaces containing the parameter block, first, and second operands, respectively.

The result is obtained as if processing starts at the left end of both the first and second operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second operand as specified in general register

$R_2 + 1$ have been processed and placed at the first-operand location (called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location and the chaining-value field are unpredictable if any of the following situations occur:

1. The cryptographic-key field overlaps any portion of the first operand.
2. The chaining-value field overlaps any portion of the first operand or the second operand.
3. The first and second operands overlap destructively. Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, and second operands are not accessed, general registers R_1, R_2, and $R_2 + 1$ are not changed, and condition code 0 is set.

When the contents of the R_1 and R_2 fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by twice the number of bytes processed.

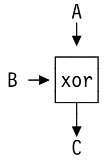
As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers and chaining value to reflect the last unit of the first and second operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

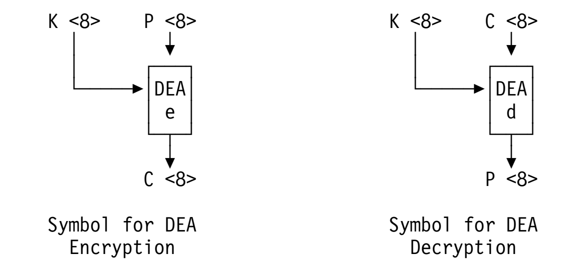
Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key. Further description of the data-encryption algorithm may be found in *Data Encryption Algorithm*, ANSI-X3.92.1981, American National Standard for Information Systems.



C = A XOR B

Figure 7-6. Symbol For Bit-Wise Exclusive Or



Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-7. Symbols for DEA Encryption and Decryption

KM-Query (KM Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
     ┌──────────────────────────────┐
  0  │                              │
     ├         Status Word          ┤
  8  │                              │
     └──────────────────────────────┘
     0                             63
```

Figure 7-8. Parameter Block for KM-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KM-Query function completes; condition code 3 is not applicable to this function.

KM-DEA (KM Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
     ┌──────────────────────────┐
  0  │  Cryptographic Key (K)   │
     └──────────────────────────┘
     0                         63
```

Figure 7-9. Parameter Block for KM-DEA

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in the following figure:

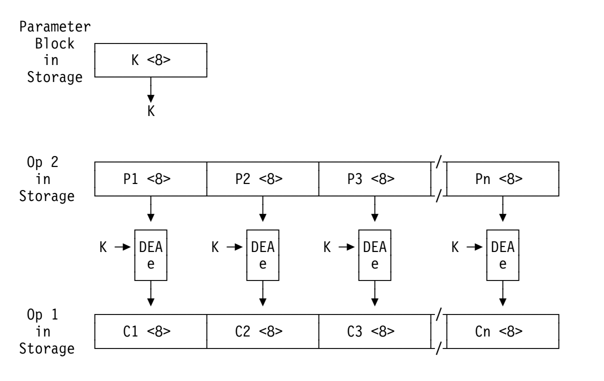


Figure 7-10. KM-DEA Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The operation is shown in the following figure:

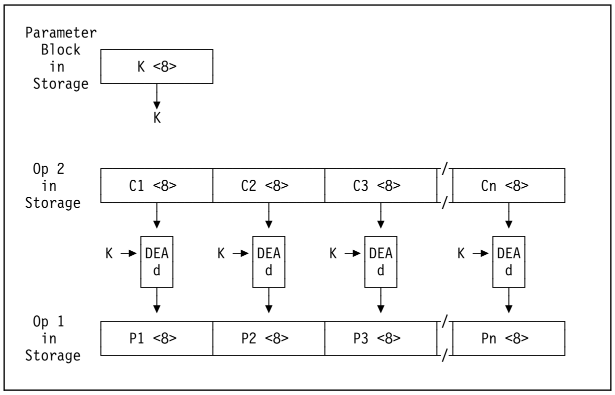


Figure 7-11. KM-DEA Decipher Operation

KM-TDEA-128 (KM Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
 0  ┌──────────────────────────┐
    │ Cryptographic Key 1 (K1) │
 8  ├──────────────────────────┤
    │ Cryptographic Key 2 (K2) │
    └──────────────────────────┘
    0                         63
```

Figure 7-12. Parameter Block for KM-TDEA-128

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA (triple DEA) algorithm with the two 64-bit cryptographic keys in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in the following figure:

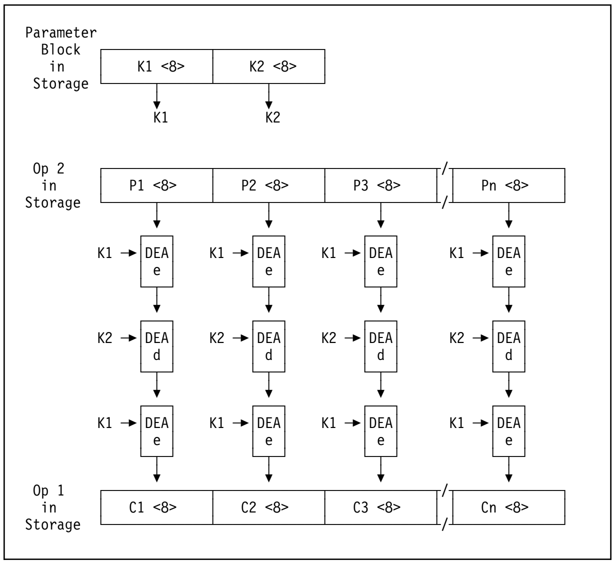


Figure 7-13. KM-TDEA-128 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The operation is shown in the following figure:

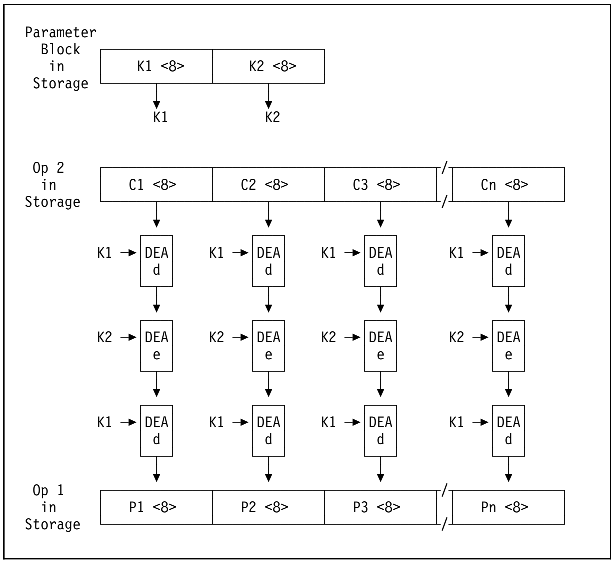


Figure 7-14. KM-TDEA-128 Decipher Operation

KM-TDEA-192 (KM Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
    ┌──────────────────────────┐
  0 │Cryptographic Key 1 (K1)  │
    ├──────────────────────────┤
  8 │Cryptographic Key 2 (K2)  │
    ├──────────────────────────┤
 16 │Cryptographic Key 3 (K3)  │
    └──────────────────────────┘
    0                        63
```

Figure 7-15. Parameter Block for KM-TDEA-192

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in the following figure:

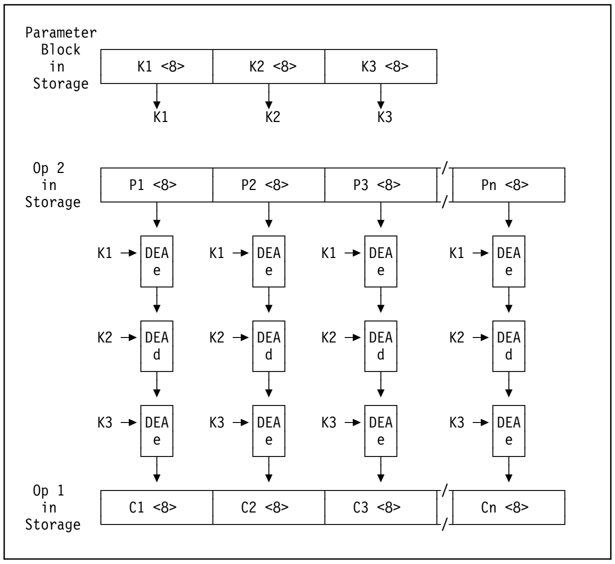


Figure 7-16. KM-TDEA-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The operation is shown in the following figure:

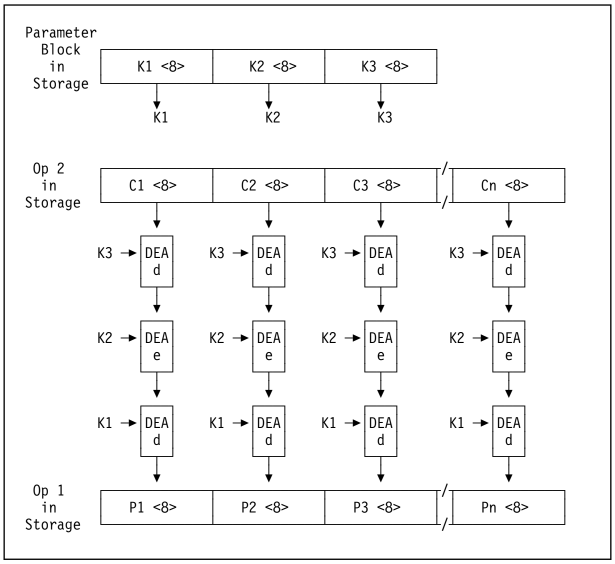


Figure 7-17. KM-TDEA-192 Decipher Operation

KMC-Query (KMC Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
   ┌──────────────────────────┐
 0 │                          │
   ├       Status Word        ┤
 8 │                          │
   └──────────────────────────┘
   0                        63
```

Figure 7-18. Parameter Block for KMC-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH CHAINING instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMC-Query function completes; condition code 3 is not applicable to this function.

KMC-DEA (KMC Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
 0  | Chaining Value (CV)   |
 8  | Cryptographic Key (K) |
    0                     63
```

Figure 7-19. Parameter Block for KMC-DEA

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in the following figure:

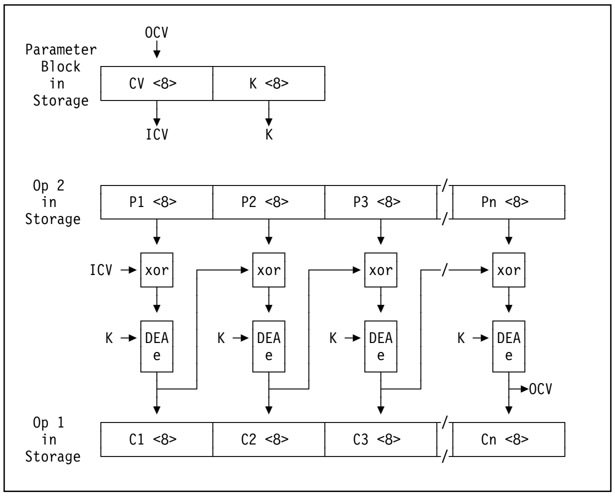


Figure 7-20. KMC-DEA Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:

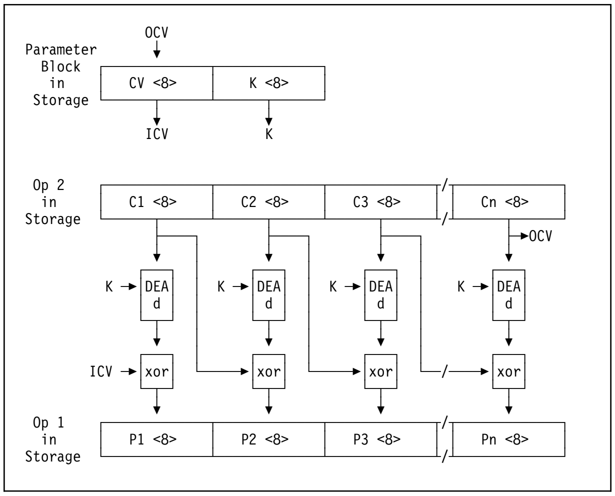


Figure 7-21. KMC-DEA Decipher Operation

KMC-TDEA-128 (KMC Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
 0  | Chaining Value (CV)        |
 8  | Cryptographic Key 1 (K1)   |
16  | Cryptographic Key 2 (K2)   |
    0                          63
```

Figure 7-22. Parameter Block for KMC-TDEA-128

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in the following figure:

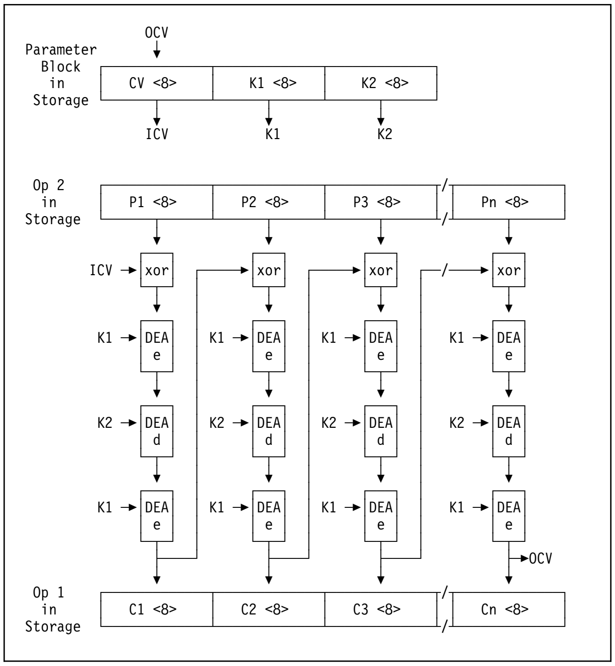


Figure 7-23. KMC-TDEA-128 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:

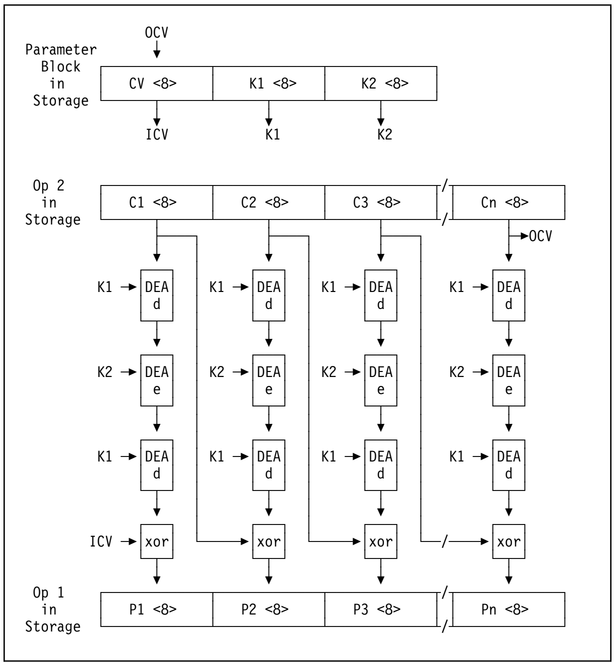


Figure 7-24. KMC-TDEA-128 Decipher Operation

KMC-TDEA-192 (KMC Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

	0 — 63
0	Chaining Value (CV)
8	Cryptographic Key 1 (K1)
16	Cryptographic Key 2 (K2)
24	Cryptographic Key 3 (K3)

Figure 7-25. Parameter Block for KMC-TDEA-192

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in the following figure:

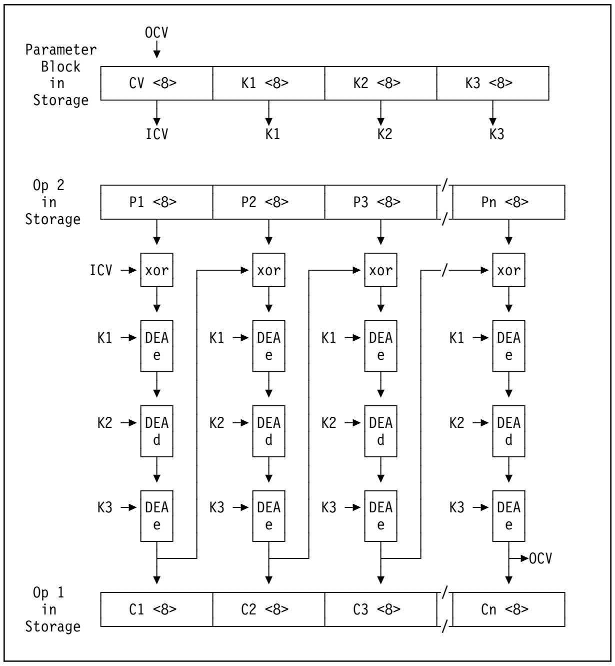


Figure 7-26. KMC-TDEA-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:

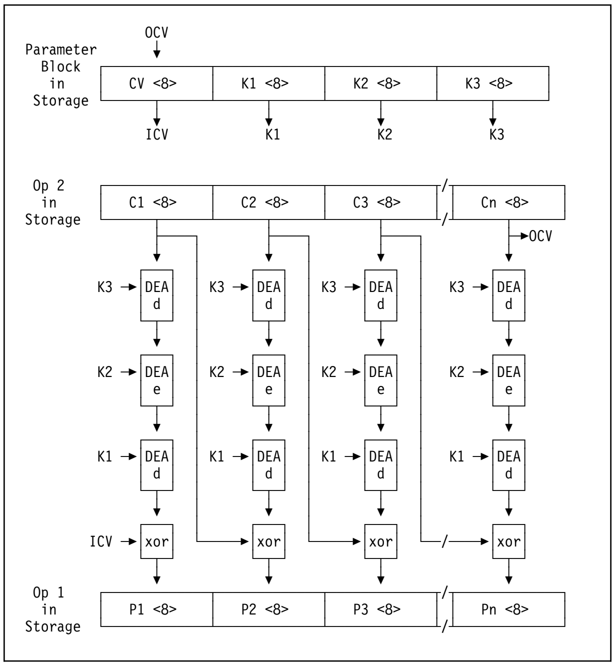


Figure 7-27. KMC-TDEA-192 Decipher Operation

Special Conditions for KM and KMC

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R_1 or R_2 field designates an odd-numbered register or general register 0.
3. The second operand length is not a multiple of the data block size of the designated function (see Figure 7-3 on page 7-35 to determine the data block sizes for CIPHER MESSAGE functions; see Figure 7-4 on page 7-35 to determine the data block sizes for CIPHER MESSAGE WITH CHAINING functions). This specification-exception condition does not apply to the query functions.

Resulting Condition Code:

0 Normal completion
1 --
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2 and cryptographic key; store, operand 1; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)

- 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 R_1,R_2 [RR]

'19'	R_1	R_2
0	8	12 15

CGR R_1,R_2 [RRE]

'B920'	////////	R_1	R_2
0	16	24	28 31

CGFR R_1,R_2 [RRE]

'B930'	////////	R_1	R_2
0	16	24	28 31

```
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

The operation consists in comparing the first and third operands halfword by halfword and incrementing the index until an unequal pair of halfwords is found or the index exceeds the index limit. This proceeds in units of operation, between which interruptions may occur.

At the start of a unit of operation, the index, bits 48-63 of general register 2, is logically compared with the index limit. If the index is larger, the instruction is completed by placing bits 32-63 of general register 3, with bit 32 set to one, in bit positions 32-63 of general register 2, and by setting condition code 0.

If the index is less than or equal to the index limit, the index is applied to the first-operand and third-operand base addresses to locate the current pair of halfwords to be compared. The index, with 48 leftmost zeros appended, and bits 32-63 of general register 1, with 32 leftmost zeros appended, are added to form a 64-bit intermediate value. A carry out of bit position 32, if any, is ignored. The address of the current first-operand halfword is generated from the intermediate value by following the normal rules for operand address generation. The address of the current third-operand halfword is formed in the same manner by adding bits 32-63 of general register 3 and the index.

The current first-operand and third-operand halfwords are logically compared. If they are equal, the contents of general register 2 are incremented by 2, and a unit of operation ends.

If the compare values are unequal, the contents of general register 2 are incremented by 2 and then shifted left logically by 16 bit positions. The shifting occurs only within bit positions 32-63. If the operand-control bit is zero, (1) the one's complement of the higher halfword is placed in bit positions 48-63 of general register 2, and (2) if operand 1 was higher, bits 32-63 of general registers 1 and 3 are interchanged. If the operand-control bit is one, (1) the lower halfword is placed in bit positions 48-63 of general register 2, and (2) if operand 1 was lower, bits 32-63 of general registers 1 and 3 are interchanged.

For the purpose of recognizing access exceptions, operand 1 and operand 3 are both considered to have a length equal to 2 more than the value of the index limit minus the index.

Operation in the 64-Bit Addressing Mode

The operation consists in comparing the first and third operands in units of six bytes at a time and incrementing the index until an unequal pair of six-byte units is found or the index exceeds the index limit. This proceeds in units of operation, between which interruptions may occur.

At the start of a unit of operation, the index, bits 48-63 of general register 2, is logically compared with the index limit. If the index is larger, the instruction is completed by placing bits 0-63 of general register 3, with bit 0 set to one, in bit positions 0-63 of general register 2, and by setting condition code 0.

If the index is less than or equal to the index limit, the index is applied to the first-operand and third-operand base addresses to locate the current pair of six-byte units to be compared. The index, with 48 leftmost zeros appended, and bits 0-63 of general register 1 are added to form the 64-bit address of the current first-operand six-byte unit. A carry out of bit position 0, if any, is ignored. The address of the current third-operand six-byte unit is formed in the same manner by adding bits 0-63 of general register 3 and the index.

The current first-operand and third-operand six-byte units are logically compared. If they are equal, the contents of general register 2 are incremented by 6, and a unit of operation ends.

If the compare values are unequal, the contents of general register 2 are incremented by 6 and then shifted left logically by 48 bit positions. If the operand-control bit is zero, (1) the one's complement of the higher six-byte unit is placed in bit positions 16-63 of general register 2, and (2) if operand 1 was higher, bits 0-63 of general registers 1 and 3 are interchanged. If the operand-control bit is one, (1) the lower six-byte unit is placed in bit positions 16-63 of general register 2, and (2) if operand 1 was lower, bits 0-63 of general registers 1 and 3 are interchanged.

For the purpose of recognizing access exceptions, operand 1 and operand 3 are both considered to have a length equal to 6 more than the value of the index limit minus the index.

Specifications Independent of Addressing Mode

The condition code is unpredictable if the instruction is interrupted.

When the index is initially larger than the index limit, access exceptions are not recognized for the storage operands. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the byte being processed. Access exceptions are not recognized when a specification-exception condition exists.

If the B_2 field designates general register 2, it is unpredictable whether or not the index limit is recomputed; thus, in this case the operand length is unpredictable. However, in no case can the operands exceed 2^{15} bytes in length.

Resulting Condition Code:

0 Operands equal
1 Operand-control bit zero and operand 1 low, or operand-control bit one and operand 3 low
2 Operand-control bit zero and operand 1 high, or operand-control bit one and operand 3 high
3 --

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Specification

Programming Notes:

1. An example of the use of COMPARE AND FORM CODEWORD is given in “Sorting Instructions” in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The offset of the halfword or six-byte unit (depending on the addressing mode) of the first and third operands at which comparison is to begin should be placed in bit positions 48-63 of general register 2 before executing COMPARE AND FORM CODEWORD. The index limit derived from the second-operand address should be the offset of the last halfword or six-byte unit of the first and third operands for which comparison can be made. When the operands do not compare equal, the leftmost 16 bits of the codeword formed in general register 2 (bits 32-47 of the register in the 24-bit or 31-bit addressing mode, or bits 0-15 in the 64-bit addressing mode) by the execution of COMPARE AND FORM CODEWORD gives the offset of the first halfword or six-byte unit not compared. If the codewords compare equal in an UPDATE TREE operation, bit positions 32-47 of general register 2 in the 24-bit or 31-bit addressing mode, or bit positions 0-15 in the 64-bit addressing mode, will contain the offset at which another COMPARE AND FORM CODEWORD should resume comparison for breaking codeword ties. Operand-control-bit values of zero or one are used for sorting operands in ascending or descending order, respectively. Refer to “Sorting Instructions” on page A-51 for a discussion of the use of codewords in sorting.

3. The condition code indicates the results of comparing operands up to 32,768 bytes long. Equal operands result in a negative codeword in bit positions 32-63 of general register 2 in the 24-bit or 31-bit addressing mode, or in bit positions 0-63 in the 64-bit addressing mode. A negative codeword also results in the 24-bit or 31-bit mode when the index limit is 32,766 and the operands that are compared differ in only their last two bytes, or in the 64-bit mode when the limit is 32,762 and the operands differ in only their last six bytes. If this latter codeword is used by UPDATE TREE, an incorrect result may be indicated in general registers 0 and 1. Therefore, the index limit should not exceed 32,764 in the 24-bit or 31-bit mode, or 32,760 in the 64-bit mode, when the resulting codeword is to be used by UPDATE TREE.

4. Special precautions should be taken if COMPARE AND FORM CODEWORD is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

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

6. The storage-operand references of COMPARE AND FORM CODEWORD may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

7. Figure 7-29 on page 7-49 and Figure 7-30 on page 7-50 contain summaries of the operation in the 24-bit or 31-bit addressing mode,

and Figure 7-31 on page 7-51 and Figure 7-32 on page 7-52 contain summaries of the operation in the 64-bit addressing mode.

Operand-Control Bit	Relation	Resulting Condition Code	Result in GR2 (Bits 32-63)	Result in GR1 (Bits 32-63)	Result in GR3 (Bits 32-63)
0	op1 = op3	0	OGR3b1	-	-
0	op1 < op3	1	X, nop3	-	-
0	op1 > op3	2	X, nop1	OGR3	OGR1
1	op1 = op3	0	OGR3b1	-	-
1	op1 < op3	2	X, top1	OGR3	OGR1
1	op1 > op3	1	X, top3	-	-

Explanation:

- The bits remain unchanged.

OGR1 The original value of GR1 bits 32-63.

OGR3 The original value of GR3 bits 32-63.

OGR3b1 The original value of GR3 bits 32-63 with bit 32 set to one.

X Bits 32-47 of GR2 are 2 more than the index of the first unequal halfword.

nop1 Bits 48-63 of GR2 are the one's complement of the first unequal halfword in operand 1.

nop3 Bits 48-63 of GR2 are the one's complement of the first unequal halfword in operand 3.

top1 Bits 48-63 of GR2 are the first unequal halfword in operand 1.

top3 Bits 48-63 of GR2 are the first unequal halfword in operand 3.

Figure 7-29. Operation of COMPARE AND FORM CODEWORD in the 24-Bit or 31-bit Addressing Mode

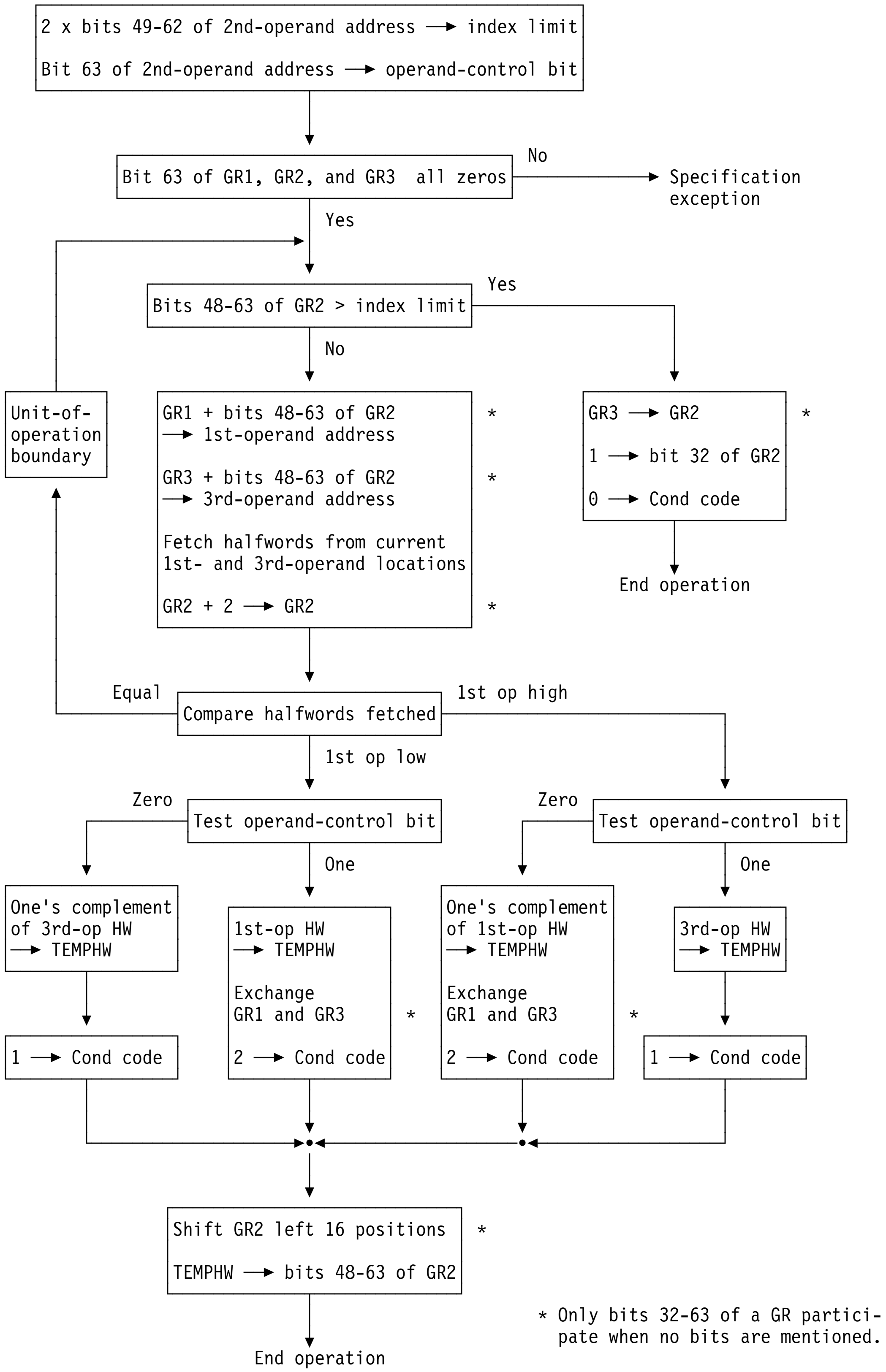

2 x bits 49-62 of 2nd-operand address → index limit
Bit 63 of 2nd-operand address → operand-control bit
Bit 63 of GR1, GR2, and GR3 all zeros
No
Specification exception
Yes
Bits 48-63 of GR2 > index limit
Yes
No
Unit-of-operation boundary
GR1 + bits 48-63 of GR2 → 1st-operand address *
GR3 + bits 48-63 of GR2 → 3rd-operand address *
Fetch halfwords from current 1st- and 3rd-operand locations
GR2 + 2 → GR2 *
GR3 → GR2 *
1 → bit 32 of GR2
0 → Cond code
End operation
Equal
Compare halfwords fetched
1st op high
1st op low
Zero
Test operand-control bit
Zero
Test operand-control bit
One
One
One's complement of 3rd-op HW → TEMPHW
1st-op HW → TEMPHW
Exchange GR1 and GR3 *
One's complement of 1st-op HW → TEMPHW
Exchange GR1 and GR3 *
3rd-op HW → TEMPHW
1 → Cond code
2 → Cond code
2 → Cond code
1 → Cond code
Shift GR2 left 16 positions *
TEMPHW → bits 48-63 of GR2
End operation
* Only bits 32-63 of a GR participate when no bits are mentioned.

Figure 7-30. Execution of COMPARE AND FORM CODEWORD in the 24-Bit or 31-bit Addressing Mode

Operand-Control Bit	Relation	Resulting Condition Code	Result in GR2 (Bits 0-63)	Result in GR1 (Bits 0-63)	Result in GR3 (Bits 0-63)
0	op1 = op3	0	OGR3b1	-	-
0	op1 < op3	1	X, nop3	-	-
0	op1 > op3	2	X, nop1	OGR3	OGR1
1	op1 = op3	0	OGR3b1	-	-
1	op1 < op3	2	X, top1	OGR3	OGR1
1	op1 > op3	1	X, top3	-	-

Explanation:

- The bits remain unchanged.

OGR1 The original value of GR1 bits 0-63.

OGR3 The original value of GR3 bits 0-63.

OGR3b1 The original value of GR3 bits 0-63 with bit 0 set to one.

X Bits 0-15 of GR2 are 6 more than the index of the first unequal six-byte unit.

nop1 Bits 16-63 of GR2 are the one's complement of the first unequal six-byte unit in operand 1.

nop3 Bits 16-63 of GR2 are the one's complement of the first unequal six-byte unit in operand 3.

top1 Bits 16-63 of GR2 are the first unequal six-byte unit in operand 1.

top3 Bits 16-63 of GR2 are the first unequal six-byte unit in operand 3.

Figure 7-31. Operation of COMPARE AND FORM CODEWORD in the 64-Bit Addressing Mode

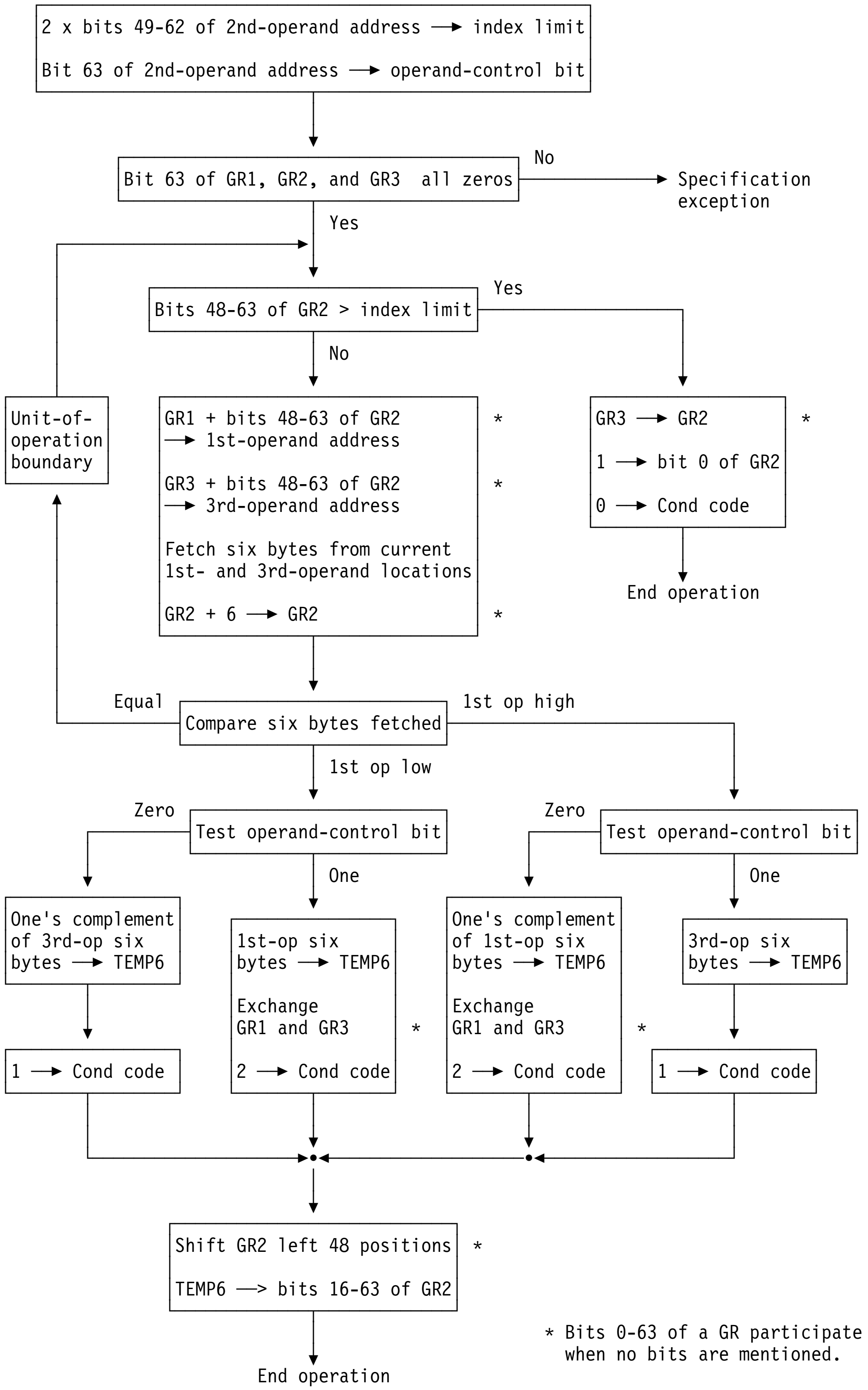
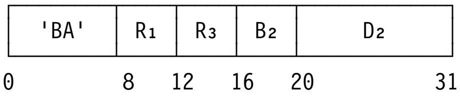

2 x bits 49-62 of 2nd-operand address ──► index limit
Bit 63 of 2nd-operand address ──► operand-control bit
Bit 63 of GR1, GR2, and GR3 all zeros
No
Specification exception
Yes
Bits 48-63 of GR2 > index limit
Yes
No
Unit-of-operation boundary
GR1 + bits 48-63 of GR2 ──► 1st-operand address *
GR3 + bits 48-63 of GR2 ──► 3rd-operand address *
Fetch six bytes from current 1st- and 3rd-operand locations
GR2 + 6 ──► GR2 *
GR3 ──► GR2 *
1 ──► bit 0 of GR2
0 ──► Cond code
End operation
Equal
Compare six bytes fetched
1st op high
1st op low
Zero
Test operand-control bit
Zero
Test operand-control bit
One
One
One's complement of 3rd-op six bytes ──► TEMP6
1st-op six bytes ──► TEMP6
Exchange GR1 and GR3 *
One's complement of 1st-op six bytes ──► TEMP6
Exchange GR1 and GR3 *
3rd-op six bytes ──► TEMP6
1 ──► Cond code
2 ──► Cond code
2 ──► Cond code
1 ──► Cond code
Shift GR2 left 48 positions *
TEMP6 ──> bits 16-63 of GR2
End operation
* Bits 0-63 of a GR participate when no bits are mentioned.

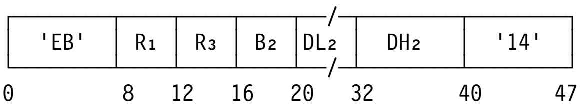
Figure 7-32. Execution of COMPARE AND FORM CODEWORD in the 64-Bit Addressing Mode

COMPARE AND SWAP

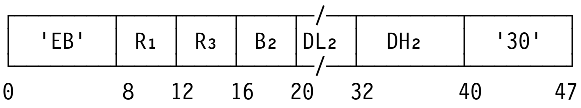
CS $R_1,R_3,D_2(B_2)$ [RS]



CSY $R_1,R_3,D_2(B_2)$ [RSY]

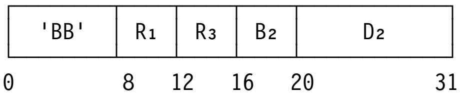


CSG $R_1,R_3,D_2(B_2)$ [RSY]

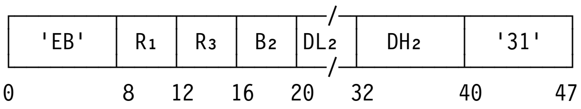


COMPARE DOUBLE AND SWAP

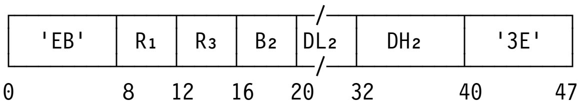
CDS $R_1,R_3,D_2(B_2)$ [RS]



CDSY $R_1,R_3,D_2(B_2)$ [RSY]



CDSG $R_1,R_3,D_2(B_2)$ [RSY]



The first and second operands are compared. If they are equal, the third operand is stored at the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. The result of the comparison is indicated in the condition code.

For COMPARE AND SWAP (CS, CSY), the first and third operands are 32 bits in length, with each operand occupying bit positions 32-63 of a general register. The second operand is a word in storage.

For COMPARE AND SWAP (CSG), the first and third operands are 64 bits in length, with each operand occupying bit positions 0-63 of a general register. The second operand is a doubleword in storage.

For COMPARE DOUBLE AND SWAP (CDS, CDSY), the first and third operands are 64 bits in length. The first 32 bits of an operand occupy bit positions 32-63 of the even-numbered register of an even-odd pair of general registers, and the second 32 bits occupy bit positions 32-63 of the odd-numbered register of the pair. The second operand is a doubleword in storage.

For COMPARE DOUBLE AND SWAP (CDSG), the first and third operands are 128 bits in length. The first 64 bits of an operand occupy bit positions 0-63 of the even-numbered register of an even-odd pair of general registers, and the second 64 bits occupy bit positions 0-63 of the odd-numbered register of the pair. The second operand is a quadword in storage.

When an equal comparison occurs, the third operand is stored at the second-operand location. The fetch of the second operand for purposes of comparison and the store into the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs.

When the result of the comparison is unequal, the second operand is loaded at the first-operand location, and the second-operand location remains unchanged. However, on some models, the contents may be fetched and subsequently stored back unchanged at the second-operand location. This update appears to be a block-concurrent interlocked-update reference as observed by other CPUs.

A serialization function is performed before the operand is fetched and again after the operation is completed.

The displacement for CS and CDS is treated as a 12-bit unsigned binary integer. The displacement for CSY, CSG, CDSY, and CDSG is treated as a 20-bit signed binary integer.

The second operand of COMPARE AND SWAP (CS, CSY) must be designated on a word boundary. The second operand of COMPARE

AND SWAP (CSG) and COMPARE DOUBLE AND SWAP (CDS, CDSY) must be designated on a doubleword boundary. The second operand of COMPARE DOUBLE AND SWAP (CDSG) must be designated on a quadword boundary. The R_1 and R_3 fields for COMPARE DOUBLE AND SWAP must each designate an even-numbered register. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 First and second operands equal, second operand replaced by third operand
1 First and second operands unequal, first operand replaced by second operand
2 --
3 --

Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (CSY and CDSY, if the long-displacement facility is not installed)
- Specification

Programming Notes:

1. Several examples of the use of the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions are given in Appendix A, "Number Representation and Instruction-Use Examples."

2. Some of the following notes are worded, with respect to operand size, for CS, CSY, CDS, and CDSY. Similar notes, worded for a larger operand size, would apply to CSG and CDSG.

3. COMPARE AND SWAP can be used by CPU programs sharing common storage areas in either a multiprogramming or multiprocessing environment. Two examples are:

   a. By performing the following procedure, a CPU program can modify the contents of a storage location even though the possibility exists that the CPU program may be interrupted by another CPU program that will update the location or that another CPU program may simultaneously update the location. First, the entire word containing the byte or bytes to be updated is loaded into a general register. Next, the updated value is computed and placed in another general register. Then COMPARE AND SWAP is executed with the R_1 field designating the register that contains the original value and the R_3 field designating the register that contains the updated value. If the update has been successful, condition code 0 is set. If the storage location no longer contains the original value, the update has not been successful, the general register designated by the R_1 field of the COMPARE AND SWAP instruction contains the new current value of the storage location, and condition code 1 is set. When condition code 1 is set, the CPU program can repeat the procedure using the new current value.

   b. COMPARE AND SWAP can be used for controlled sharing of a common storage area, including the capability of leaving a message (in a chained list of messages) when the common area is in use. To accomplish this, a word in storage can be used as a control word, with a zero value in the word indicating that the common area is not in use and that no messages exist, a negative value indicating that the area is in use and that no messages exist, and a nonzero positive value indicating that the common area is in use and that the value is the address of the most recent message added to the list. Thus, any number of CPU programs desiring to seize the area can use COMPARE AND SWAP to update the control word to indicate that the area is in use or to add messages to the list. The single CPU program which has seized the area can also safely use COMPARE AND SWAP to remove messages from the list.

4. COMPARE DOUBLE AND SWAP can be used in a manner similar to that described for COMPARE AND SWAP. In addition, it has another use. Consider a chained list, with a control word used to address the first message in the list, as described in programming note 2b above. If multiple CPU programs are to be permitted to delete messages by using COMPARE AND SWAP (and not just the single CPU program which has seized the common area), there is a possibility the list will be incorrectly updated. This would occur if, for example, after one CPU program has fetched the address of the most recent message in order to remove the message,

another CPU program removes the first two messages and then adds the first message back into the chain. The first CPU program, on continuing, cannot easily detect that the list is changed. By increasing the size of the control word to a doubleword containing both the first message address and a word with a change number that is incremented for each modification of the list, and by using COMPARE DOUBLE AND SWAP to update both fields together, the possibility of the list being incorrectly updated is reduced to a negligible level. That is, an incorrect update can occur only if the first CPU program is delayed while changes exactly equal in number to a multiple of 2^{32} take place *and* only if the last change places the original message address in the control word.

5. COMPARE AND SWAP and COMPARE DOUBLE AND SWAP do not interlock against storage accesses by channel programs. Therefore, the instructions should not be used to update a location at which a channel program may store, since the channel-program data may be lost.

6. To ensure successful updating of a common storage field by two or more CPUs, all updates must be done by means of an interlocked-update reference. COMPARE AND SWAP, COMPARE AND SWAP AND PURGE, COMPARE DOUBLE AND SWAP, and TEST AND SET are the only instructions that perform an interlocked-update reference. For example, if one CPU executes OR IMMEDIATE and another CPU executes COMPARE AND SWAP to update the same byte, the fetch by OR IMMEDIATE may occur either before the fetch by COMPARE AND SWAP or between the fetch and the store by COMPARE AND SWAP, and then the store by OR IMMEDIATE may occur after the store by COMPARE AND SWAP, in which case the change made by COMPARE AND SWAP is lost.

7. For the case of a condition-code setting of 1, COMPARE AND SWAP and COMPARE DOUBLE AND SWAP may or may not, depending on the model, cause any of the following to occur for the second-operand location: a PER storage-alteration event may be recognized; a protection exception for storing may be recognized; and, provided no access exception exists, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.

8. The performance of CDSG on some models may be significantly slower than that of CSG. When quadword consistency is not required by the program, alternate code sequences should be used.

COMPARE HALFWORD

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

 ┌─────────┬────┬────┬────┬────────────┐
 │  '49'   │ R1 │ X2 │ B2 │     D2     │
 └─────────┴────┴────┴────┴────────────┘
 0         8    12   16   20          31

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

 ┌─────────┬────┬────┬────┬───/┬────────┬─────────┐
 │  'E3'   │ R1 │ X2 │ B2 │DL2 │  DH2   │  '79'   │
 └─────────┴────┴────┴────┴───/┴────────┴─────────┘
 0         8    12   16   20   32       40       47
```

COMPARE HALFWORD IMMEDIATE

```
CHI     R1,I2     [RI]

 ┌─────────┬────┬────┬─────────────────┐
 │  'A7'   │ R1 │'E' │       I2        │
 └─────────┴────┴────┴─────────────────┘
 0         8    12   16               31

CGHI     R1,I2     [RI]

 ┌─────────┬────┬────┬─────────────────┐
 │  'A7'   │ R1 │'F' │       I2        │
 └─────────┴────┴────┴─────────────────┘
 0         8    12   16               31
```

The first operand is compared with the second operand, and the result is indicated in the condition code. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For COMPARE HALFWORD (CH, CHY) and COMPARE HALFWORD IMMEDIATE (CHI), the first operand is treated as a 32-bit signed binary integer. For COMPARE HALFWORD IMMEDIATE (CGHI), the first operand is treated as a 64-bit signed binary integer.

The displacement for CH is treated as a 12-bit unsigned binary integer. The displacement for CHY 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 CH, CHY only)
- Operation (CHY, if the long-displacement facility is not installed)

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

COMPARE LOGICAL

```
CLR    R1,R2      [RR]

┌─────────┬────┬────┐
│  '15'   │ R1 │ R2 │
└─────────┴────┴────┘
0         8   12   15
```

```
CLGR     R1,R2      [RRE]

┌────────────────┬────────┬────┬────┐
│     'B921'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

```
CLGFR     R1,R2      [RRE]

┌────────────────┬────────┬────┬────┐
│     'B931'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

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

┌─────────┬────┬────┬────┬────────────┐
│  '55'   │ R1 │ X2 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8   12   16   20           31
```

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

┌─────────┬────┬────┬────┬───/┬────────┬────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │  DH2   │  '55'  │
└─────────┴────┴────┴────┴───/┴────────┴────────┘
0         8   12   16   20   32       40      47
```

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

┌─────────┬────┬────┬────┬───/┬────────┬────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │  DH2   │  '21'  │
└─────────┴────┴────┴────┴───/┴────────┴────────┘
0         8   12   16   20   32       40      47
```

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

┌─────────┬────┬────┬────┬───/┬────────┬────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │  DH2   │  '31'  │
└─────────┴────┴────┴────┴───/┴────────┴────────┘
0         8   12   16   20   32       40      47
```

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

┌─────────┬─────────┬────┬────────────┐
│  '95'   │   I2    │ B1 │     D1     │
└─────────┴─────────┴────┴────────────┘
0         8        16   20           31
```

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

┌─────────┬─────────┬────┬───/┬────────┬────────┐
│  'EB'   │   I2    │ B1 │DL1 │  DH1   │  '55'  │
└─────────┴─────────┴────┴───/┴────────┴────────┘
0         8        16   20   32       40      47
```

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

┌─────────┬─────────┬────┬───/┬────┬───/┐
│  'D5'   │    L    │ B1 │ D1 │ B2 │ D2 │
└─────────┴─────────┴────┴───/┴────┴───/┘
0         8        16   20   32   36  47
```

The first operand is compared with the second operand, and the result is indicated in the condition code.

For COMPARE LOGICAL (CLR, CL, CLY), the operands are treated as 32 bits. For COMPARE LOGICAL (CLGR, CLG), the operands are treated as 64 bits. For COMPARE LOGICAL (CLGFR, CLGF), the first operand is treated as 64 bits, and the second operand is treated as 32 bits with 32 zeros appended on the left.

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the fields is reached. For COMPARE LOGICAL (CL, CLY, CLG, CLGF, CLC), access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte.

The displacements for CL, CLI, and both operands of CLC are treated as 12-bit unsigned binary integers. The displacement for CLY, CLG, CLGF, and CLIY 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, CL, CLY, CLG, CLGF, and CLC; fetch, operand 1, CLI, CLIY, and CLC)
- Operation (CLY and CLIY if the long-displacement facility is not installed)

Programming Notes:

1. Examples of the use of the COMPARE LOGICAL instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. COMPARE LOGICAL treats all bits of each operand alike as part of a field of unstructured logical data. For COMPARE LOGICAL (CLC), the comparison may extend to field lengths of 256 bytes.

COMPARE LOGICAL CHARACTERS UNDER MASK

```
CLM    R1,M3,D2(B2)     [RS]

 ┌─────────┬────┬────┬────┬────────────┐
 │  'BD'   │ R1 │ M3 │ B2 │     D2     │
 └─────────┴────┴────┴────┴────────────┘
 0         8   12   16   20           31
```

```
CLMY   R1,M3,D2(B2)     [RSY]

 ┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
 │  'EB'   │ R1 │ M3 │ B2 │DL2 │   DH2   │  '21'   │
 └─────────┴────┴────┴────┴────┴─────────┴─────────┘
 0         8   12   16   20   32        40        47
```

```
CLMH     R1,M3,D2(B2)     [RSY]

 ┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
 │  'EB'   │ R1 │ M3 │ B2 │DL2 │   DH2   │  '20'   │
 └─────────┴────┴────┴────┴────┴─────────┴─────────┘
 0         8   12   16   20   32        40        47
```

The first operand is compared with the second operand under control of a mask, and the result is indicated in the condition code.

The contents of the M_3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R_1. For COMPARE LOGICAL CHARACTERS UNDER MASK (CLM, CLMY), the four bytes to which the mask bits correspond are in bit positions 32-63 of general register R_1. For COMPARE LOGICAL CHARACTERS UNDER MASK (CLMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The byte positions corresponding to ones in the mask are considered as a contiguous field and are compared with the second operand. The second operand is a contiguous field in storage, starting at the second-operand address and equal in length to the number of ones in the mask. The bytes in the general register corresponding to zeros in the mask do not participate in the operation.

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the fields is reached.

When the mask is not zero, exceptions associated with storage-operand access are recognized for no more than the number of bytes specified by the mask. Access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte. When the mask is zero, access exceptions are recognized for one byte at the second-operand address.

The displacement for CLM is treated as a 12-bit unsigned binary integer. The displacement for CLMY and CLMH is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Operands equal, or mask bits all zeros
1 First operand low
2 First operand high
3 --

Program Exceptions:

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

Programming Note: An example of the use of the COMPARE LOGICAL CHARACTERS UNDER MASK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

COMPARE LOGICAL LONG

CLCL R_1,R_2 [RR]

'0F'	R_1	R_2
0	8	12 15

The first operand is compared with the second operand, and the result is indicated in the condition code. The shorter operand is considered to be extended on the right 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 7-33 on page 7-59.

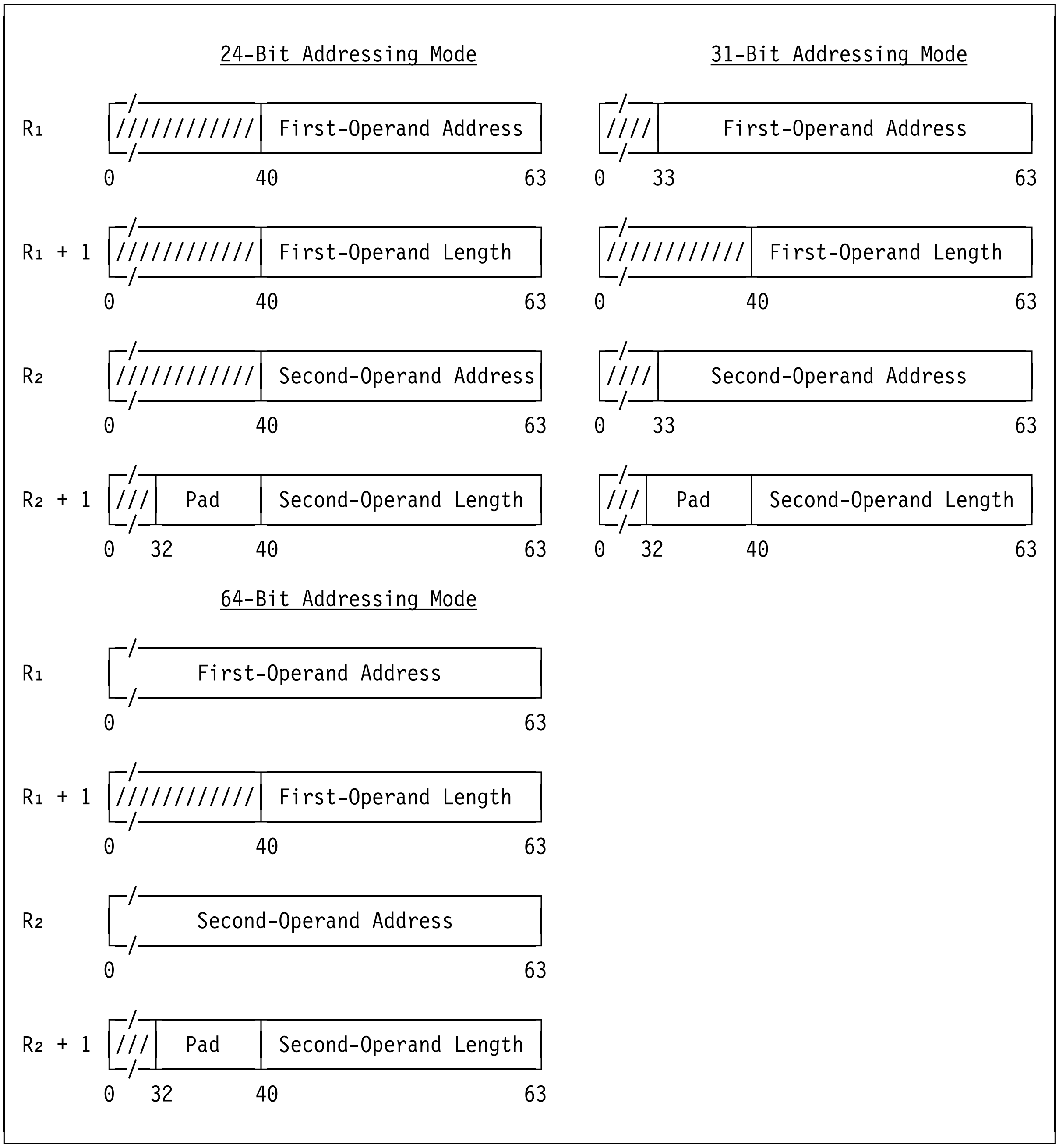


Figure 7-33. Register Contents for COMPARE LOGICAL LONG

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the longer operand is reached. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If both operands are of zero length, the operands are considered to be equal.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the lengths in general registers R_1 + 1 and R_2 + 1 are decremented by the number of bytes compared, 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 after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and its address is updated accordingly.

If the operation ends because of an inequality, the address fields in general registers R_1 and R_2 at completion identify the first unequal byte in each operand. The lengths in bit positions 40-63 of general registers R_1 + 1 and R_2 + 1 are decremented by the number of bytes that were equal, unless the inequality occurred with the padding byte, in which case the length field for the shorter operand is set to zero. The addresses in general registers R_1 and R_2 are incremented by the amounts by which the corresponding length fields were reduced.

If the two operands, including the padding byte, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values.

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_2 are set to zeros, even when one or both of the initial length values are zero. In any addressing mode, the contents of bit positions 0-39 of general registers R_1 + 1 and R_2 + 1 remain unchanged.

Access exceptions for the portion of a storage operand to the right of the first unequal byte may or may not be recognized. For operands longer than 2K bytes, access exceptions are not recognized more than 2K bytes beyond the byte being processed. Access exceptions are not indicated for locations more than 2K bytes beyond the first unequal byte.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Operands equal, or both zero length
1 First operand low
2 First operand high
3 --

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Specification

Programming Notes:

1. An example of the use of the COMPARE LOGICAL LONG instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When the R_1 and R_2 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by the number of bytes compared, not by twice the number of bytes compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, condition code 0 is set. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed.
3. Special precautions should be taken when COMPARE LOGICAL LONG is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.
4. Other programming notes concerning interruptible instructions are included in “Interruptible Instructions” in Chapter 5, “Program Execution.”
5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

COMPARE LOGICAL LONG EXTENDED

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

┌──────────┬────┬────┬────┬────────────┐
│   'A9'   │ R1 │ R3 │ B2 │     D2     │
└──────────┴────┴────┴────┴────────────┘
0          8    12   16   20          31
```

The first operand is compared with the third operand until unequal bytes are compared, the end of the longer operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with padding bytes. 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 7-34 on page 7-62.

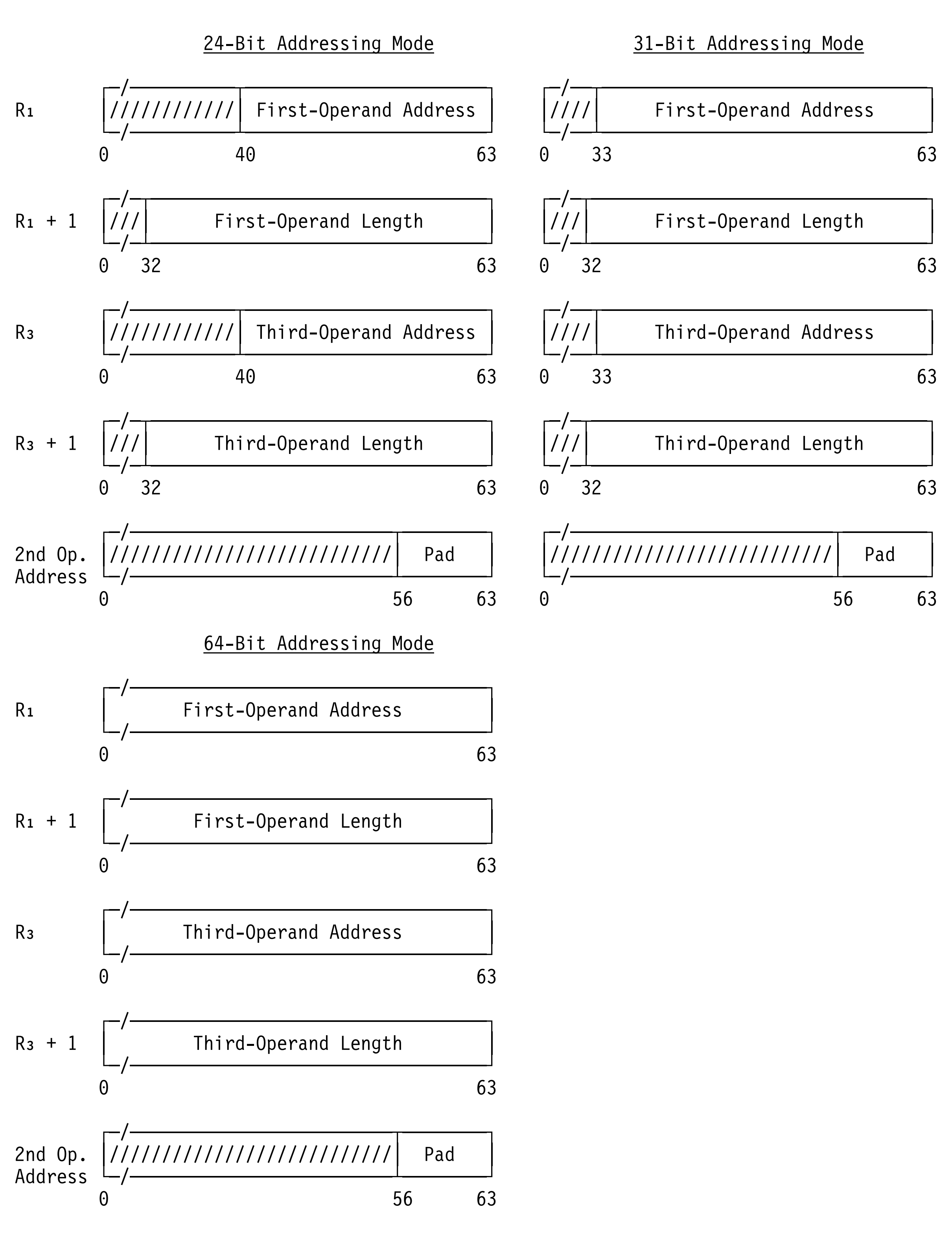


Figure 7-34. Register Contents and Second-Operand Address for COMPARE LOGICAL LONG EXTENDED

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found, the end of the longer operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If both operands are of zero length, the operands are considered to be equal.

If the operation ends because of an inequality, the address fields in general registers R_1 and R_3 at completion identify the first unequal byte in each operand. The lengths in bit positions 32-63, in the 24-bit or 31-bit addressing mode, or in bit positions 0-63, in the 64-bit addressing mode, of general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes that were equal, unless the inequality occurred with the padding byte, in which case the length field for the shorter operand is set to zero. The addresses in general registers R_1 and R_3 are incremented by the amounts by which the corresponding length fields were decremented. Condition code 1 is set if the first operand is low, or condition code 2 is set if the first operand is high.

If the two operands, including the padding byte, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values. Condition code 0 is set.

If the operation is completed because a CPU-determined number of bytes have been compared without finding an inequality or reaching the end of the longer operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes compared, 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 bytes to be compared. If the operation is completed after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and the operand address is updated accordingly. Condition code 3 is set.

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 initial length values are zero.

Access exceptions for the portion of a storage operand to the right of the first unequal byte may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the byte being processed. Access exceptions are not indicated for locations more than 4K bytes beyond the first unequal byte.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 All bytes compared; operands equal, or both zero length
1 All bytes compared, first operand low
2 All bytes compared, first operand high
3 CPU-determined number of bytes compared without finding an inequality

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Specification

Programming Notes:

1. COMPARE LOGICAL LONG EXTENDED is intended for use in place of COMPARE LOGICAL LONG when the operand lengths are specified as 32-bit binary integers. COMPARE LOGICAL LONG EXTENDED sets condition code 3 in cases in which COMPARE LOGICAL LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of bytes that were compared.

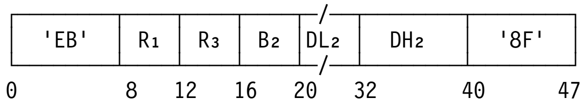
3. The function of not processing more than approximately 4K bytes of either operand is intended to permit software polling of a flag that may be set by a program on another CPU during long operations.

4. When the R_1 and R_3 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by the number of bytes compared, not by twice the number of bytes compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, the condition code is finally set to 0 after possible settings to 3. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed. If storage is not accessed, condition code 3 may or may not be set regardless of the operand length.

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

COMPARE LOGICAL LONG UNICODE

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



The first operand is compared with the third operand until unequal two-byte Unicode characters are compared, the end of the longer operand is reached, or a CPU-determined number of characters have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with two-byte padding characters. 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 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 7-35 on page 7-65.

The comparison proceeds left to right, character by character, and ends as soon as an inequality is found, the end of the longer operand is reached, or a CPU-determined number of characters have been compared, whichever occurs first. If the operands are not of the same length, the shorter operand is considered to be extended on the right

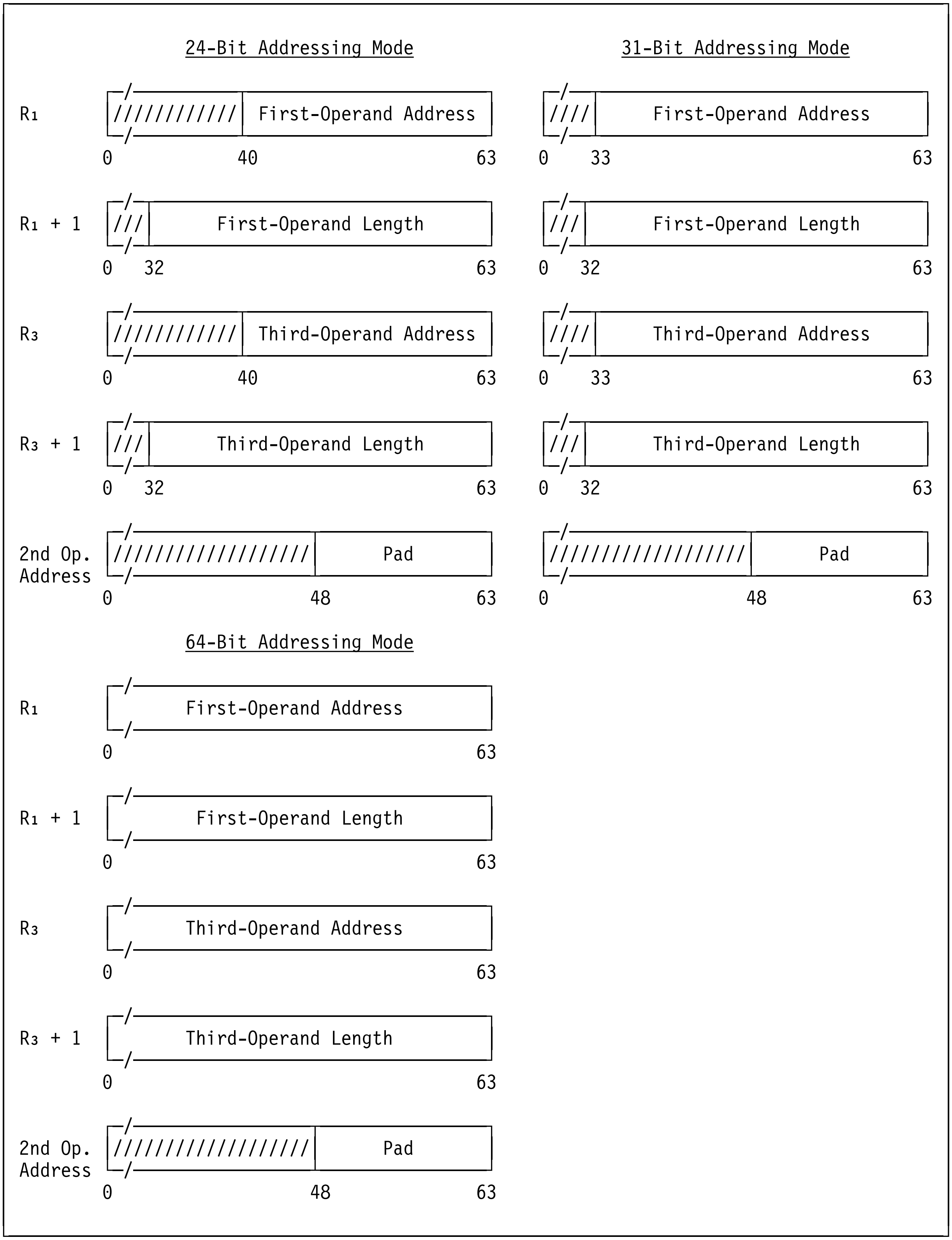


Figure 7-35. Register Contents and Second-Operand Address for COMPARE LOGICAL LONG UNICODE

with the appropriate number of two-byte padding characters.

If both operands are of zero length, the operands are considered to be equal.

If the operation ends because of an inequality, the address fields in general registers R_1 and R_3 at completion identify the first unequal two-byte character in each operand. The lengths in bit positions 32-63, in the 24-bit or 31-bit addressing mode, or in bit positions 0-63, in the 64-bit addressing mode, of general registers $R_1 + 1$ and $R_3 + 1$ are decremented by 2 times the number of characters that were equal, unless the inequality occurred with the two-byte padding character, in which case the length field for the shorter operand is set to zero. The addresses in general registers R_1 and R_3 are incremented by the amounts by which the corresponding length fields were decremented. Condition code 1 is set if the first operand is low, or condition code 2 is set if the first operand is high.

If the two operands, including the two-byte padding character, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values. Condition code 0 is set.

If the operation is completed because a CPU-determined number of characters have been compared without finding an inequality or reaching the end of the longer operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by 2 times the number of characters compared, 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 characters to be compared. If the operation is completed after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and the operand address is updated accordingly. Condition code 3 is set.

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 initial length values are zero.

Access exceptions for the portion of a storage operand to the right of the first unequal character may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the character being processed. Access exceptions are not indicated for locations more than 4K bytes beyond the first unequal character.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field or length associated with that operand is odd.

Resulting Condition Code:

0 All characters compared; operands equal, or both zero length
1 First operand low
2 First operand high
3 CPU-determined number of characters compared without finding an inequality

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

Programming Notes:

1. COMPARE LOGICAL LONG UNICODE is intended for use in place of COMPARE LOGICAL LONG or COMPARE LOGICAL LONG EXTENDED when two-byte characters are to be compared. The characters may be Unicode characters or any other double-byte characters. COMPARE LOGICAL LONG UNICODE sets condition code 3 in cases in which COMPARE LOGICAL LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of characters that were compared.

3. When the R_1 and R_3 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by 2 times the number of characters compared, not by 4 times the number of characters compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, the condition code is finally set to 0 after possible settings to 3. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed. If storage is not accessed, condition code 3 may or may not be set regardless of the operand length.

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

5. 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:

```
        CLCLU   6,8,X'020'
```

COMPARE LOGICAL STRING

```
CLST    R1,R2     [RRE]

┌───────────────┬────────┬────┬────┐
│    'B25D'     │////////│ R1 │ R2 │
└───────────────┴────────┴────┴────┘
0               16       24   28  31
```

The first operand is compared with the second operand until unequal bytes are compared, the end of either operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. The CPU-determined number is at least 256. 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 first and second operands may be of the same or different lengths. The end of an 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 an 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, byte by byte, and ends as soon as the ending character is encountered in either or both operands, unequal bytes which do not include an ending character are compared, or a CPU-determined number of bytes have been compared, whichever occurs first. The CPU-determined number is at least 256. When the ending character is encountered simultaneously in both operands, including when it is in the first byte position of the operands, the operands are of the same length and are considered to be equal, and condition code 0 is set. When the ending character is encountered in only one operand, that operand, which is the shorter operand, is considered to be low, and condition code 1 or 2 is set. Condition code 1 is set if the first operand is low, or condition code 2 is set if the second operand is low. Similarly, when unequal bytes which do not include an ending character are compared, condition code 1 is set if the lower byte is in the first operand, or condition code 2 is set if the lower byte is in the second operand. When a CPU-determined number of bytes have been compared, condition code 3 is set.

When condition code 1 or 2 is set, the address of the last byte processed in the first and second operands is placed in general registers R_1 and R_2, respectively. That is, when condition code 1 is set, the address of the ending character or first unequal byte in the first operand, whichever was encountered, is placed in general register R_1, and the address of the second-operand byte corresponding in position to the first-operand byte is placed in general register R_2. When condition code 2 is set, the address of the ending character or first unequal byte in the second operand, whichever was encountered, is placed in general register R_2, and the address of the first-operand byte corresponding in position to the second-operand byte is placed in general register R_1. 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.

When condition code 0 is set, the contents of general registers R_1 and R_2 remain unchanged.

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 which is necessarily examined in the operation.

Resulting Condition Code:

0 Entire operands equal; general registers R_1 and R_2 unchanged
1 First operand low; general registers R_1 and R_2 updated with addresses of last bytes processed
2 First operand high; general registers R_1 and R_2 updated with addresses of last bytes processed
3 CPU-determined number of bytes equal; general registers R_1 and R_2 updated with addresses of next bytes

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Specification

Programming Notes:

1. Several examples of the use of the COMPARE LOGICAL STRING instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When condition code 0 is set, no indication is given of the position of either ending character.
3. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of bytes that were compared.
4. 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.
5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

COMPARE UNTIL SUBSTRING EQUAL

```
CUSE     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B257'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

The first operand is compared with the second operand until equal substrings (sequences of bytes) of a specified length are found, the end of the longer operand is reached, or a CPU-determined number of unequal bytes have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with padding bytes. The CPU-determined number is at least 256. The result is indicated in the condition code.

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 specified by the contents of the R_1 and R_2 general registers, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the 32-bit signed binary integer in bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively. In the 64-bit addressing mode, the number of bytes is specified by the 64-bit signed binary integer in bit positions 0-63 of those registers. When an operand length is negative, it is treated as zero, and it remains unchanged upon completion of the instruction.

Bits 56-63 of general register 0 specify the unsigned substring length, a value of 0-255, in bytes. Bits 56-63 of general register 1 are the padding byte. Bits 0-55 of general registers 0 and 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 7-36 on page 7-70.

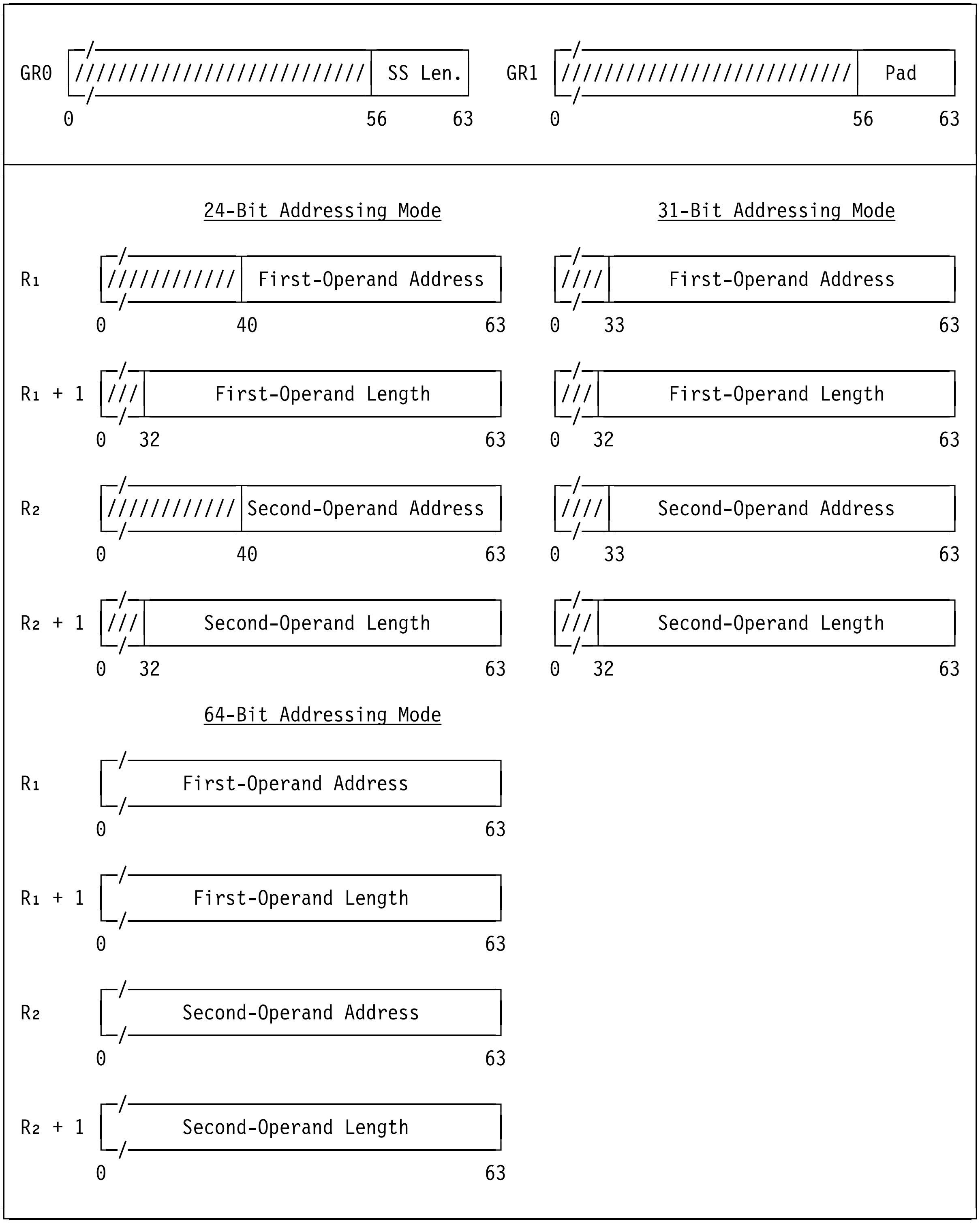


Figure 7-36. Register Contents for COMPARE UNTIL SUBSTRING EQUAL

The result is obtained as if the operands were processed from left to right. However, multiple accesses may be made to all or some of the bytes of each operand.

The comparison proceeds left to right, byte by byte, and ends as soon as (1) equal substrings of the specified length are found, (2) the end of the longer operand is reached without finding equal substrings of the specified length, or (3) the last bytes compared are unequal, and a

CPU-determined number of bytes have been compared. The CPU-determined number is at least 256. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If the operation ends because equal substrings of the specified length were found, the condition code is set to 0. If the operation ends because the end of the longer operand was reached without finding equal substrings of the specified length, the condition code is set to 1 if equal bytes were the last bytes compared, or it is set to 2 if unequal bytes were the last bytes compared. If the operation ends because unequal bytes were compared when a CPU-determined number of bytes had been compared, the condition code is set to 3.

If the specified substring length is zero, it is considered that equal substrings of the specified length were found, and condition code 0 is set.

If both operands are of zero length but the specified substring length is not zero, it is considered that the end of the longer operand was reached when unequal bytes were the last bytes compared, and condition code 2 is set.

If equal bytes have been compared but then unequal bytes are compared, it is considered that all bytes so far compared are unequal.

At the completion of the operation, the operand-length fields in the $R_1 + 1$ and $R_2 + 1$ registers are decremented by the number of unequal bytes compared (including equal bytes before unequal bytes compared), and the addresses in the R_1 and R_2 registers are incremented by the same number. However, in the case when a byte of the longer operand is compared against the padding byte, the length field for the shorter operand is not decremented below zero, and the corresponding address is not incremented above the address of the first byte after the shorter operand. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the addresses in bit positions 32-63 of registers R_1 and R_2 are set to zeros, even if the substring length is zero or both operand lengths are initially zero.

Thus, when condition code 0 or 1 is set, the resulting addresses in the R_1 and R_2 registers designate the first bytes of equal substrings in the two operands, and the lengths in the $R_1 + 1$ and $R_2 + 1$ registers have been decremented by the number of bytes preceding the equal substrings, except when the equal substring in the shorter operand begins with the padding byte, in which case the length field for the shorter operand is zero, and the corresponding address field has been incremented by the operand length. When condition code 2 is set, each address field designates the first byte after the corresponding operand, and both length fields are zero. When condition code 3 is set, each address field designates the first byte after the last compared byte of the corresponding operand, and both length fields have been decremented by the number of bytes compared, except that a length field is not decremented below zero.

When the contents of the R_1 and R_2 fields are the same, the first and second operands may be compared, or the condition code may be set to 0 or 1 without comparing the operands.

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 substring length and padding byte may be fetched from general registers 0 and 1 multiple times during the execution of the instruction, and the registers designated by R_1 and R_2 may be updated multiple times. Therefore, if R_1 or R_2 is zero, the results are unpredictable.

When condition code 3 is set, the general registers used by the instruction have been set so that the remainder of the operands can be processed by simply branching back and reexecuting the instruction.

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 execution of the instruction is interruptible when the last bytes compared are unequal; it is not interruptible when the last bytes compared are equal. When an interruption occurs, other than one that follows termination, the contents of the registers designated by the R_1 and R_2 fields are updated the same as upon normal completion of the instruction, so that the instruction, when reexecuted, resumes at the point of interruption. The condition code is unpredictable.

Access exceptions for the portion of a storage operand to the right of the last byte processed may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Although the operand address and length fields remain unchanged when a zero substring length is specified, the recognition of access exceptions is not necessarily prevented.

Resulting Condition Code:

0 Equal substrings of specified length found
1 End of longer operand reached when last bytes compared are equal
2 End of longer operand reached when last bytes compared are unequal
3 Last bytes compared are unequal, and CPU-determined number of bytes compared

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Specification

Programming Notes:

1. When the R_1 and R_2 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, and, in the absence of dynamic modification of the operand area by another CPU or by a channel program, condition code 0, 1, or 2 is set (as explained in the next note). However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed.

2. If the contents of the R_1 and R_2 fields are the same and the operand length is nonzero, and provided that another CPU or a channel program is not changing an operand, condition code 0 is set if the operand length is equal to or greater than the specified substring length, or condition code 1 is set if the operand length is less than the specified substring length. Whether or not R_1 equals R_2, if both operand lengths are zero, condition code 0 is set if the specified substring length is zero, or condition code 2 is set if the specified substring length is nonzero. In all of these cases, the addresses in the R_1 and R_2 registers and the lengths in the R_1 + 1 and R_2 + 1 registers remain unchanged.

3. Special precautions should be taken when COMPARE UNTIL SUBSTRING EQUAL is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

4. Other programming notes concerning interruptible instructions are included in “Interruptible Instructions” on page 5-21.

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

6. The storage-operand references of COMPARE UNTIL SUBSTRING EQUAL may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

COMPRESSION CALL

```
CMPSC   R₁,R₂             [RRE]
```

```
┌────────────────┬────────┬────┬────┐
│     'B263'     │////////│ R₁ │ R₂ │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

This definition assumes knowledge of the introductory information and information about dictionary formats in *Enterprise Systems Architecture/390 Data Compression,* SA22-7208-01.

The second operand is compressed or expanded, depending on a specification in general register 0, and the results are placed at the first-operand location. The compressed-data operand normally consists of index symbols corresponding to entries in a dictionary designated by an address in general register 1. This dictionary is a compression dictionary during a compression operation or an expansion dictionary during an expansion operation. During compression when format-1 sibling descriptors are specified in general register 0, an expansion dictionary immediately follows the compression dictionary. During compression when the symbol-translation option is specified in general register 0, the index symbols resulting from compression are translated to interchange symbols by means of a symbol-translation table designated by the address and an offset in general register 1, and it is the interchange symbols that are placed at the first-operand location. The number of bits in a symbol in the compressed-data operand is specified in general register 0. The operation proceeds until the end of either operand is reached or a CPU-determined amount of data has been processed, whichever occurs first. The results are indicated in the condition code.

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 containing any bit of the first operand and second operand is designated by an address in general registers R_1 and R_2, respectively. The number of bytes containing any bits of the first operand and second operand is specified by bits 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, in the 24-bit or 31-bit addressing mode or by bits 0-63 of the registers in the 64-bit addressing mode. The contents of general registers $R_1 + 1$ and $R_2 + 1$ are treated as 32-bit unsigned binary integers in the 24-bit or 31-bit addressing mode or as 64-bit unsigned binary integers in the 64-bit addressing mode.

The location of the leftmost byte of the compression dictionary during compression, or of the expansion dictionary during expansion, is designated on a 4K-byte boundary by an address in general register 1.

The handling of the addresses in general registers R_1, R_2, and 1 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of 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 registers R_1 and R_2 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 of registers R_1 and R_2 constitute the address. In the 24-bit addressing mode, the contents of bit positions 40-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 0-39 and 52-63 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 0-32 and 52-63 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 52-63 are ignored.

Although the contents of bit positions 52-63 of general register 1 are ignored as just described, those contents are used as follows. The contents of bit positions 61-63 of the register are the compressed-data bit number (CBN). At the beginning of the operation, the CBN designates the leftmost bit within the leftmost byte of the compressed-data operand. The compressed-data operand is the first operand during compression, or it is the second operand during expansion. When the symbol-translation option is specified during compression, the contents of bit positions 52-60 of the register, with seven rightmost zeros appended, are the byte offset from the beginning of the compression dictionary to the leftmost byte of the symbol-translation table. Symbol translation cannot be specified during expansion, and the contents of bit positions 52-60 are ignored during expansion.

The contents of the registers just described and also of general register 0 are shown in Figure 7-37 on page 7-74.

Bit 55 (E) of general register 0 specifies the compression operation if zero or the expansion operation if one.

Bit 47 (ST) of general register 0 is the symbol-translation-option bit. During compression when bit 47 is zero, the operation produces indexes, called index symbols, to compression-dictionary entries that represent character strings, and the operation then places the index symbols in the compressed-data operand. During compression when bit 47 is one, the operation still produces index symbols but then translates the index symbols to other symbols, called interchange symbols, that it then places in the compressed-data operand. This symbol translation is done by using the symbol-translation table specified by the address and offset in general register 1. Bit 47

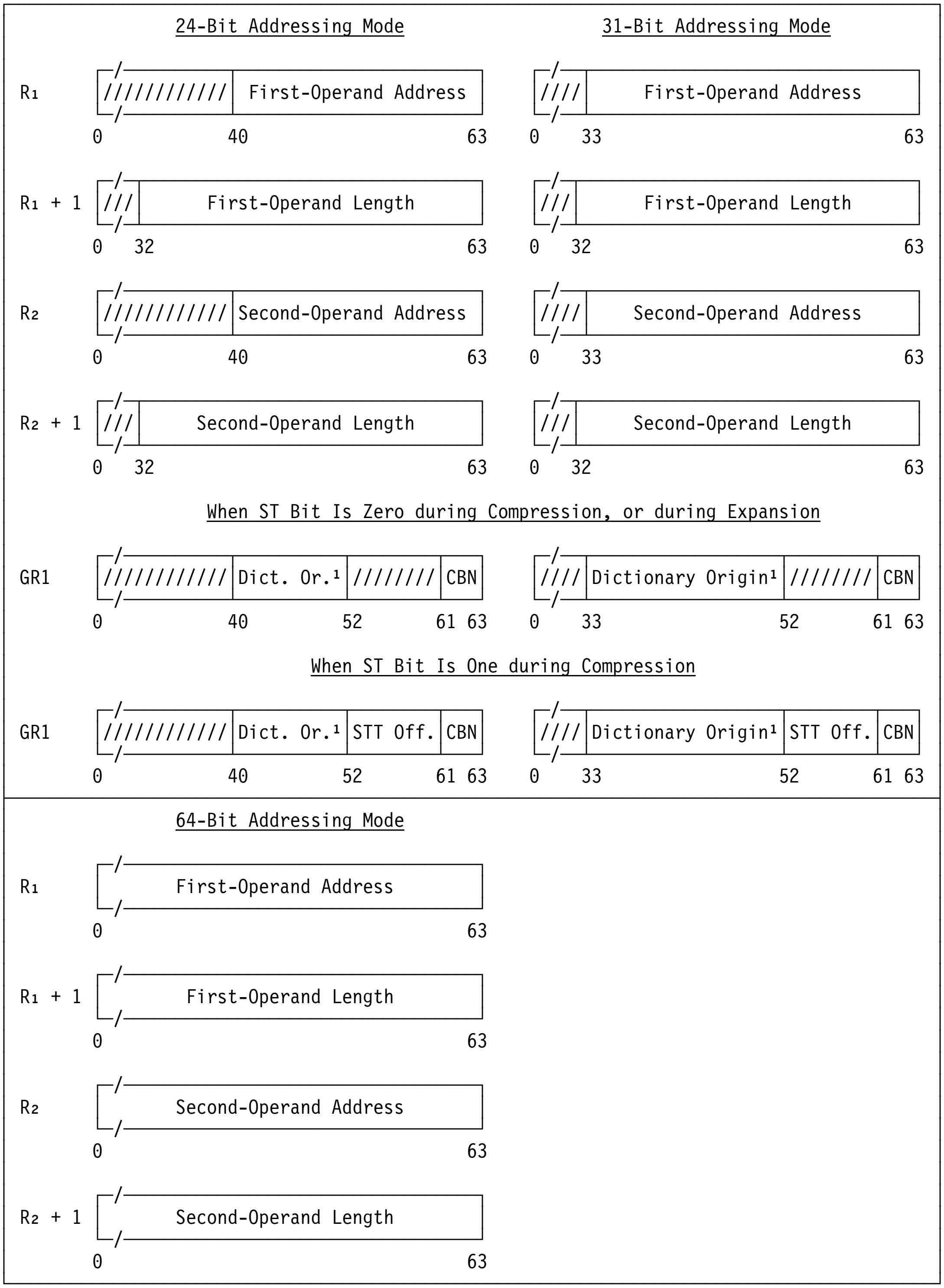


Figure 7-37 (Part 1 of 2). Register Contents for COMPRESSION CALL

and the offset in general register 1 are ignored during expansion. During expansion, the compressed-data operand always contains index symbols that designate entries in the expansion dictionary.

Bits 48-51 (CDSS) of general register 0 specify the number of bits in the index symbols or interchange symbols in the compressed-data operand, as shown in the figure. Bits 48-51 must not have any of the values 0000 or 0110-1111 binary; otherwise, a specification exception is recognized. When symbol-translation is not specified, bits

```
          64-Bit Addressing Mode (Continued)

        When ST Bit Is Zero during Compression,
        or during Expansion

        ┌─/──────────────────────┬────────┬───┐
GR1     │  Dictionary Origin¹    │////////│CBN│
        └─/──────────────────────┴────────┴───┘
        0                        52       61 63

        When ST Bit Is One during Compression

        ┌─/──────────────────────┬────────┬───┐
GR1     │  Dictionary Origin¹    │STT Off.│CBN│
        └─/──────────────────────┴────────┴───┘
        0                        52       61 63

        ┌─/────────────┬─┬────┬──┬─┬─┬────────┐
        │              │S│    │  │F│ │        │
GR0     │//////////////│T│CDSS│//│1│E│////////│
        └─/────────────┴─┴────┴──┴─┴─┴────────┘
        0             47     52 54  56       63
```

Explanation:

¹ Compression dictionary during compression, or expansion dictionary during expansion

CBN Compressed-data bit number

CDSS Compressed-data symbol size, and dictionary size when symbol translation not specified

CDSS (binary)	Symbol Size	Dictionary Size
0000	Causes a specification exception to be recognized	
0001	9 bits	512 entries, 4K bytes
0010	10 bits	1K entries, 8K bytes
0011	11 bits	2K entries, 16K bytes
0100	12 bits	4K entries, 32K bytes
0101	13 bits	8K entries, 64K bytes
0110-1111	Causes a specification exception to be recognized	

E Expansion operation

F1 Format-1 sibling descriptors (ignored during expansion)

ST Symbol-translation option (ignored during expansion)

STT Off. Symbol-translation-table offset (ignored during expansion)

Figure 7-37 (Part 2 of 2). Register Contents for COMPRESSION CALL

48-51 also specify, as shown in the figure, the number of eight-byte entries in each of the compression and expansion dictionaries, and, thus, they specify the size in bytes of each of the dictionaries. When symbol translation is specified, the compression dictionary is considered to extend to the beginning of the symbol-translation table, that is, the size in bytes of the compression dictionary is the offset in bit positions 52-60 of general register 1, with seven rightmost zeros appended. The size in bytes of the symbol-translation table is considered to be one fourth that of the compression dictionary. However, the offset in general register 1 must be at least as large as the size of the compression dictionary would be if symbol translation were not specified and the CDSS were one less than it actually is, and, when the CDSS is 0001 binary, the offset must be at least 2K bytes; otherwise, the results are unpredictable. For example, if the CDSS is 0101, the offset must be at least 32K bytes.

Bit 54 (F1) of general register 0 specifies that the compression dictionary contains format-0 sibling descriptors if the bit is zero or format-1 sibling descriptors if the bit is one. Sibling descriptors are used during the compression operation. A format-0 sibling descriptor is eight bytes at an index position in the compression dictionary. A format-1 sibling descriptor is 16 bytes, with the first eight bytes at an index position in the compression dictionary and the second eight bytes at the same index position in the expansion dictionary. During compression when bit 54 is one, an expansion dictionary is considered to immediately follow the compression dictionary specified by the address in general register 1. Bit 54 is ignored during expansion.

Bits 47 and 54 of general register 0 must not both be ones; otherwise, the results are unpredictable.

The unused bit positions in general register 0 are reserved for possible future extensions and should contain zeros; otherwise, the program may not operate compatibly in the future.

In the access-register mode, the contents of access register R_1 are used for accessing the first operand, and the contents of access register R_2 are used for accessing the second operand and the dictionaries and the symbol-translation table.

The operation starts at the left end of both operands and proceeds to the right. The operation is ended when the end of either operand is reached or when a CPU-determined amount of data has been processed, whichever occurs first.

During a compression operation, the end of the first operand is considered to be reached when the number of unused bit positions remaining in the first-operand location is not sufficient to contain additional compressed data.

During an expansion operation, the end of the first-operand location is considered to be reached when either of the following two conditions is met:

1. The number of unused byte positions remaining in the first-operand location is not sufficient to contain all the characters that would result from expansion of the next index symbol.
2. Immediately when the number of unused byte positions is zero, that is, immediately when the expansion of an index symbol completely fills the first-operand location.

During an expansion operation, the end of the second-operand location is considered to be reached when the next index symbol does not reside entirely within the second-operand location. The second-operand location ends at the beginning of the byte designated by the sum of the address in general register R_2 and the length in general register $R_2 + 1$, regardless of the compressed-data bit number in bit positions 61-63 of general register 1.

If the operation is ended because the end of the second operand is reached, condition code 0 is set. If the operation is ended because the end of the first operand is reached, condition code 1 is set, except that condition code 0 is set if the end of the second operand is also reached. If the operation is ended because a CPU-determined amount of data has been processed, condition code 3 is set.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of complete bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. During compression, a complete byte is considered to be stored only if all of its bit positions contain bits of compressed data. During compression when the first bit of compressed data stored is not in bit position 0 of a byte, the bits in the byte to the left of the first bit of compressed data remain unchanged. During compression if the last byte stored does not completely contain compressed data, the bits in the byte to the right of the rightmost bit of compressed data in the byte either are unchanged or are set to zeros.

The length in general register $R_2 + 1$ is decremented by the number of complete bytes processed at the second-operand location, and the address in general register R_2 is incremented by the same amount. During expansion, a complete byte is considered to be processed only if all of its bits have been used to produce expanded data.

The leftmost bits which are not part of the address in general registers R_1 and R_2 may be set to zeros or may remain unchanged. However, in the 24-bit or 31-bit addressing mode, bits 0-31 of these registers and also of general registers $R_1 + 1$ and $R_2 + 1$ always remain unchanged.

The bit number of the bit following the last bit of compressed data processed is placed in bit positions 61-63 of general register 1, and bits 52-60 of the register and the leftmost bits which are not part of the address in the register may be set to zeros or may remain unchanged, except that when one or both of the original length values are so small that no compressed data can be processed, all bits in the register may remain unchanged. However, when bit 47 of general register 0 is one, bits 52-60 of general register 1 always remain unchanged. Also, in the 24-bit or 31-bit addressing mode, bits 0-31 of the register always remain unchanged.

If the operands overlap one another or the first operand overlaps the dictionaries or the symbol-translation table in storage in any way, the results are unpredictable.

When symbol translation is specified, the symbol-translation table consists of two-byte entries, and an entry contains an interchange symbol in the rightmost bit positions of the entry. The length of the interchange symbol is specified by bits 48-51 of general register 0. The left-hand bits that are not part of the interchange symbol in a symbol-translation-table entry must be zeros; otherwise, the results are unpredictable.

To translate an index symbol to an interchange symbol, the index symbol is multiplied by 2 and then added to the address of the beginning of the symbol-translation table to locate an entry in the table, and then the interchange symbol is obtained from the entry.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the contents of the registers designated by the R_1 and R_2 fields and of general register 1 are updated the same as upon normal completion of the instruction, so that the instruction, when reexecuted, resumes at the point of interruption. The condition code is unpredictable.

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 may be recognized for all locations in the dictionaries and symbol-translation table if those areas are specified to be used and even if the locations would not be used during the operation. Access exceptions are not recognized for an operand, the dictionaries, or the symbol-translation table 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.

If an access exception is due to be recognized for either of the operands or for a dictionary or the symbol-translation table, the result is that either the exception is recognized or condition code 3 is set. If condition code 3 is set, the exception will be recognized when the instruction is executed again to continue processing the same operands, assuming that the exception condition still exists.

During compression, regardless of whether the exception is recognized or condition code 3 is set, a nullifying access-exception condition or a suppressing page-protection exception condition is handled so that an index symbol is generated only if it is the one that would result if there were no access-exception condition.

During compression or expansion, regardless of whether the exception is recognized or condition code 3 is set, a nullifying or suppressing access-exception condition may result in data having been stored at the first-operand location at or to the right of the location designated by the final address in general register R_1, which result is not true nullification or suppression. The amount of data stored depends on the reason for the access-exception condition. If the condition is due to a reference to a dictionary or the symbol-translation table, up to 4K bytes of data may have been stored at or to the right of the location designated by the final address. If the condition is due to a reference to the first or second operand, part of one index or interchange symbol, during compression, or part of one character symbol, during expansion, may have been stored at or to the right of the location designated by the final address. In all cases, the storing will be repeated when the instruction is executed again to continue processing the same operands.

If the end of the first operand is reached and an access exception is due to be recognized for the second operand, it is unpredictable whether condition code 1 is set or the access exception is recognized.

During expansion when the expansion dictionary is not logically correct, unusual storing may occur as described in the section “Expansion Process” in Chapter 1 of *Enterprise Systems Architecture/390 Data Compression,* SA22-7208-01. The results of an access exception in this case may not be true nullification or suppression.

Special Conditions

During compression of each character symbol, either the characters in the symbol or the dictionary character entries (not sibling descriptors) representing characters of the symbol are counted, and a data exception is recognized if this count becomes too large. The count can reach at least 260 without the exception being recognized.

During compression, the number of child characters or sibling characters processed during the processing of each parent entry are counted, and a data exception is recognized if this count becomes too large. The count can reach at least 260 without the exception being recognized. That is, a parent must not have more than 260 children; otherwise, a data exception may be recognized.

During expansion of each character symbol, either the characters in the symbol or the dictionary entries representing characters of the symbol are counted, and a data exception is recognized if this count becomes too large. If the characters in the symbol are counted, the count can reach at least 260 without the exception being recognized. If the dictionary entries representing characters of the symbol are counted, the count can reach at least 127 without the exception being recognized.

Certain error conditions in the dictionaries cause a data exception to be recognized and the operation to be terminated. Some of these error conditions are described in the sections “Expansion Process” and “Results of Dictionary Errors” in Chapter 1 of SA22-7208-01. The others are described in Chapter 2 of SA22-7208-01.

Resulting Condition Code:

0 End of second operand reached
1 End of first operand reached and end of second operand not reached
2 --
3 CPU-determined amount of data processed

Program Exceptions:

- Access (fetch, operand 2, dictionaries, and symbol-translation table; store, operand 1)
- Data
- Specification

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the operation. The program need not determine the amount of data processed.

2. During compression when a nullifying access exception is due to be recognized, an index symbol is generated only if it is the one that would result if there were no access-exception condition. The result of this is that compression of the same expanded data by means of one or more executions of the instruction and by using the same dictionary always results in the same compressed data. That is, (1) the best possible matches in the the dictionary are always found for the characters in the second operand, or else the execution is ended by either setting CC3 or recognizing the exception, and (2) the results of compression are repeatable (although possibly by means of a different number of executions of the instruction) and predictable.

   For example, if the next characters of the string being compressed are ABC, and dictionary entry A has a child B, which has a child C, the normal operation is to compress ABC as a single index symbol, but if the C of the string is in the first byte of an invalid page (a page-translation exception is due to be recognized), no index symbol is generated. More specifically, an index symbol corresponding to the character symbol AB is not generated because this is not the index symbol that would be generated if the access-exception condition did not exist.

   In the above, “best possible match” refers only to the characters in the second operand. In the example above, if the next two characters of the second operand (the string) are AB, and these are the last two characters of the second operand, the best possible match is on AB, even though there could be a match on ABC if the second operand included one more byte containing C.

   Expansion is normally always repeatable. An index symbol is always expanded to exactly the character symbol it represents unless an exception that causes termination is recognized.

3. During expansion, if at least one unused byte position remains in the first operand location,

COMPRESSION CALL may completely process the next index symbol in the second operand before it determines that the first-operand location does not have sufficient unused byte positions to contain the expanded data that would result from the next index symbol. If that next index symbol causes encountering of bad dictionary entries, the result can be either a data exception or condition code 1.

COMPRESSION CALL immediately sets condition code 1 when processing of an index symbol exactly fills the first-operand location, except that it sets condition code 0 if the end of the second-operand location also has been reached. Immediately setting condition code 1 has the advantage that data can be compressed using one dictionary and then followed immediately, possibly on a bit boundary, by a different type of data compressed using another dictionary. The compressed data can be successfully expanded if, during the expansion of the data compressed using the first dictionary, the length of the first-operand location is specified to be exactly the length of the expanded data that will be produced. Condition code 1 will then be set when the first-operand location is full, at which time the specification of the dictionary can be changed in order to expand the remainder of the compressed data using the second dictionary. If the definition allowed condition code 1 not to be set, it might be attempted to expand the next index symbol, which resulted from use of the second dictionary, by means of the first dictionary, and this might cause recognition of a data exception. For example, the next index symbol, which properly designates a character entry in the second dictionary, might designate the second half of a format-1 sibling descriptor in the first dictionary, and that second half might begin with a character, such as 0 (F0 hex), that would appear to be an invalid partial symbol length in a character entry.

4. A nullifying access-exception condition due to a reference to a dictionary or the symbol-translation table may result in the storing of data at or to the right of the location designated by the final address in general register R_1. This storing and the processing needed to produce the data stored will be repeated when COMPRESSION CALL is executed again to continue processing the same operands. The repeated processing will reduce the performance of the instruction execution, and it should be avoided by ensuring that the environment in which the program is executed is one in which page-translation-exception conditions for the dictionaries and symbol-translation table are infrequent.

5. Following is an example of how the compressed-data bit number (CBN) is used and set. In this example:

   - The operation is an expansion operation.
   - The CDSS in general register 0 is 0010 binary. Therefore, there are 1K entries in the expansion dictionary, and the length of an index symbol is 10 bits.
   - The second operand (compressed-data operand) begins at location 6000 hex and has a length of five bytes. The initial CBN is 7. Therefore, there are three index symbols to be expanded, and the final CBN will be 5.
   - The compressed data beginning at location 6000 hex is 0081FF9FF8 hex. Therefore, the three index symbols are 103, 3FC, and 3FF hex.
   - The first operand (expanded-data operand) begins at location 5000 hex and has a length of 64 bytes. The three index symbols are expanded to a total of 14 bytes of expanded data.

   The following figure shows the initial and final contents of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, the contents of locations 6000-6004 hex in binary, and the way a cursor corresponding to the CBN is advanced during the expansion operation.

Register	Initial Contents in Hex	Final Contents in Hex
R_1	5000	500E
R_1+1	40	32
R_2	6000	6004
R_2+1	5	1

Contents of Locations 6000-6004 Hex in Binary

```
00000000 10000001 11111111 10011111 11111000
       ↑           ↑          ↑          ↑
       │           │          │          │
Initial│           │          │   Final  │
CBN (7)┘                          CBN (5)┘
```

6. The reason for allowing a parent to have no more than 260 children is as follows. The parent can contain five identical child characters. Then, 255 different sibling characters are possible — all of these must be different from the child characters and each other, or else they may be wasted (never matched against), depending on the implementation. Thus, every possible child is permitted.

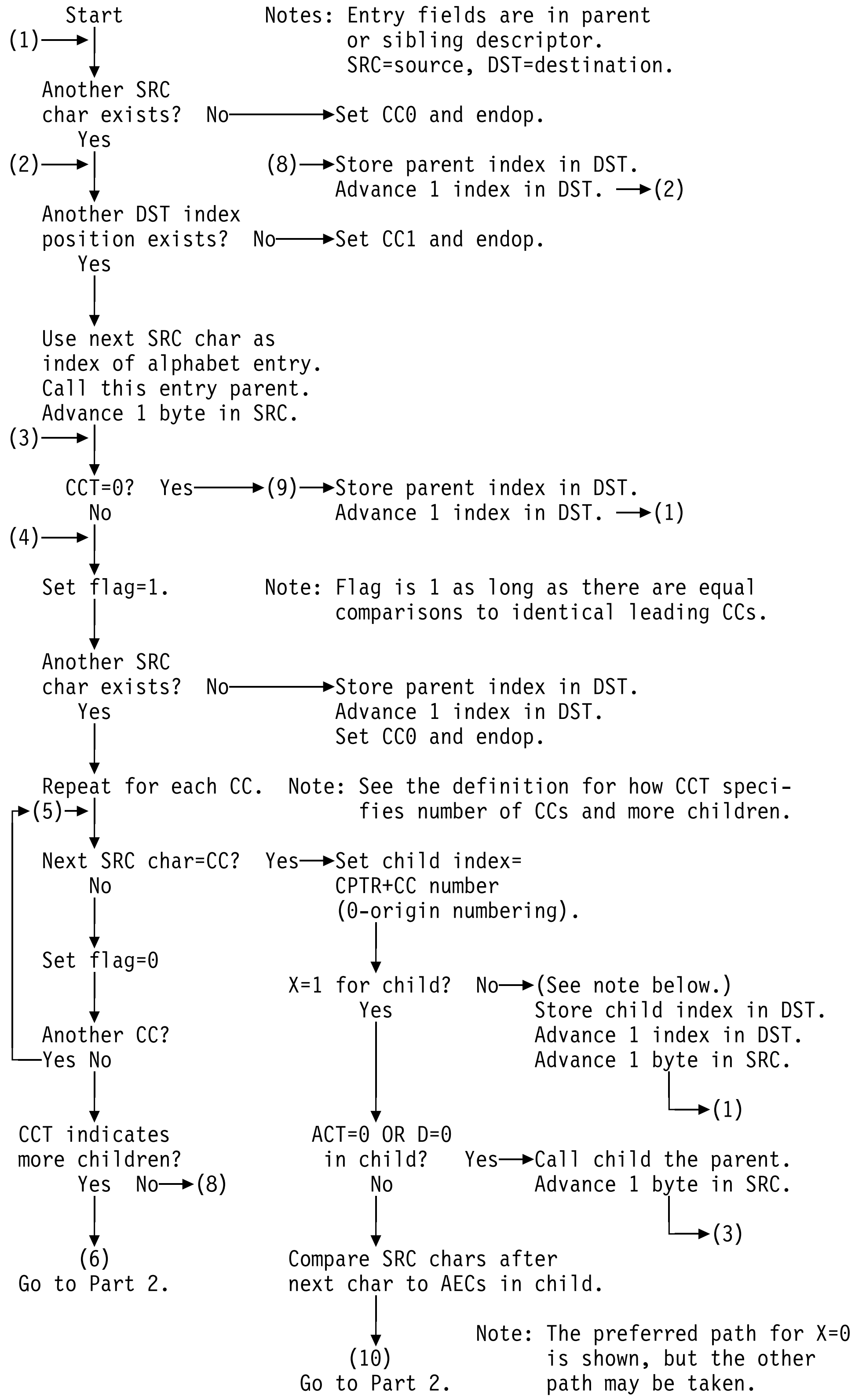
7. Symbol translation is for use by VTAM. VTAM will begin by doing compression by means of software and an adaptive dictionary. When the adaptive dictionary has matured such that the degree of compression becomes sufficiently good (crosses some threshold), VTAM will “freeze” (stop adapting) its dictionary, inform the other end of the session to freeze also, transform its adaptive dictionary to the dictionary form used by COMPRESSION CALL, and then use COMPRESSION CALL to continue on with the compression. The other end of the session can continue to use its frozen adaptive dictionary.

   Following is clarification about the STT offset. Assume VTAM uses a 4K-entry adaptive dictionary, which is the largest size VTAM uses. All of the entries in this dictionary correspond to character symbols because there are no sibling descriptors in the VTAM dictionary. The VTAM dictionary cannot map one-to-one to a COMPRESSION CALL dictionary because the latter requires that some of the entries be sibling descriptors. Therefore, VTAM must have an 8K-entry dictionary for use in the basic compression operation. Only the first hundred or so entries in the second 4K of the 8K need to be used, and these entries compensate for (take the place of) the entries in the first 4K that must be sibling descriptors. The STT can and should, to save space, begin immediately after those hundred or so entries in the second 4K. In this example, the index symbols will be 13 bits but will be transformed to 12-bit interchange symbols.

8. A program may place the dictionaries in pages that are managed by means of chaining fields at their beginnings. In this case, either the parts of a dictionary have to be moved to be compacted into contiguous locations or there have to be holes in the dictionaries. The definition of COMPRESSION CALL contains nothing explicitly to support holes. However, assuming there is at least one character that never appears in the expanded data, that character can be used as a child character in a parent entry or as a sibling character in a sibling descriptor to specify a child or children that will never be referenced, thus creating a hole.

9. The references to the operands, dictionaries, and symbol-translation table for COMPRESSION CALL may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

10. Figure 7-38 on page 7-81 and Figure 7-39 on page 7-83 show possible forms (not the only possible forms) of the compression and expansion processes. The figures do not show testing for or the results of dictionary errors.



```
     Start           Notes: Entry fields are in parent
(1)───►|                    or sibling descriptor.
       ▼                    SRC=source, DST=destination.
   Another SRC
   char exists?  No────────►Set CC0 and endop.
      Yes
(2)───►|              (8)──►Store parent index in DST.
       ▼                    Advance 1 index in DST. ──►(2)
   Another DST index
   position exists?  No────►Set CC1 and endop.
      Yes
       |
       ▼

   Use next SRC char as
   index of alphabet entry.
   Call this entry parent.
   Advance 1 byte in SRC.
(3)───►|
       ▼
     CCT=0?  Yes─────►(9)──►Store parent index in DST.
       No                   Advance 1 index in DST. ──►(1)
(4)───►|
       ▼
   Set flag=1.        Note: Flag is 1 as long as there are equal
       |                    comparisons to identical leading CCs.
       ▼
   Another SRC
   char exists?  No────────►Store parent index in DST.
      Yes                   Advance 1 index in DST.
       |                    Set CC0 and endop.
       ▼
   Repeat for each CC.  Note: See the definition for how CCT speci-
┌►(5)─►|                      fies number of CCs and more children.
│      ▼
│  Next SRC char=CC?  Yes──►Set child index=
│      No                   CPTR+CC number
│      |                    (0-origin numbering).
│      ▼                           |
│  Set flag=0                      ▼
│      |                X=1 for child?  No──►(See note below.)
│      ▼                      Yes            Store child index in DST.
│  Another CC?                 |             Advance 1 index in DST.
└──Yes No                      |             Advance 1 byte in SRC.
       |                       |                        |
       ▼                       ▼                        └──►(1)
 CCT indicates            ACT=0 OR D=0
 more children?            in child?   Yes──►Call child the parent.
      Yes  No──►(8)            No            Advance 1 byte in SRC.
       |                       |                        |
       ▼                       ▼                        └──►(3)
      (6)               Compare SRC chars after
 Go to Part 2.          next char to AECs in child.
                               |
                               ▼        Note: The preferred path for X=0
                             (10)             is shown, but the other
                         Go to Part 2.        path may be taken.
```

Figure 7-38 (Part 1 of 2). Compression Process

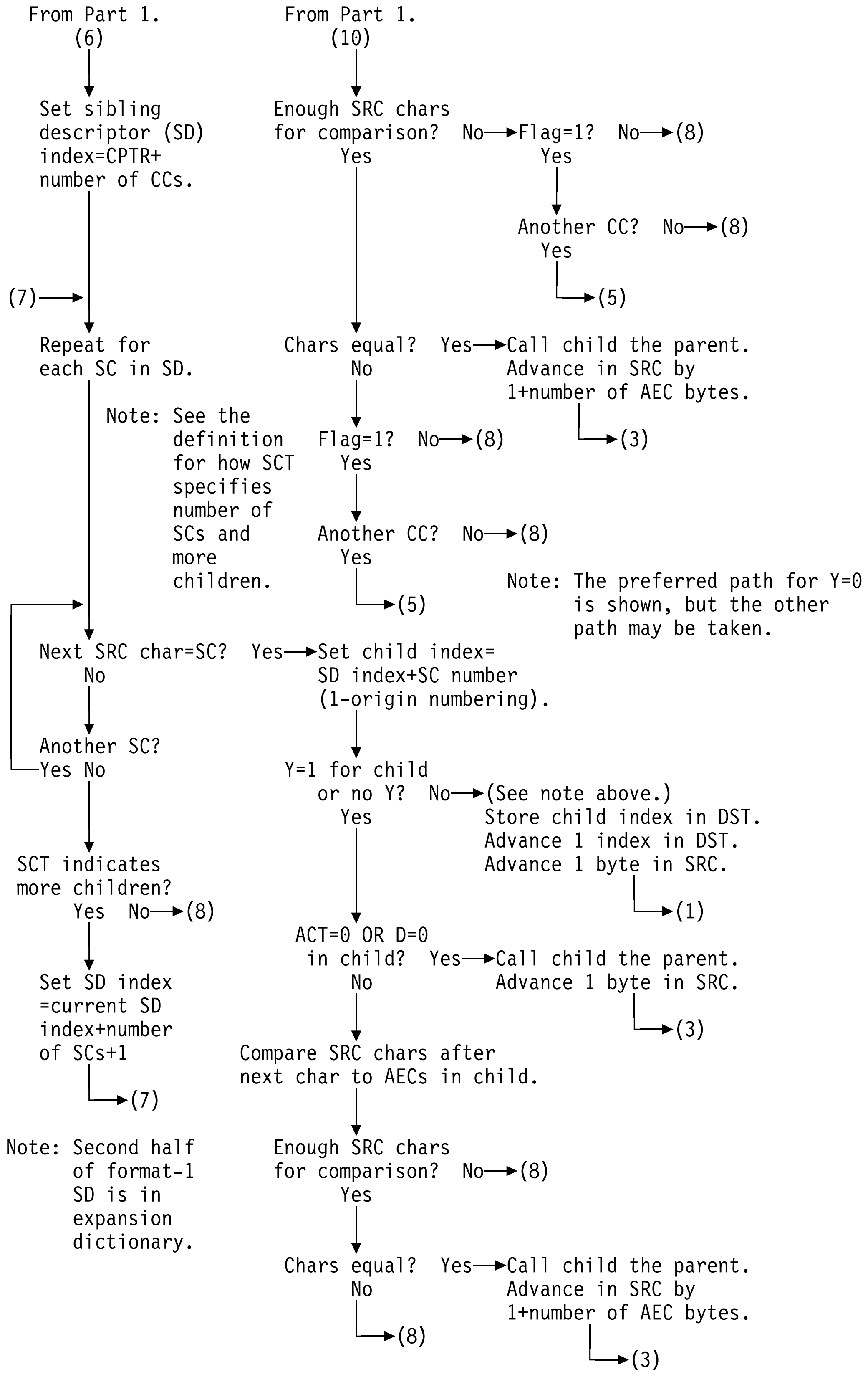

From Part 1.
(6)
Set sibling descriptor (SD) index=CPTR+ number of CCs.
(7)
Repeat for each SC in SD.
Note: See the definition for how SCT specifies number of SCs and more children.
Next SRC char=SC?
Yes
No
Another SC?
Yes No
SCT indicates more children?
Yes No
(8)
Set SD index =current SD index+number of SCs+1
(7)
Note: Second half of format-1 SD is in expansion dictionary.
From Part 1.
(10)
Enough SRC chars for comparison?
No
Yes
Flag=1?
No
(8)
Yes
Another CC?
No
(8)
Yes
(5)
Chars equal?
Yes
Call child the parent. Advance in SRC by 1+number of AEC bytes.
(3)
No
Flag=1?
No
(8)
Yes
Another CC?
No
(8)
Yes
(5)
Note: The preferred path for Y=0 is shown, but the other path may be taken.
Set child index= SD index+SC number (1-origin numbering).
Y=1 for child or no Y?
No
(See note above.) Store child index in DST. Advance 1 index in DST. Advance 1 byte in SRC.
(1)
Yes
ACT=0 OR D=0 in child?
Yes
Call child the parent. Advance 1 byte in SRC.
(3)
No
Compare SRC chars after next char to AECs in child.
Enough SRC chars for comparison?
No
(8)
Yes
Chars equal?
Yes
Call child the parent. Advance in SRC by 1+number of AEC bytes.
(3)
No
(8)

Figure 7-38 (Part 2 of 2). Compression Process

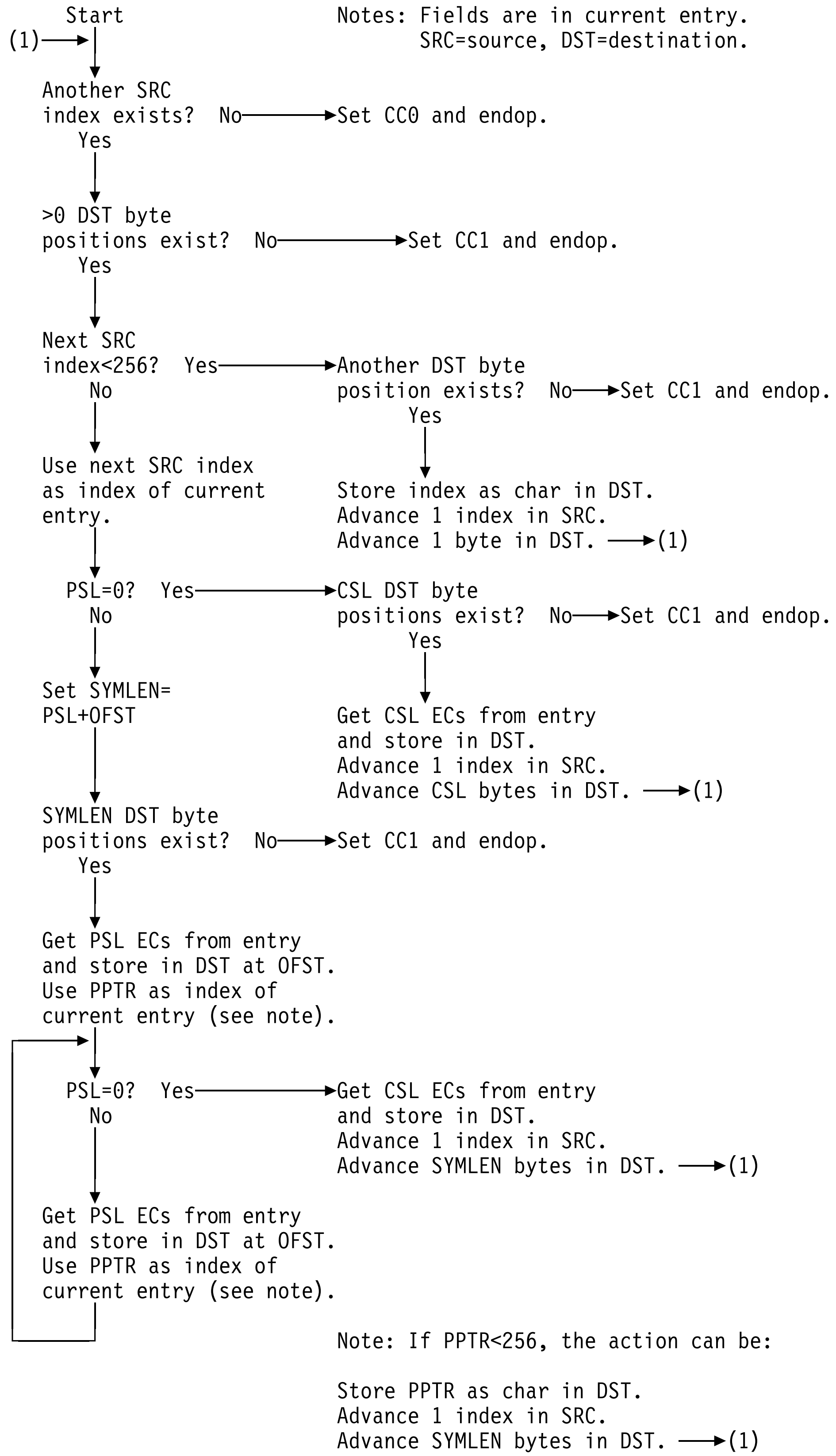


Figure 7-39. Expansion Process

COMPUTE INTERMEDIATE MESSAGE DIGEST (KIMD)

```
KIMD     R1,R2        [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B93E'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0               16       24   28   31
```

COMPUTE LAST MESSAGE DIGEST (KLMD)

```
KLMD     R1,R2        [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B93F'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0               16       24   28   31
```

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction and the R_1 field are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figures 7-40 and 7-41 show the assigned function codes for COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST, respectively. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 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 general register 1 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 of general register 1 constitute the address.

The function codes for COMPUTE INTERMEDIATE MESSAGE DIGEST are as follows.

Figure 7-40. Function Codes for COMPUTE INTERMEDIATE MESSAGE DIGEST

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KIMD-Query	16	—
1	KIMD-SHA-1	20	64

Explanation:

— Not applicable

The function codes for COMPUTE LAST MESSAGE DIGEST are as follows.

Figure 7-41. Function Codes for COMPUTE LAST MESSAGE DIGEST

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KLMD-Query	16	—
1	KLMD-SHA-1	28	64

Explanation:

— Not applicable

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_2 and $R_2 + 1$ are ignored for the query function.

For all other functions, the second operand is processed as specified by the function code using an initial chaining value in the parameter block, and the result replaces the chaining value. For COMPUTE LAST MESSAGE DIGEST, the operation also uses a message bit length in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R_2 field designates 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 second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified in general register $R_2 + 1$.

As part of the operation, the address in general register R_2 is incremented by the number of bytes processed from the second operand, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_2 constitute the address of second operand, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general register R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register R_2 constitute the address of second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R_2 constitute the address of second operand; bits 0-63 of the updated address replace the contents of general register R_2 and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register $R_2 + 1$.

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

Figure 7-42 on page 7-86 shows the contents of the general registers just described.

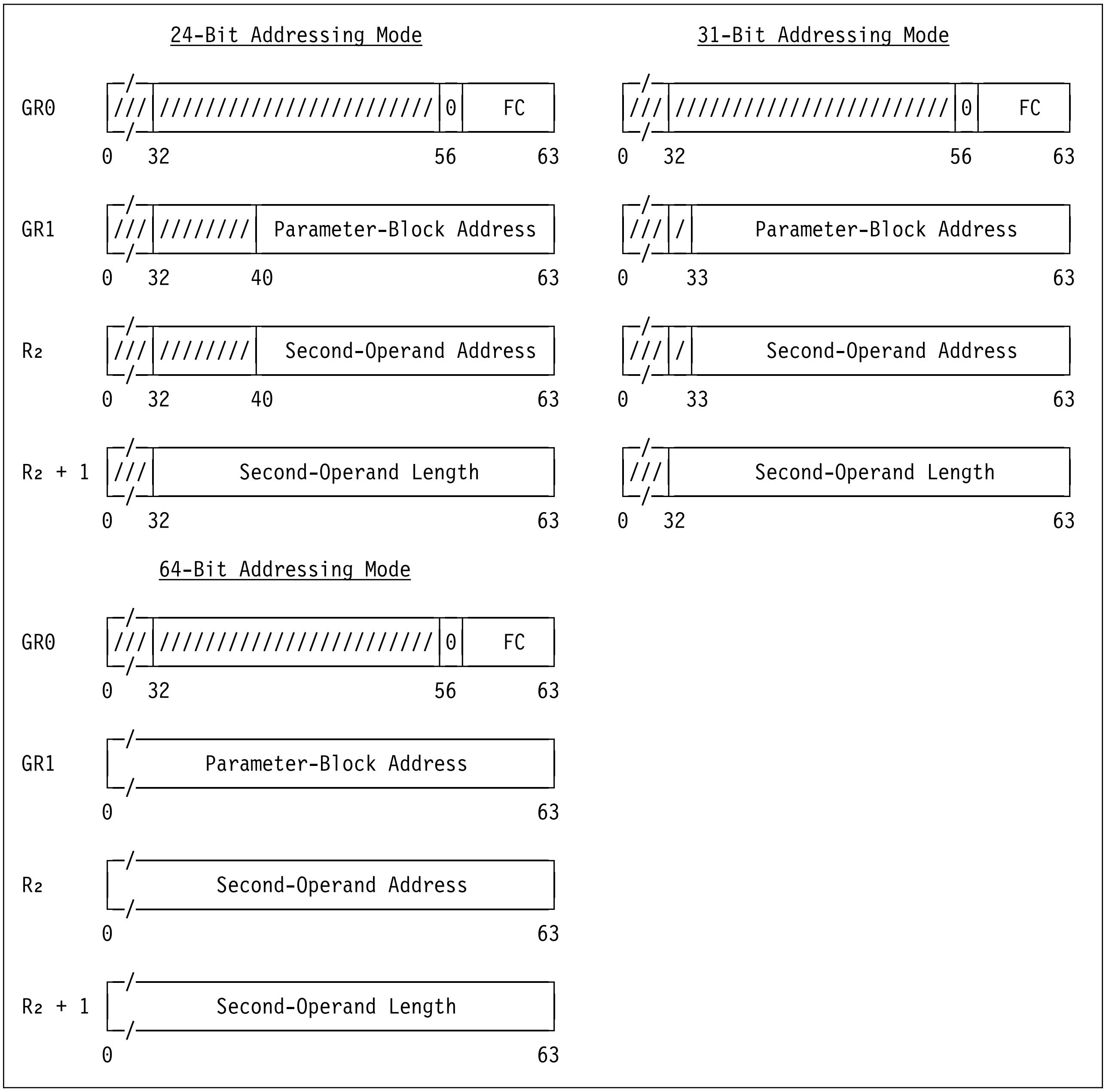


Figure 7-42. General Register Assignment for KIMD and KLMD

In the access-register mode, access registers 1 and R$_2$ specify the address spaces containing the parameter block and second operand, respectively.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. The operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the chaining-value field overlaps any portion of the second operand, the result in the chaining-value field is unpredictable.

For COMPUTE INTERMEDIATE MESSAGE DIGEST, normal completion occurs when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed. For COMPUTE LAST MESSAGE DIGEST, after all bytes in the second operand as specified in general register $R_2 + 1$ have been processed, the padding operation is performed, and then normal completion occurs.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

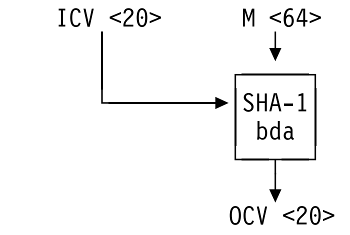
When the second-operand length is initially zero, the second operand is not accessed, general registers R_2 and $R_2 + 1$ are not changed, and condition code 0 is set. For COMPUTE INTERMEDIATE MESSAGE DIGEST, the parameter block is not accessed. However, for COMPUTE LAST MESSAGE DIGEST, the empty block (L = 0) case padding operation is performed and the result is stored into the parameter block.

As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

Access exceptions may be reported for a larger portion of the second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the second operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST functions. Further description of the secure hash algorithm may be found in *Secure Hash Standard*, Federal Information Processing Standards publication 180-1, National Institute of Standards and Technology, Washington DC, April 17, 1995.



Symbol	Explanation
<n>	Length of item in bytes
ICV	Initial chaining value
M	Message block
OCV	Output chaining value

Figure 7-43. Symbol for SHA-1 Block Digest Algorithm

KIMD-Query (KIMD Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-42 on page 7-86.

The parameter block used for the function has the following format:

```
     ┌──────────────────────┐
  0  │                      │
     ├     Status Word      ┤
  8  │                      │
     └──────────────────────┘
     0                     63
```

Figure 7-44. Parameter Block for KIMD-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KIMD-Query function completes; condition code 3 is not applicable to this function.

KIMD-SHA-1 (KIMD Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-42 on page 7-86.

The parameter block used for the function has the following format:

```
       ┌──────────────┐
   0   │      H0      │
       ├──────────────┤
   4   │      H1      │
       ├──────────────┤
   8   │      H2      │
       ├──────────────┤
  12   │      H3      │
       ├──────────────┤
  16   │      H4      │
       └──────────────┘
       0             31
```

Figure 7-45. Parameter Block for KIMD-SHA-1

A 20-byte intermediate message digest is generated for the the 64-byte message blocks in operand 2 using the SHA-1 block digest algorithm with the 20-byte chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

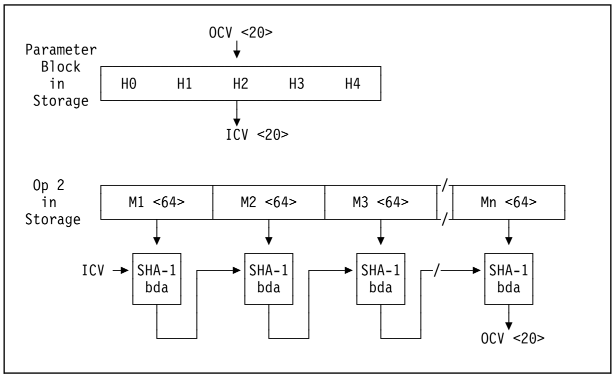


Figure 7-46. KIMD-SHA-1

KLMD-Query (KLMD Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-42 on page 7-86.

The parameter block used for the function has the following format:

```
       ┌──────────────────────────┐
   0   │                          │
       ┤       Status Word        ├
   8   │                          │
       └──────────────────────────┘
       0                         63
```

Figure 7-47. Parameter Block for KLMD-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the COMPUTE LAST MESSAGE DIGEST instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KLMD-Query function completes; condition code 3 is not applicable to this function.

KLMD-SHA-1 (KLMD Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-42 on page 7-86.

The parameter block used for the function has the following format:

```
       ┌──────────────┐
   0   │      H0      │
       ├──────────────┤
   4   │      H1      │
       ├──────────────┤
   8   │      H2      │
       ├──────────────┤
  12   │      H3      │
       ├──────────────┤
  16   │      H4      │
       ├──────────────┤
  20   │   Message    │
       │  Bit Length  │
  24   │    (mbl)     │
       └──────────────┘
       0             31
```

Figure 7-48. Parameter Block for KLMD-SHA-1

The message digest for the message (M) in operand 2 is generated using the SHA-1 algorithm with the chaining value and message-bit-length information in the parameter block.

If the length of the message in operand 2 is equal to or greater than 64 bytes, an intermediate message digest is generated for each 64-byte message block using the SHA-1 block digest algorithm with the 20-byte chaining value in the parameter block, and the generated intermediate message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. This operation is shown in Figure 7-49 on page 7-89 and repeats until the remaining message is less than 64 bytes.

If the length of the message or the remaining message is zero bytes, then the operation in Figure 7-50 on page 7-89 is performed. If the length of the message or the remaining message is between one byte and 55 bytes inclusive, then the operation in Figure 7-51 on page 7-89 is performed; if the length is between 56 bytes and 63 bytes inclusive, then the operation in Figure 7-52 on page 7-90 is performed; The message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block.

Additional Symbols Used in KLMD Functions

The following additional symbols are used in the description of the COMPUTE LAST MESSAGE DIGEST functions.

Symbol	Explanation for KLMD Function Figures
L	Byte length of operand 2 in storage.
p <n>	n padding bytes; leftmost byte is 80 hex; all other bytes are 00 hex.
z <56>	56 padding bytes of zero.
mbl	an 8-byte value specifying the bit length of the total message.
q <64>	a padding block, consisting of 56 bytes of zero followed by an 8-byte mbl.

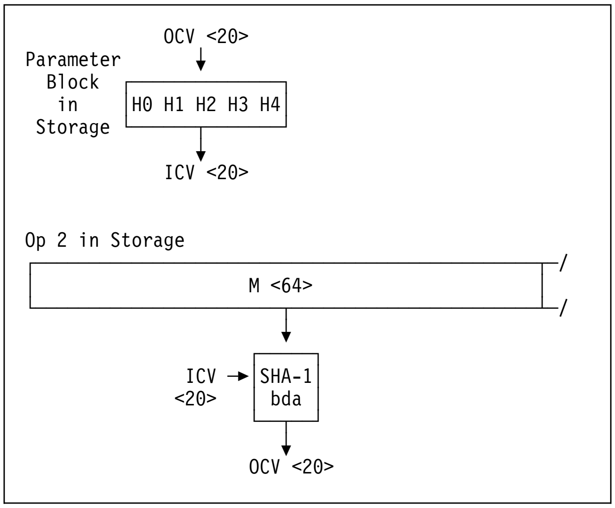


Figure 7-49. KLMD-SHA-1 Full Block (L ≥ 64)

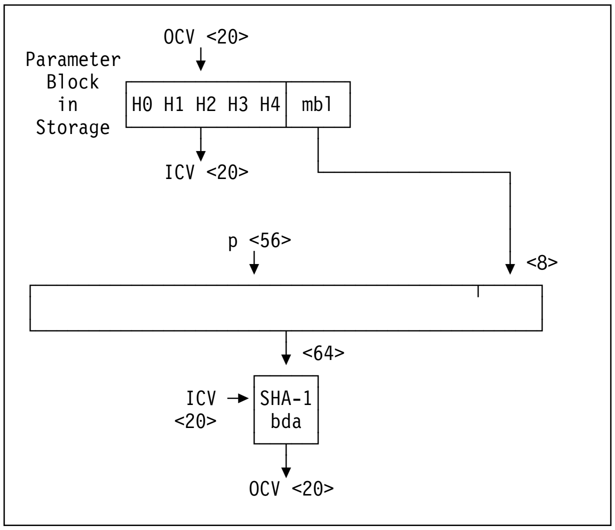


Figure 7-50. KLMD-SHA-1 Empty Block (L = 0)

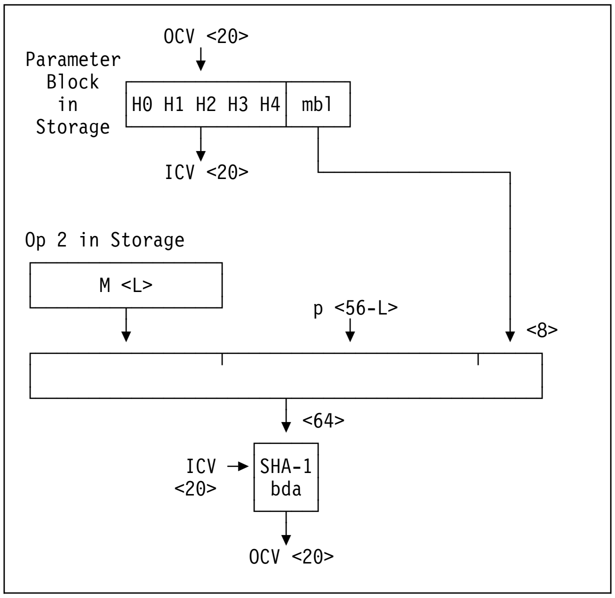


Figure 7-51. KLMD-SHA-1 Partial-Block Case 1 (1 ≤ L ≤ 55)

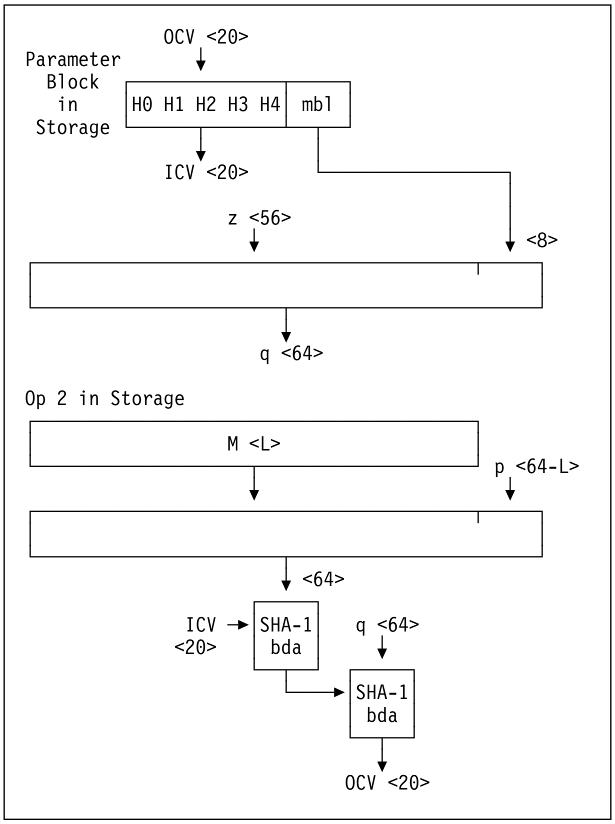


Figure 7-52. KLMD-SHA-1 Partial-Block Case 2 (56 ≤ L ≤ 63)

Special Conditions for KIMD and KLMD

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bit 56 of general register 0 is not zero.
2. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
3. The R_2 field designates an odd-numbered register or general register 0.
4. For COMPUTE INTERMEDIATE MESSAGE DIGEST, the second-operand length is not a multiple of the data block size of the designated function (see Figure 7-40 on page 7-84 to determine the data block sizes for COMPUTE INTERMEDIATE MESSAGE DIGEST functions). This specification-exception condition does not apply to the the query function, nor does it apply to COMPUTE LAST MESSAGE DIGEST.

Resulting Condition Code:

0	Normal completion
1	--
2	--
3	Partial completion

Program Exceptions:

- Access (fetch, operand 2 and message bit length; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- 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 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 7-53. Priority of Execution: KIMD and KLMD

Programming Notes:

1. Bit 56 of general register 0 is reserved for future extension and should be set to zero.

2. When condition code 3 is set, the second operand address and length in general registers R_2 and R_2 + 1, respectively, and the 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 for the no-progress case. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.

3. 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; the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

4. The instructions COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST are designed to be used by a security service application programming interface (API). These APIs provide the program with means to compute the digest of messages of almost unlimited size, including those too large to fit in storage all at once. This is accomplished by permitting the program to pass the message to the API in parts. The following programming notes are described in terms of these APIs.

5. Before processing the first part of a message, the program must set the initial values for the chaining-value field. For SHA-1, the initial chaining values are listed as follows:

   ```
   H0 = x'6745 2301'
   H1 = x'EFCD AB89'
   H2 = x'98BA DCFE'
   H3 = x'1032 5476'
   H4 = x'C3D2 E1F0'
   ```

6. When processing message parts other than the last, the program must process message parts in multiples of 512 bits (64 bytes) and use the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction.

7. When processing the last message part, the program must compute the length of the original message in bits and place this 64-bit value in the message-bit-length field of the parameter block, and use the COMPUTE LAST MESSAGE DIGEST instruction.

8. The COMPUTE LAST MESSAGE DIGEST instruction does not require the second operand to be a multiple of the block size. It first processes complete blocks, and may set condition code 3 before processing all blocks. After processing all complete blocks, it then performs the padding operation including the remaining portion of the second operand. This may require one or two iterations of the SHA-1 block digest algorithm.

9. The COMPUTE LAST MESSAGE DIGEST instruction provides the SHA-1 padding for messages that are a multiple of eight bits in length. If SHA-1 is to be applied to a bit string which is not a multiple of eight bits, the program must perform the padding and use the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction.

COMPUTE MESSAGE AUTHENTICATION CODE (KMAC)

```
KMAC     R1,R2       [RRE]

┌────────────────┬────────┬────┬────┐
│     'B91E'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction and the R_1 field are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-54 shows the assigned function codes. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 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 general register 1 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 of general register 1 constitute the address.

The function codes for COMPUTE MESSAGE AUTHENTICATION CODE are as follows.

Figure 7-54. Function Codes for COMPUTE MESSAGE AUTHENTICATION CODE

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KMAC-Query	16	—
1	KMAC-DEA	16	8
2	KMAC-TDEA-128	24	8
3	KMAC-TDEA-192	32	8

Explanation:

— Not applicable

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_2 and $R_2 + 1$ are ignored.

For all other functions, the second operand is processed as specified by the function code using an initial chaining value in the parameter block and the result replaces the chaining value. The operation also uses a cryptographic key in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R_2 field designates 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 second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified in general register $R_2 + 1$.

As part of the operation, the address in general register R_2 is incremented by the number of bytes processed from the second operand, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_2 constitute the address of second operand, and are ignored; bits 40-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 40 of the updated address are ignored and, the contents of bit positions 32-39 of general register R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register R_2 constitute the address of second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R_2 constitute the address of second operand; bits 0-63 of the updated address replace the contents of general register R_2 and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register $R_2 + 1$.

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

Figure 7-55 on page 7-93 shows the contents of the general registers just described.

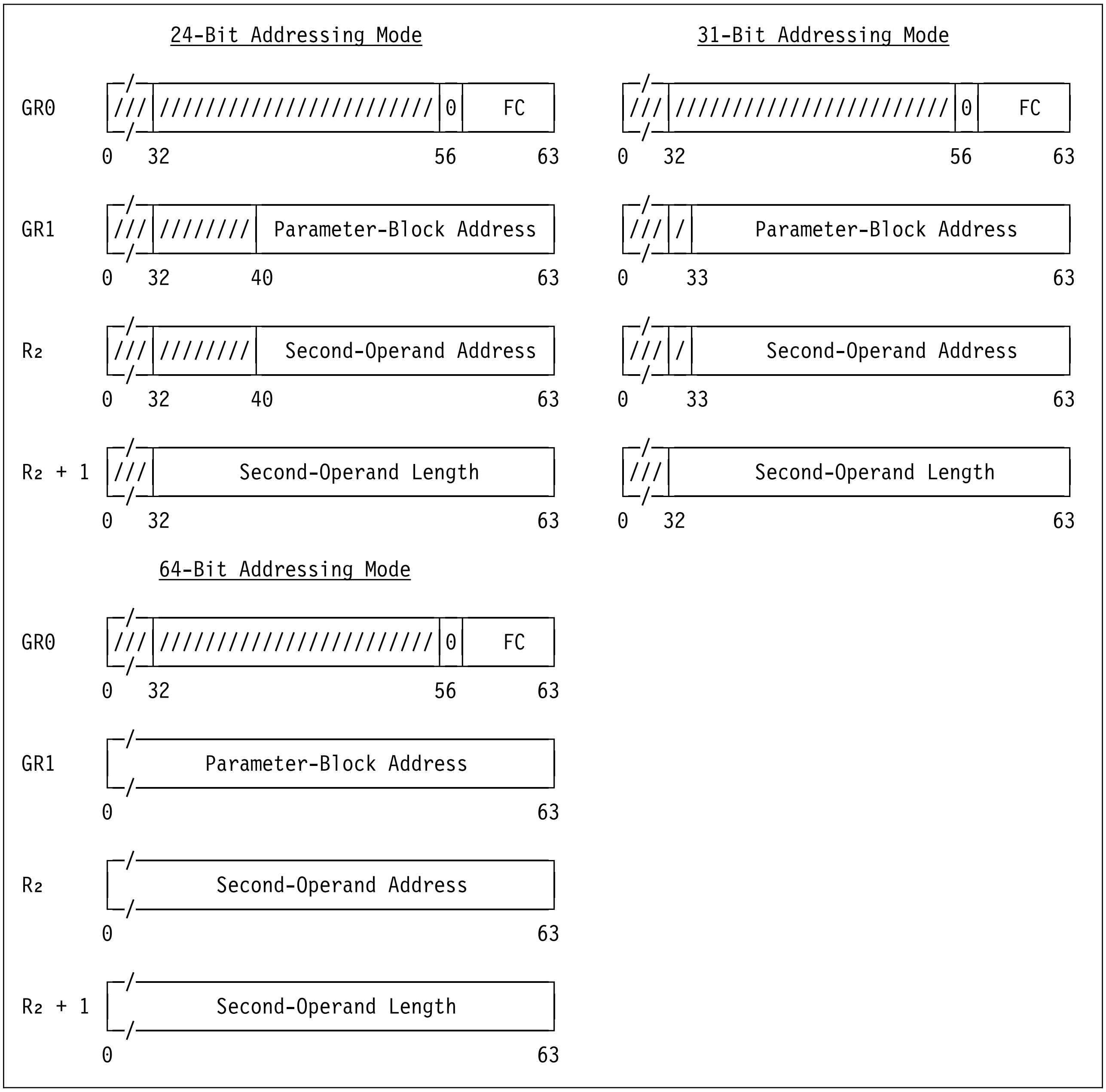


Figure 7-55. General Register Assignment for KMAC

In the access-register mode, access registers 1 and R_2 specify the address spaces containing the parameter block and second operand, respectively.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. The operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the chaining-value field overlaps any portion of the second operand, the result in the chaining-value field is unpredictable.

Normal completion occurs when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

When the second-operand length is initially zero, the second operand and the parameter block are not accessed, general registers R_2 and $R_2 + 1$ are not changed, and condition code 0 is set.

As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

Access exceptions may be reported for a larger portion of the second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the second operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the COMPUTE MESSAGE AUTHENTICATION CODE functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key. Further description of the data-encryption algorithm may be found in *Data Encryption Algorithm*, ANSI-X3.92.1981, American National Standard for Information Systems.

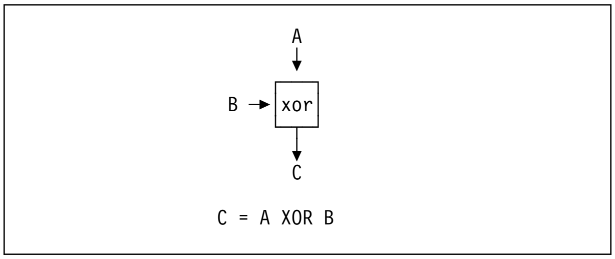
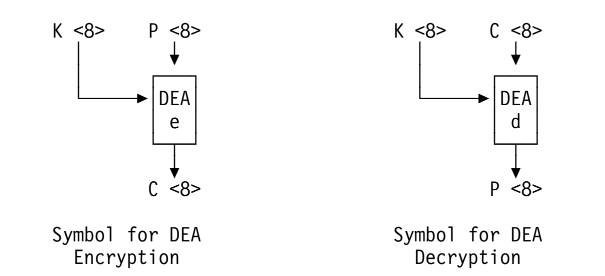


Figure 7-56. Symbol For Bit-Wise Exclusive Or



Symbol	**Explanation**
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-57. Symbols for DEA Encryption and Decryption

KMAC-Query (Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-55 on page 7-93.

The parameter block used for the function has the following format:

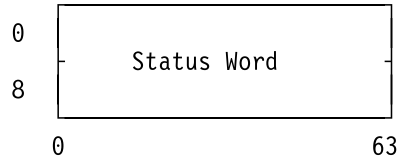


Figure 7-58. Parameter Block for KMAC-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the KMAC instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMAC-Query function completes; condition code 3 is not applicable to this function.

KMAC-DEA (Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-55 on page 7-93.

The parameter block used for the function has the following format:

0	Chaining Value (CV)
8	Cryptographic Key (K)
	0 63

Figure 7-59. Parameter Block for KMAC-DEA

The message authentication code for the 8-byte message blocks (M1, M2, ..., Mn) in operand 2 is computed using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The message authentication code, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

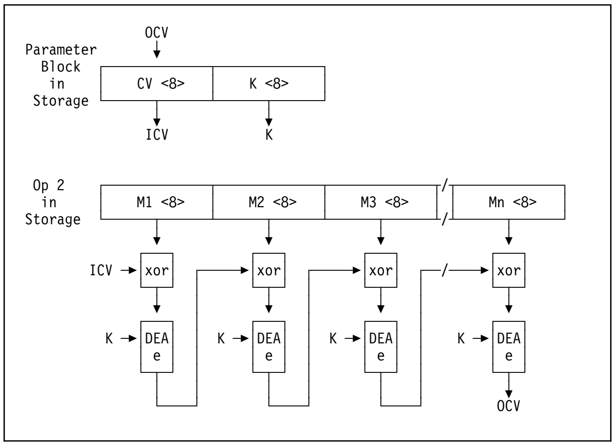


Figure 7-60. KMAC-DEA

KMAC-TDEA-128 (Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-55 on page 7-93.

The parameter block used for the function has the following format:

0	Chaining Value (CV)
8	Cryptographic Key 1 (K1)
16	Cryptographic Key 2 (K2)
	0 63

Figure 7-61. Parameter Block for KMAC-TDEA-128

The message authentication code for the 8-byte message blocks (M1, M2, ..., Mn) in operand 2 is computed using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The message authentication code, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

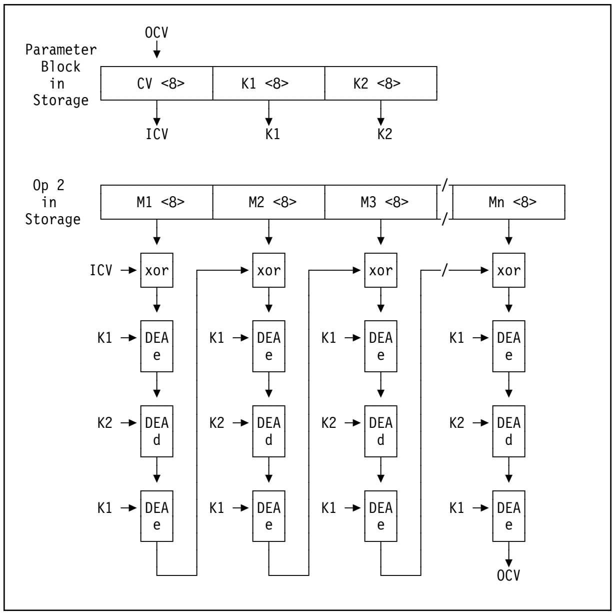


Figure 7-62. KMAC-TDEA-128

KMAC-TDEA-192 (Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-55 on page 7-93.

The parameter block used for the function has the following format:

0	Chaining Value (CV)
8	Cryptographic Key 1 (K1)
16	Cryptographic Key 2 (K2)
24	Cryptographic Key 3 (K3)
	0 63

Figure 7-63. Parameter Block for KMAC-TDEA-192

The message authentication code for the 8-byte message blocks (M1, M2, ..., Mn) in operand 2 is computed using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The message authentication code, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

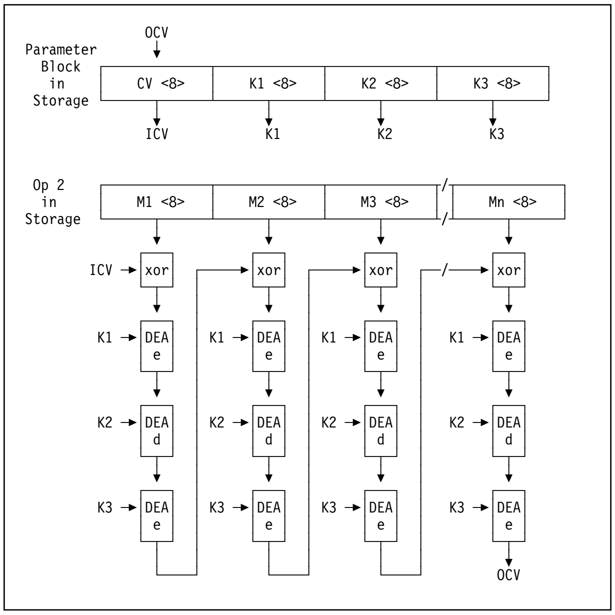


Figure 7-64. KMAC-TDEA-192

Special Conditions for KMAC

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bit 56 of general register 0 is not zero.
2. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
3. The R_2 field designates an odd-numbered register or general register 0.
4. The second-operand length is not a multiple of the data block size of the designated function (see Figure 7-54 on page 7-92 to determine the data block size for COMPUTE MESSAGE AUTHENTICATION CODE functions).

Resulting Condition Code:

0	Normal completion
1	--
2	--
3	Partial completion

Program Exceptions:

- Access (fetch, operand 2, cryptographic key; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- 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 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 7-65. Priority of Execution: KMAC

Programming Notes:

1. Bit 56 of general register 0 is reserved for future extension and should be set to zero.

2. When condition code 3 is set, the second operand address and length in general registers R_2 and $R_2 + 1$, respectively, and 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 for the no-progress case. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.

3. 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; the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

4. Before processing the first part of a message, the program must set the initial values for the chaining-value field. To comply with ANSI X9.9 or X9.19, the initial chaining value shall be set to all binary zeros.

CONVERT TO BINARY

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

┌────────┬────┬────┬────┬────────────┐
│  '4F'  │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0         8    12   16   20          31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '06'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0         8    12   16   20   32       40       47
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '0E'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0         8    12   16   20   32       40       47
```

The second operand is changed from decimal to binary, and the result is placed at the first-operand location.

For CONVERT TO BINARY (CVB, CVBY), the second operand occupies eight bytes in storage, and, for CONVERT TO BINARY (CVBG), the second operand occupies sixteen bytes in storage. The second operand has the format of packed decimal data, as described in Chapter 8, “Decimal Instructions.” It is checked for valid sign and digit codes, and a decimal-operand data exception is recognized when an invalid code is detected.

For CONVERT TO BINARY (CVB, CVBY), the result of the conversion is a 32-bit signed binary integer, which is placed in bit positions 32-63 of general register R_1. Bits 0-31 of the register remain unchanged. The maximum positive number that can be converted and still be contained in 32 bit positions is 2,147,483,647; the maximum negative number (the negative number with the greatest absolute value) that can be converted is −2,147,483,648. For any decimal number outside this range, the operation is completed by placing the 32 rightmost bits of the binary result in the register, and a fixed-point-divide exception is recognized.

For CONVERT TO BINARY (CVBG), the result of the conversion is a 64-bit signed binary integer, which is placed in bit positions 0-63 of general register R_1. The maximum positive number that can be converted and still be contained in a 64-bit register is 9,223,372,036,854,775,807; the maximum negative number (the negative number with the greatest absolute value) that can be converted is −9,223,372,036,854,775,808. For any decimal number outside this range, a fixed-point-divide exception is recognized, and the operation is suppressed.

The displacement for CVB is treated as a 12-bit unsigned binary integer. The displacement for CVBY and CVBG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Data
- Fixed-point divide
- Operation (CVBY, if the long-displacement facility is not installed)

Programming Notes:

1. An example of the use of the CONVERT TO BINARY instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When the second operand is negative, the result is in two's-complement notation.

3. The storage-operand references for CONVERT TO BINARY may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

CONVERT TO DECIMAL

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

┌─────────┬────┬────┬────┬────────────┐
│  '4E'   │ R1 │ X2 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0          8   12   16   20           31
```

```
CVDY   R1,D2(X2,B2)     [RXY]
                          ┌/┐
┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '26'   │
└─────────┴────┴────┴────┴────┴─────────┴─────────┘
                          └/┘
0          8   12   16   20   32        40       47
```

```
CVDG      R1,D2(X2,B2)     [RXY]
                          ┌/┐
┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '2E'   │
└─────────┴────┴────┴────┴────┴─────────┴─────────┘
                          └/┘
0          8   12   16   20   32        40       47
```

The first operand is changed from binary to decimal, and the result is stored at the second-operand location.

For CONVERT TO DECIMAL (CVD, CVDY), the first operand is treated as a 32-bit signed binary integer, and the result occupies eight bytes in storage. For CONVERT TO DECIMAL (CVDG), the first operand is treated as a 64-bit signed binary integer, and the result occupies sixteen bytes in storage.

The result is in the format for packed decimal data, as described in Chapter 8, “Decimal Instructions.” The rightmost four bits of the result represent the sign. A positive sign is encoded as 1100; a negative sign is encoded as 1101.

The displacement for CVD is treated as a 12-bit unsigned binary integer. The displacement for CVDY and CVDG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

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

Programming Notes:

1. An example of the use of the CONVERT TO DECIMAL instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. For CVD and CVDY, the number to be converted is a 32-bit signed binary integer obtained from a general register. Since 15 decimal digits are available for the result, and the decimal equivalent of 31 bits requires at most 10 decimal digits, an overflow cannot occur. Similarly, for CVDG, 31 decimal digits are available, the decimal equivalent of 63 bits is at most 19 digits, and an overflow cannot occur.

3. The storage-operand references for CONVERT TO DECIMAL may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

CONVERT UNICODE TO UTF-8

```
CUUTF    R1,R2     [RRE]

┌─────────────────┬────────┬────┬────┐
│     'B2A6'      │////////│ R1 │ R2 │
└─────────────────┴────────┴────┴────┘
0                 16       24   28  31
```

The two-byte Unicode characters of the second operand are converted to UTF-8 characters and placed at the first-operand location. The UTF-8 characters are one, two, three, or four bytes, depending on the Unicode characters that are converted. The operation proceeds until the end of the first or second operand is reached or a CPU-determined number of characters have been converted, whichever occurs first. The result is indicated in the condition code.

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 the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, 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 second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_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 7-66 on page 7-100.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted or a CPU-determined number of second-operand characters have been converted.

To show the method of converting a Unicode character to a UTF-8 character, the bits of a Unicode character are identified by letters as follows:

```
Unicode Character                   111111
Bit Numbers                1234567 89012345

Identifying Bit Letters   abcdefgh ijklmnop
```

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

```
Unicode High Surrogate              111111
Bit Numbers               01234567 89012345

Identifying Bit Letters   110110ab cdefghij


Unicode Low Surrogate     11112222 22222233
Bit Numbers               67890123 45678901

Identifying Bit Letters   110111kl mnopqrst
```

Any Unicode character in the range 0000 to 007F hex is converted to a one-byte UTF-8 character as follows:

```
Unicode    00000000 0jklmnop
Character

UTF-8      0jklmnop
Character
```

Any Unicode character in the range 0080 to 07FF hex is converted to a two-byte UTF-8 character as follows:

```
Unicode    00000fgh ijklmnop
Character

UTF-8      110fghij 10klmnop
Character
```

Any Unicode character in the range 0800 to D7FF and DC00 to FFFF hex is converted to a three-byte UTF-8 character as follows:

```
Unicode    abcdefgh ijklmnop
Character

UTF-8      1110abcd 10efghij 10klmnop
Character
```

Any Unicode surrogate pair starting with a high surrogate in the range D800 to DBFF hex is converted to a four-byte UTF-8 character as follows:

```
Unicode    110110ab cdefghij 110111kl mnopqrst
Characters

UTF-8      11110uvw 10xyefgh 10ijklmn 10opqrst
Character

           where uvwxy = abcd + 1
```

The first six bits of the second Unicode character are ignored.

The second-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes when the first two bytes are a Unicode high surro-

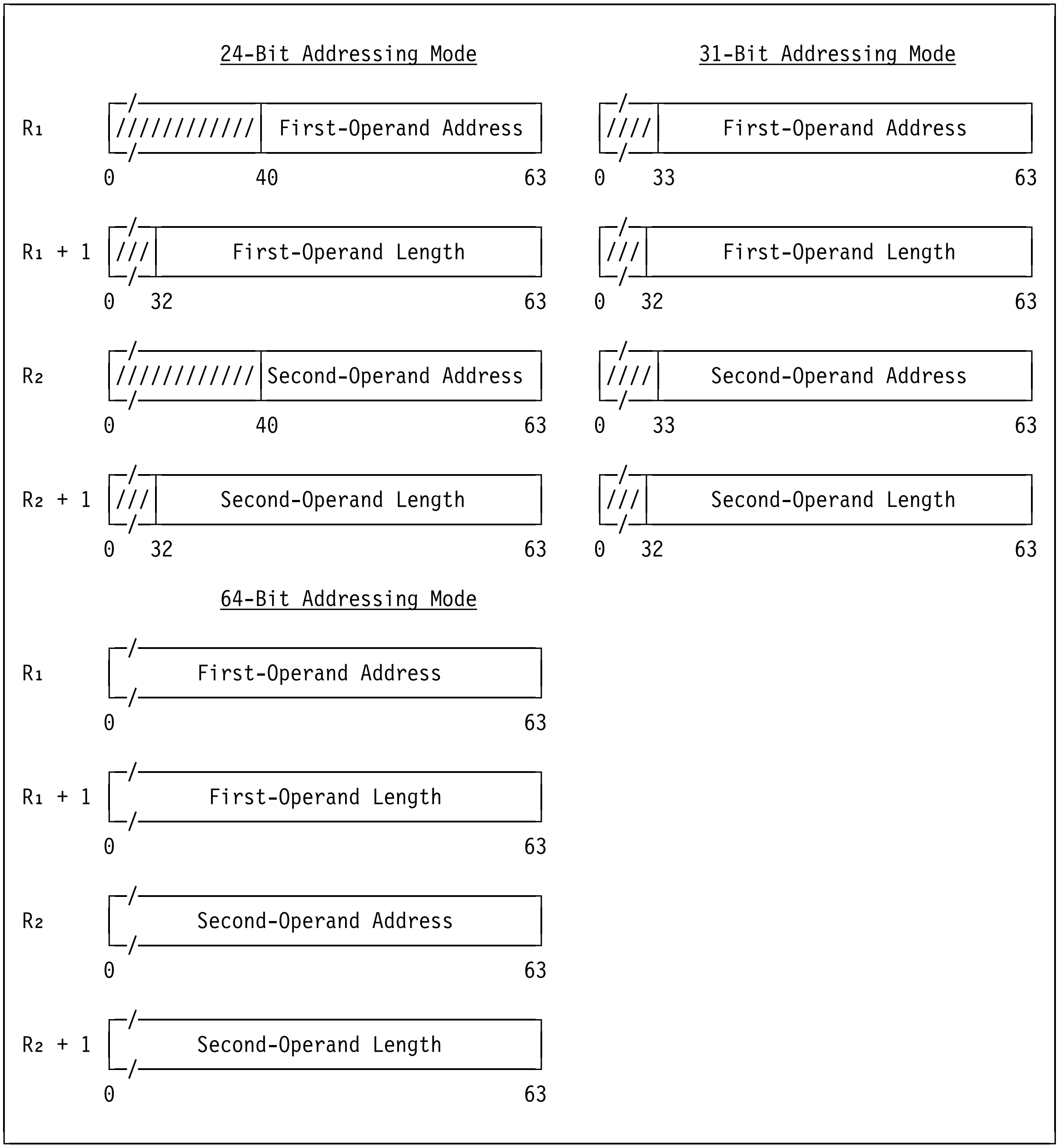


Figure 7-66. Register Contents for CONVERT UNICODE TO UTF-8

gate. The first-operand location is considered exhausted when it does not contain at least the one, two, three, or four remaining bytes required to contain the UTF-8 character resulting from the conversion of the next second-operand character or surrogate pair.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the operation is completed, the contents of general register R_2 + 1 are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register R_1 + 1 are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1

and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

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.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

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.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 --
3 CPU-determined number of characters converted

Program Exceptions:

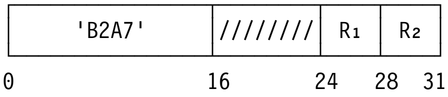
- Access (fetch, operand 2; store, operand 1)
- Specification

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. The storage-operand references of CONVERT UNICODE TO UTF-8 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

CONVERT UTF-8 TO UNICODE

CUTFU R_1,R_2 [RRE]



The one-, two-, three-, or four-byte UTF-8 characters of the second operand are converted to two-byte Unicode characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-8 character is encountered, whichever occurs first. The result is indicated in the condition code.

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 the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers R_1 + 1 and R_2 + 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 second-operand locations is specified by the entire contents of general registers R_1 + 1 and R_2 + 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_2 is dependent on the addressing mode.

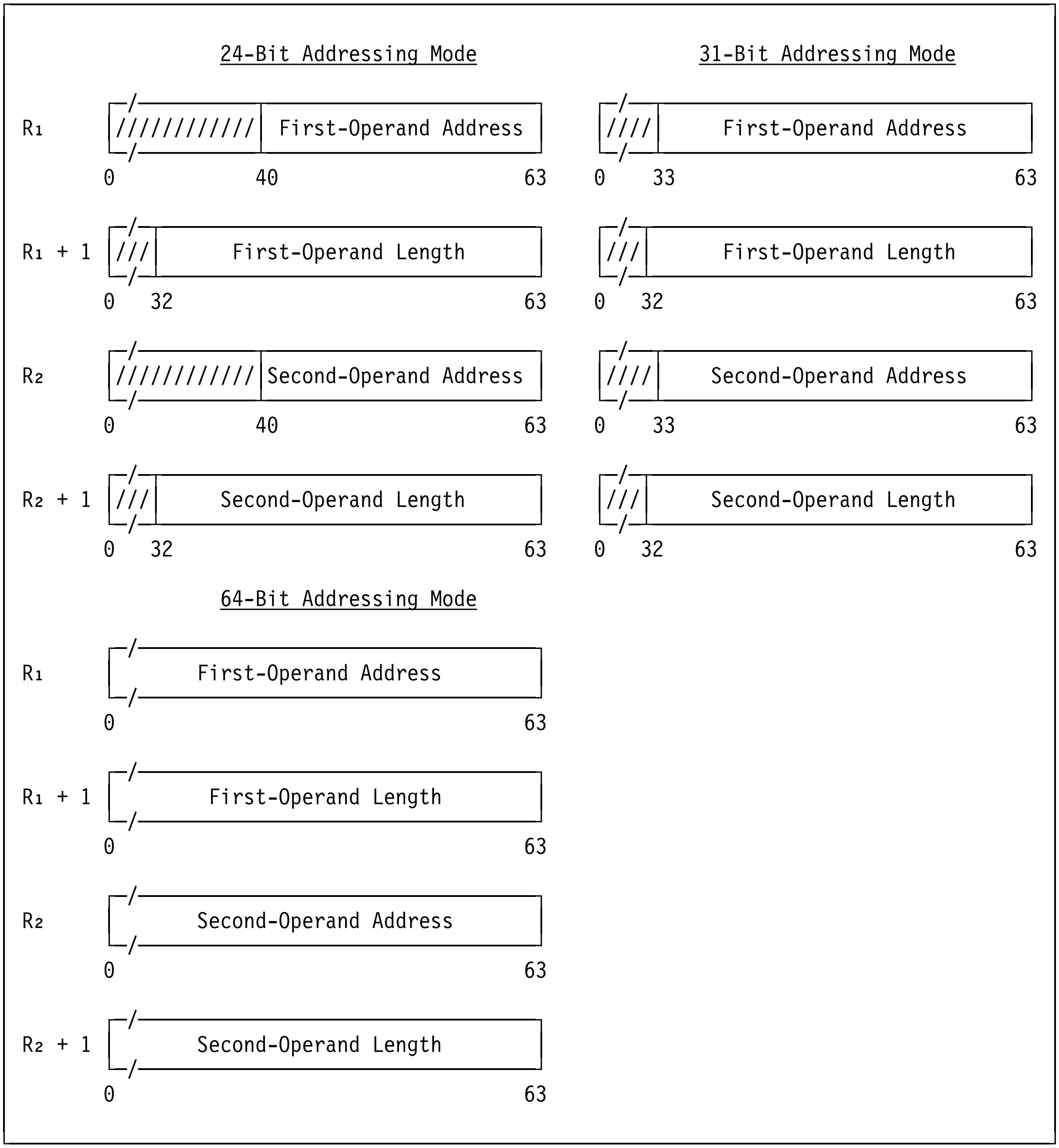


Figure 7-67. Register Contents for CONVERT UTF-8 TO UNICODE

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R₁ and R₂ 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 7-67.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid UTF-8 character is encountered in the second operand.

To show the method of converting a UTF-8 character to a Unicode character, the bits of a Unicode character are identified by letters as follows:

```
Unicode Character                   111111
Bit Numbers               01234567 89012345

Identifying Bit Letters  abcdefgh ijklmnop
```

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

```
Unicode High Surrogate              111111
Bit Numbers               01234567 89012345

Identifying Bit Letters  110110ab cdefghij


Unicode Low Surrogate    11112222 22222233
Bit Numbers               67890123 45678901

Identifying Bit Letters  110111kl mnopqrst
```

When the contents of the first byte of a UTF-8 character are in the range 00 to 7F hex, the character is a one-byte character, and it is converted to a two-byte Unicode character as follows:

```
UTF-8      0jklmnop
Character

Unicode    00000000 0jklmnop
Character
```

When the contents of the first byte of a UTF-8 character are in the range C0 to DF hex, the character is a two-byte character, and it is converted to a two-byte Unicode character as follows:

```
UTF-8      110fghij 10klmnop
Character

Unicode    00000fgh ijklmnop
Character
```

The first two bits in the second byte of the UTF-8 character are ignored.

When the contents of the first byte of a UTF-8 character are in the range E0 to EF hex, the character is a three-byte character, and it is converted to a two-byte Unicode character as follows:

```
UTF-8      1110abcd 10efghij 10klmnop
Character

Unicode    abcdefgh ijklmnop
Character
```

The first two bits in the second and third bytes of the UTF-8 character are ignored.

When the contents of the first byte of a UTF-8 character are in the range F0 to F7 hex, the character is a four-byte character, and it is converted to two two-byte Unicode characters (a surrogate pair) as follows:

```
UTF-8      11110uvw 10xyefgh 10ijklmn 10opqrst
Character

Unicode    110110ab cdefghij 110111kl mnopqrst
Characters

where zabcd = uvwxy -1
```

The first two bits in the second, third, and fourth bytes of the UTF-8 character are ignored. The high order bit (z) produced by the subtract operation should be zero but is ignored.

The second-operand location is considered exhausted when it does not contain at least one remaining byte or when it does not contain at least the two, three, or four remaining bytes required to contain the two-, three-, or four-byte UTF-8 character indicated by the contents of the first remaining byte. The first-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes in the case when a four byte UTF-8 character is to be converted.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the contents of the first byte of the next UTF-8 character are in the range 80 to BF hex or F8 to FF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register R_2 + 1 are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register R_1 + 1 are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

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.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

When condition code 2 is set, general register R_2 contains the address of the invalid UTF-8 character.

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.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid UTF-8 character
3 CPU-determined number of characters processed

Program Exceptions:

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

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. Bits 0 and 1 of the continuation bytes of multiple-byte UTF-8 characters are not checked in order to improve the performance of the conversion. Therefore, invalid continuation bytes are not detected.
3. The storage-operand references of CONVERT UTF-8 TO UNICODE may be multiple-access references. (See "Storage-Operand Consistency" on page 5-87.)

COPY ACCESS

```
CPYA      R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B24D'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

The contents of access register R_2 are placed in access register R_1.

Condition Code: The code remains unchanged.

Program Exceptions: None.

DIVIDE

```
DR     R1,R2     [RR]

┌────────┬────┬────┐
│  '1D'  │ R1 │ R2 │
└────────┴────┴────┘
0        8   12   15

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

┌────────┬────┬────┬────┬────────────┐
│  '5D'  │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8   12   16   20           31
```

The 64-bit first operand (the dividend) is divided by the 32-bit second operand (the divisor), and the

32-bit remainder and quotient are placed at the first-operand location.

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

The dividend is treated as a 64-bit signed binary integer. The leftmost 32 bits of the dividend are in bit positions 32-63 of general register R_1, and the rightmost 32 bits are in bit positions 32-63 of general register $R_1 + 1$.

The divisor, remainder, and quotient are treated as 32-bit signed binary integers. For DIVIDE (DR), the divisor is in bit positions 32-63 of general register R_2. The remainder is placed in bit positions 32-63 of general register R_1, and the quotient is placed in bit positions 32-63 of general register $R_1 + 1$. Bits 0-31 of the registers remain unchanged.

The sign of the quotient is determined by the rules of algebra, and the remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed by a 32-bit signed binary integer, a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of D only)
- Fixed-point divide
- Specification

DIVIDE LOGICAL

```
DLR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B997'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

```
DLGR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B987'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

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

┌────────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2│ DH2│  '97'  │
└────────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20  32      40     47
```

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

┌────────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2│ DH2│  '87'  │
└────────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20  32  36  40     47
```

The 64-bit or 128-bit first operand (the dividend) is divided by the 32-bit or 64-bit second operand (the divisor), and the 32-bit or 64-bit remainder and quotient are placed at the first-operand location.

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

For DIVIDE LOGICAL (DLR, DL), the dividend is treated as a 64-bit unsigned binary integer. The leftmost 32 bits of the dividend are in bit positions 32-63 of general register R_1, and the rightmost 32 bits are in bit positions 32-63 of general register $R_1 + 1$.

The divisor, remainder, and quotient are treated as 32-bit unsigned binary integers. For DIVIDE LOGICAL (DLR), the divisor is in bit positions 32-63 of general register R_2. The remainder is placed in bit positions 32-63 of general register R_1, and the quotient is placed in bit positions 32-63 of general register $R_1 + 1$. Bits 0-31 of the registers remain unchanged.

For DIVIDE LOGICAL (DLGR, DLG), the dividend is treated as a 128-bit unsigned binary integer. The leftmost 64 bits of the dividend are in general register R_1, and the rightmost 64 bits are in general register $R_1 + 1$. The divisor, remainder, and quotient are treated as 64-bit unsigned binary integers. The remainder is placed in general register R_1, and the quotient is placed in general register $R_1 + 1$.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed as a 32-bit unsigned binary integer for DIVIDE LOGICAL (DLR, DL), or a 64-bit unsigned binary integer for DIVIDE LOGICAL (DLGR, DLG), a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of DL or DLG only)
- Fixed-point divide
- Specification

DIVIDE SINGLE

```
DSGR      R1,R2     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B90D'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

```
DSGFR     R1,R2     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B91D'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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

 ┌────────┬────┬────┬────┬────/────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │   DH2   │  '0D'  │
 └────────┴────┴────┴────┴────/────┴────────┴────────┘
 0        8    12   16   20   32        40       47
```

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

 ┌────────┬────┬────┬────┬────/────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │   DH2   │  '1D'  │
 └────────┴────┴────┴────┴────/────┴────────┴────────┘
 0        8    12   16   20   32        40       47
```

The 64-bit contents of general register $R_1 + 1$ (the dividend) are divided by the 64-bit or 32-bit second operand (the divisor), the 64-bit remainder is placed in general register R_1, and the 64-bit quotient is placed in general register $R_1 + 1$.

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

The dividend, quotient, and remainder are treated as 64-bit signed binary integers. For DIVIDE SINGLE (DSGR, DSG), the divisor is treated as a 64-bit signed binary integer. For DIVIDE SINGLE (DSGFR, DSGF), the divisor is treated as a 32-bit signed binary integer. For DSGFR, the divisor is in bit positions 32-63 of general register R_2.

The sign of the quotient is determined by the rules of algebra, and the remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed by a 64-bit signed binary integer, a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of DSG and DSGF only)
- Fixed-point divide
- Specification

EXCLUSIVE OR

```
XR        R1,R2     [RR]

 ┌────────┬────┬────┐
 │  '17'  │ R1 │ R2 │
 └────────┴────┴────┘
 0        8    12  15
```

```
XGR       R1,R2     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B982'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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

 ┌────────┬────┬────┬────┬────────────┐
 │  '57'  │ R1 │ X2 │ B2 │     D2     │
 └────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31
```

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

 ┌────────┬────┬────┬────┬────/────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │   DH2   │  '57'  │
 └────────┴────┴────┴────┴────/────┴────────┴────────┘
 0        8    12   16   20   32        40       47
```

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

 ┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
 │  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '82'   │
 └─────────┴────┴────┴────┴────┴─────────┴─────────┘
 0         8   12   16   20   32        40        47
```

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

 ┌─────────┬─────────┬────┬────────────┐
 │  '97'   │   I2    │ B1 │     D1     │
 └─────────┴─────────┴────┴────────────┘
 0         8        16   20           31
```

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

 ┌─────────┬─────────┬────┬────┬─────────┬─────────┐
 │  'EB'   │   I2    │ B1 │DL1 │   DH1   │  '57'   │
 └─────────┴─────────┴────┴────┴─────────┴─────────┘
 0         8        16   20   32        40        47
```

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

 ┌─────────┬─────────┬────┬────┬────┬────┐
 │  'D7'   │    L    │ B1 │ D1 │ B2 │ D2 │
 └─────────┴─────────┴────┴────┴────┴────┘
 0         8        16   20   32   36  47
```

The EXCLUSIVE OR of the first and second operands is placed at the first-operand location.

The connective EXCLUSIVE OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero.

For EXCLUSIVE OR (XC), 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.

For EXCLUSIVE OR (XI, XIY), the first operand is one byte in length, and only one byte is stored.

For EXCLUSIVE OR (XR, X, XY), the operands are 32 bits, and for EXCLUSIVE OR (XGR, XG), they are 64 bits.

The displacements for X, XI, and both operands of XC are treated as 12-bit unsigned binary integers. The displacement for XY, XIY, and XG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0	Result zero
1	Result not zero
2	--
3	--

Program Exceptions:

- Access (fetch, operand 2, X, XY, XG, and XC; fetch and store, operand 1, XI, XIY, and XC)
- Operation (XY and XIY, if the long-displacement facility is not installed)

Programming Notes:

1. An example of the use of the EXCLUSIVE OR instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. EXCLUSIVE OR may be used to invert a bit, an operation particularly useful in testing and setting programmed binary switches.
3. A field EXCLUSIVE-ORed with itself becomes all zeros.
4. For EXCLUSIVE OR (XR or XGR), the sequence A EXCLUSIVE-OR B, B EXCLUSIVE-OR A, A EXCLUSIVE-OR B results in the exchange of the contents of A and B without the use of an additional general register.
5. Accesses to the first operand of EXCLUSIVE OR (XI) and EXCLUSIVE OR (XC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, EXCLUSIVE OR 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.”

EXECUTE

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

 ┌─────────┬────┬────┬────┬────────────┐
 │  '44'   │ R1 │ X2 │ B2 │     D2     │
 └─────────┴────┴────┴────┴────────────┘
 0         8   12   16   20           31
```

The single instruction at the second-operand address is modified by the contents of general register R_1, and the resulting instruction, called the target instruction, is executed.

When the R_1 field is not zero, bits 8-15 of the instruction designated by the second-operand address are ORed with bits 56-63 of general register R_1. The ORing does not change either the contents of general register R_1 or the instruction in storage, and it is effective only for the interpretation of the instruction to be executed. When the R_1 field is zero, no ORing takes place.

The target instruction may be two, four, or six bytes in length. The execution and exception handling of the target instruction are exactly as if the target instruction were obtained in normal sequential operation, except for the instruction address and the instruction-length code.

The instruction address in the current PSW is increased by the length of EXECUTE. This updated address and the instruction-length code of EXECUTE are used, for example, as part of the link information when the target instruction is BRANCH AND LINK. When the target instruction is a successful branching instruction, the instruction address in the current PSW is replaced by the branch address specified by the target instruction.

When the target instruction is in turn EXECUTE, an execute exception is recognized.

The effective address of EXECUTE must be even; otherwise, a specification exception is recognized. When the target instruction is two or three halfwords in length but can be executed without fetching its second or third halfword, it is unpredictable whether access exceptions are recognized for the unused halfwords. Access exceptions are not recognized for the second-operand address when the address is odd.

The second-operand address of EXECUTE is an instruction address rather than a logical address; thus, the target instruction is fetched from the primary address space when in the primary-space, secondary-space, or access-register mode.

Condition Code: The code may be set by the target instruction.

Program Exceptions:

- Access (fetch, target instruction)
- Execute
- Specification

Programming Notes:

1. An example of the use of the EXECUTE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The ORing of eight bits from the general register with the designated instruction permits the indirect specification of the length, index, mask, immediate-data, register, or extended-op-code field.

3. The fetching of the target instruction is considered to be an instruction fetch for purposes of program-event recording and for purposes of reporting access exceptions.

4. An access or specification exception may be caused by EXECUTE or by the target instruction.

5. When an interruptible instruction is made the target of EXECUTE, the program normally should not designate any register updated by the interruptible instruction as the R_1, X_2, or B_2 register for EXECUTE. Otherwise, on resumption of execution after an interruption, or if the instruction is refetched without an interruption, the updated values of these registers will be used in the execution of EXECUTE. Similarly, the program should normally not let the destination field in storage of an interruptible instruction include the location of EXECUTE, since the new contents of the location may be interpreted when resuming execution.

EXTRACT ACCESS

EAR R_1,R_2 [RRE]

'B24F'	////////	R_1	R_2
0	16	24	28 31

The contents of access register R_2 are placed in bit positions 32-63 of general register R_1. Bits 0-31 of general register R_1 remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions: None.

EXTRACT PSW

```
EPSW     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B98D'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

Bits 0-31 of the current PSW are placed in bit positions 32-63 of the first operand, and bits 0-31 of the operand remain unchanged. Subsequently, bits 32-63 of the current PSW are placed in bit positions 32-63 of the second operand, and bits 0-31 of the operand remain unchanged. The action associated with the second operand is not performed if the R_2 field is zero.

Bits 0-63 of the PSW have the following format:

```
┌─┬─┬─┬─┬─┬─┬─┬─┬───────┬─┬─┬─┬─┬───┬───┬─────┬─────────────┬─┐
│ │ │ │ │ │ │I│E│       │ │ │ │ │   │   │Prog │             │E│
│0│R│0│0│0│T│O│X│  Key  │0│M│W│P│A S│C C│Mask │0 0 0 0 0 0 0│A│
└─┴─┴─┴─┴─┴─┴─┴─┴───────┴─┴─┴─┴─┴───┴───┴─────┴─────────────┴─┘
0         5     8       12      16  18  20    24              31

┌─┬─────────────────────────────────────────────────────────────┐
│B│                                                             │
│A│0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0│
└─┴─────────────────────────────────────────────────────────────┘
32                                                             63
```

Condition Code: The code remains unchanged.

Program Exceptions: None.

INSERT CHARACTER

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

┌────────┬────┬────┬────┬────────────┐
│  '43'  │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20          31

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '73'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47
```

The byte at the second-operand location is inserted into bit positions 56-63 of general register R_1. The remaining bits in the register remain unchanged.

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

Condition Code: The code remains unchanged.

Program Exceptions:

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

INSERT CHARACTERS UNDER MASK

```
ICM     R1,M3,D2(B2)     [RS]

┌────────┬────┬────┬────┬────────────┐
│  'BF'  │ R1 │ M3 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20          31

ICMY    R1,M3,D2(B2)     [RSY]

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'  │ R1 │ M3 │ B2 │DL2 │  DH2   │  '81'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47

ICMH       R1,M3,D2(B2)     [RSY]

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'  │ R1 │ M3 │ B2 │DL2 │  DH2   │  '80'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47
```

Bytes from contiguous locations beginning at the second-operand address are inserted into general register R_1 under control of a mask.

The contents of the M_3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R_1. For INSERT CHARACTERS UNDER MASK (ICM, ICMY), the four bytes to which the mask bits correspond are in the low-order half, bit positions 32-63 of general register R_1. For INSERT CHARACTERS UNDER MASK (ICMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The byte positions corresponding to ones in the mask are filled, left to right, with bytes from successive storage locations beginning at the second-operand address. When the mask is not zero, the length of the second operand is equal to the number of ones in the mask. The bytes in the general register corresponding to zeros in the mask remain unchanged. For ICM and ICMY, bits

0-31 of the register remain unchanged, and, for ICMH, bits 32-63 remain unchanged.

The resulting condition code is based on the mask and on the value of the bits inserted. When the mask is zero or when all inserted bits are zeros, the condition code is set to 0. When the inserted bits are not all zeros, the code is set according to the leftmost bit of the storage operand: if this bit is one, the code is set to 1; if this bit is zero, the code is set to 2.

When the mask is not zero, exceptions associated with storage-operand access are recognized only for the number of bytes specified by the mask. When the mask is zero, access exceptions are recognized for one byte at the second-operand address.

The displacement for ICM is treated as a 12-bit unsigned binary integer. The displacement for ICMY and ICMH is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 All inserted bits zeros, or mask bits all zeros
1 Leftmost inserted bit one
2 Leftmost inserted bit zero, and not all inserted bits zeros
3 --

Program Exceptions:

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

Programming Notes:

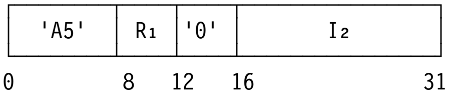
1. Examples of the use of the INSERT CHARACTERS UNDER MASK instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The condition code for INSERT CHARACTERS UNDER MASK (ICM, ICMY only) is defined such that, when the mask is 1111, the instruction causes the same condition code to be set as for LOAD AND TEST (LTR only) Thus, the instruction may be used as a storage-to-register load-and-test operation.

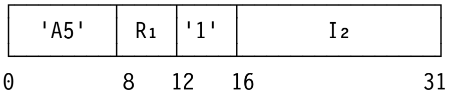
3. INSERT CHARACTERS UNDER MASK (ICM, ICMY) with a mask of 1111 or 0001 performs a function similar to that of a LOAD (L) or INSERT CHARACTER (IC) instruction, respectively, with the exception of the condition-code setting. However, the performance of INSERT CHARACTERS UNDER MASK may be slower.

INSERT IMMEDIATE

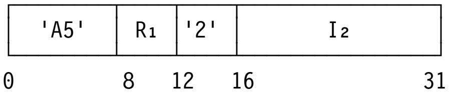
IIHH R_1,I_2 [RI]



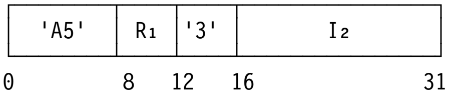
IIHL R_1,I_2 [RI]



IILH R_1,I_2 [RI]



IILL R_1,I_2 [RI]



The second operand is placed in bit positions of the first operand. The remainder of the first operand remains unchanged.

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

Instruction	Bit Positions Loaded
IIHH	0-15
IIHL	16-31
IILH	32-47
IILL	48-63

Condition Code: The code remains unchanged.

Program Exceptions: None.

INSERT PROGRAM MASK

```
IPM    R1               [RRE]

┌────────────────┬────────┬────┬────┐
│     'B222'     │////////│ R1 │////│
└────────────────┴────────┴────┴────┘
0               16       24   28  31
```

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]

┌────────┬────┬────┐
│  '18'  │ R1 │ R2 │
└────────┴────┴────┘
0        8   12  15
```

```
LGR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B904'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0               16       24   28  31
```

```
LGFR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B914'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0               16       24   28  31
```

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

┌────────┬────┬────┬────┬────────────┐
│  '58'  │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8   12   16   20           31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '58'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40      47
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '04'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40      47
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '14'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40      47
```

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]

┌────────┬────┬────┬────┬────────────┐
│  '9A'  │ R1 │ R3 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
0        8   12   16   20           31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'  │ R1 │ R3 │ B2 │DL2 │  DH2   │  '9A'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40      47
```

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]

┌─────────┬────┬────┬────┬────────────┐
│  '41'   │ R1 │ X2 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0          8    12   16   20          31
```

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

┌─────────┬────┬────┬────┬─────/─┬─────────┬─────────┐
│  'E3'   │ R1 │ X2 │ B2 │ DL2   │   DH2   │  '71'   │
└─────────┴────┴────┴────┴─────/─┴─────────┴─────────┘
0          8    12   16   20     32        40        47
```

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]

┌─────────┬────┬────┬────┬────────────┐
│  '51'   │ R1 │ X2 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0          8    12   16   20          31
```

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]

 _____________________________________/___________
|   'C0'   |  R1  | '0' |            I2            |
|__________|______|_____|_____________/____________|
0          8      12    16                        47
```

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]

 ┌─────────┬────┬────┐
 │  '12'   │ R₁ │ R₂ │
 └─────────┴────┴────┘
 0         8   12   15
```

```
LTGR    R₁,R₂     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B902'     │////////│ R₁ │ R₂ │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

```
LTGFR    R₁,R₂     [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B912'     │////////│ R₁ │ R₂ │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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]

 ┌─────────┬────┬────┬────┬───/───┬─────────┬─────────┐
 │  'E3'   │ R₁ │ X₂ │ B₂ │DL₂    │   DH₂   │  '76'   │
 └─────────┴────┴────┴────┴───/───┴─────────┴─────────┘
 0         8   12   16   20    32        40        47
```

```
LGB     R₁,D₂(X₂,B₂)     [RXY]

 ┌─────────┬────┬────┬────┬───/───┬─────────┬─────────┐
 │  'E3'   │ R₁ │ X₂ │ B₂ │DL₂    │   DH₂   │  '77'   │
 └─────────┴────┴────┴────┴───/───┴─────────┴─────────┘
 0         8   12   16   20    32        40        47
```

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]

 ┌─────────┬────┬────┐
 │  '13'   │ R₁ │ R₂ │
 └─────────┴────┴────┘
 0         8   12   15
```

```
LCGR     R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B903'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0               16       24   28  31

LCGFR     R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B913'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0               16       24   28  31
```

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]

 ┌────────┬────┬────┬────┬────────────┐
 │  '48'  │ R1 │ X2 │ B2 │     D2     │
 └────────┴────┴────┴────┴────────────┘
 0        8   12   16   20           31

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

 ┌────────┬────┬────┬────┬────/┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2  │  DH2   │  '78'  │
 └────────┴────┴────┴────┴────/┴────────┴────────┘
 0        8   12   16   20    32       40      47

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

 ┌────────┬────┬────┬────┬────/┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2  │  DH2   │  '15'  │
 └────────┴────┴────┴────┴────/┴────────┴────────┘
 0        8   12   16   20    32       40      47
```

LOAD HALFWORD IMMEDIATE

```
LHI     R1,I2     [RI]

 ┌────────┬────┬────┬────────────────┐
 │  'A7'  │ R1 │'8' │       I2       │
 └────────┴────┴────┴────────────────┘
 0        8   12   16               31

LGHI     R1,I2     [RI]

 ┌────────┬────┬────┬────────────────┐
 │  'A7'  │ R1 │'9' │       I2       │
 └────────┴────┴────┴────────────────┘
 0        8   12   16               31
```

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 R_1,R_2 [RRE]

'B916'	////////	R_1	R_2
0	16	24	28 31

LLGF $R_1,D_2(X_2,B_2)$ [RXY]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'16'
0	8	12	16	20	32	40 47

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 $R_1,D_2(X_2,B_2)$ [RXY]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'90'
0	8	12	16	20	32	40 47

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 $R_1,D_2(X_2,B_2)$ [RXY]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'91'
0	8	12	16	20	32	40 47

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 R_1,I_2 [RI]

'A5'	R_1	'C'	I_2
0	8	12	16 31

LLIHL R_1,I_2 [RI]

'A5'	R_1	'D'	I_2
0	8	12	16 31

LLILH R_1,I_2 [RI]

'A5'	R_1	'E'	I_2
0	8	12	16 31

LLILL R_1,I_2 [RI]

'A5'	R_1	'F'	I_2
0	8	12	16 31

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]

┌────────────────┬────────┬─────┬─────┐
│     'B917'     │////////│ R1  │ R2  │
└────────────────┴────────┴─────┴─────┘
0                16       24    28   31

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

┌────────┬─────┬─────┬─────┬─────┬─────────┬─────────┐
│  'E3'  │ R1  │ X2  │ B2  │ DL2 │   DH2   │  '17'   │
└────────┴─────┴─────┴─────┴─────┴─────────┴─────────┘
0        8     12    16    20    32        40       47
```

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]

┌────────┬─────┬─────┬─────┬────────────┐
│  '98'  │ R1  │ R3  │ B2  │     D2     │
└────────┴─────┴─────┴─────┴────────────┘
0        8     12    16    20          31

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

┌────────┬─────┬─────┬─────┬─────┬─────────┬─────────┐
│  'EB'  │ R1  │ R3  │ B2  │ DL2 │   DH2   │  '98'   │
└────────┴─────┴─────┴─────┴─────┴─────────┴─────────┘
0        8     12    16    20    32        40       47

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

┌────────┬─────┬─────┬─────┬─────┬─────────┬─────────┐
│  'EB'  │ R1  │ R3  │ B2  │ DL2 │   DH2   │  '04'   │
└────────┴─────┴─────┴─────┴─────┴─────────┴─────────┘
0        8     12    16    20    32        40       47
```

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]

┌──────────┬─────┬─────┬─────┬──/──┬─────┬──/──┐
│  'EF'    │ R1  │ R3  │ B2  │ D2  │ B4  │ D4  │
└──────────┴─────┴─────┴─────┴──/──┴─────┴──/──┘
0          8    12    16    20    32    36    47
```

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]

┌──────────┬─────┬─────┬─────┬──/──┬──────────┬──────────┐
│  'EB'    │ R1  │ R3  │ B2  │ DL2 │   DH2    │   '96'   │
└──────────┴─────┴─────┴─────┴──/──┴──────────┴──────────┘
0          8    12    16    20    32         40         47
```

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]

┌──────────┬─────┬─────┐
│  '11'    │ R1  │ R2  │
└──────────┴─────┴─────┘
0          8    12    15
```

LNGR R_1,R_2 [RRE]

'B901'	////////	R_1	R_2	
0	16	24	28	31

LNGFR R_1,R_2 [RRE]

'B911'	////////	R_1	R_2	
0	16	24	28	31

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 $R_1,D_2(X_2,B_2)$ [RXY]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'8F'	
0	8	12	16	20	32	40	47

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 R_1, and the right doubleword is loaded into general register $R_1 + 1$.

The R_1 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 R_1,R_2 [RR]

'10'	R_1	R_2	
0	8	12	15

LPGR R_1,R_2 [RRE]

'B900'	////////	R_1	R_2	
0	16	24	28	31

LPGFR R_1,R_2 [RRE]

'B910'	////////	R_1	R_2	
0	16	24	28	31

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]

┌─────────────────┬────────┬────┬────┐
│     'B91F'      │////////│ R₁ │ R₂ │
└─────────────────┴────────┴────┴────┘
0                16       24   28   31

LRVGR      R₁,R₂      [RRE]

┌─────────────────┬────────┬────┬────┐
│     'B90F'      │////////│ R₁ │ R₂ │
└─────────────────┴────────┴────┴────┘
0                16       24   28   31

LRVH      R₁,D₂(X₂,B₂)      [RXY]

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '1F'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40       47

LRV      R₁,D₂(X₂,B₂)      [RXY]

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '1E'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40       47

LRVG      R₁,D₂(X₂,B₂)      [RXY]

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '0F'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8   12   16   20   32       40       47
```

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]

 ┌─────────┬─────────┬────┬────────────┐
 │  'AF'   │   I2    │ B1 │     D1     │
 └─────────┴─────────┴────┴────────────┘
0          8        16   20           31
```

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.

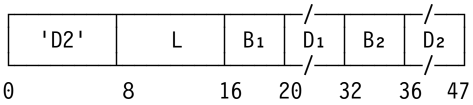
2. The monitor masks provide a means of disallowing all monitor-event program interruptions or allowing monitor-event program interruptions for all or selected classes.

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]

   ┌─────────┬─────────┬────┬────────────┐
   │  '92'   │   I2    │ B1 │     D1     │
   └─────────┴─────────┴────┴────────────┘
  0          8        16   20           31

| MVIY   D1(B1),I2         [SIY]
|                              /
|  ┌─────────┬─────────┬────┬────┬─────────┬─────────┐
|  │  'EB'   │   I2    │ B1 │DL1 │   DH1   │  '52'   │
|  └─────────┴─────────┴────┴────┴─────────┴─────────┘
|                              /
| 0          8        16   20   32        40        47
```

MVC $D_1(L,B_1),D_2(B_2)$ [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:

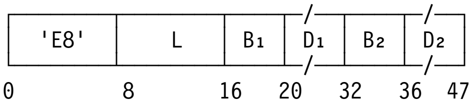
- Access (fetch, operand 2 of MVC; store, operand 1, MVI, MVIY, and MVC)
- Operation (MVIY, if the long-displacement facility is not installed)

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 $D_1(L,B_1),D_2(B_2)$ [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 R_1,R_2 [RR]

'0E'	R_1	R_2	
0	8	12	15

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

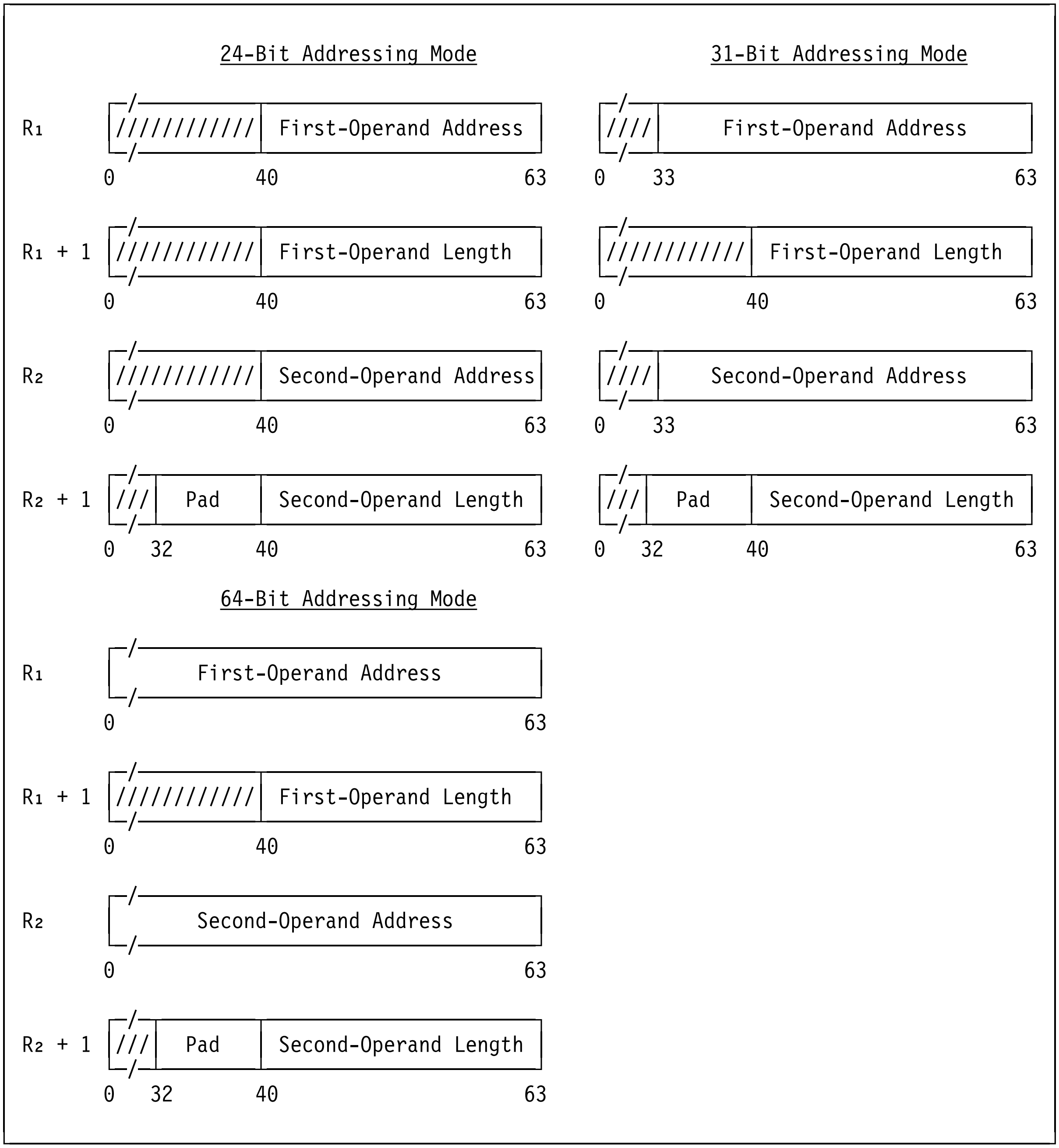


Figure 7-68. Register Contents for MOVE LONG

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]
```

'A8'	R_1	R_3	B_2	D_2	
0	8	12	16	20	31

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.

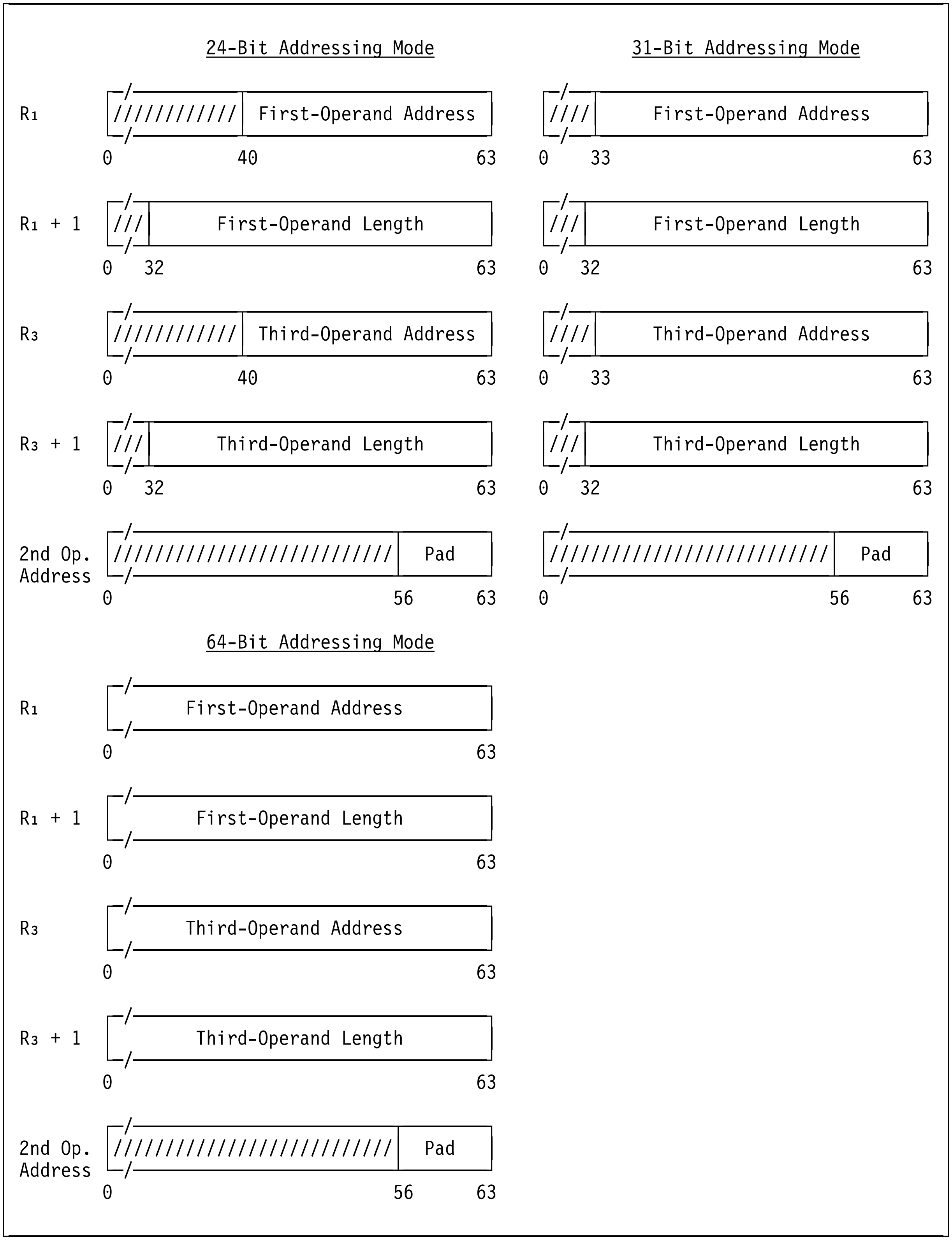


Figure 7-69. Register Contents and Second-Operand Address for MOVE LONG EXTENDED

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]

┌─────────┬────┬────┬────┬───/──┬─────────┬─────────┐
│  'EB'   │ R1 │ R3 │ B2 │DL2   │   DH2   │  '8E'   │
└─────────┴────┴────┴────┴───/──┴─────────┴─────────┘
0         8    12   16   20     32        40       47
```

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

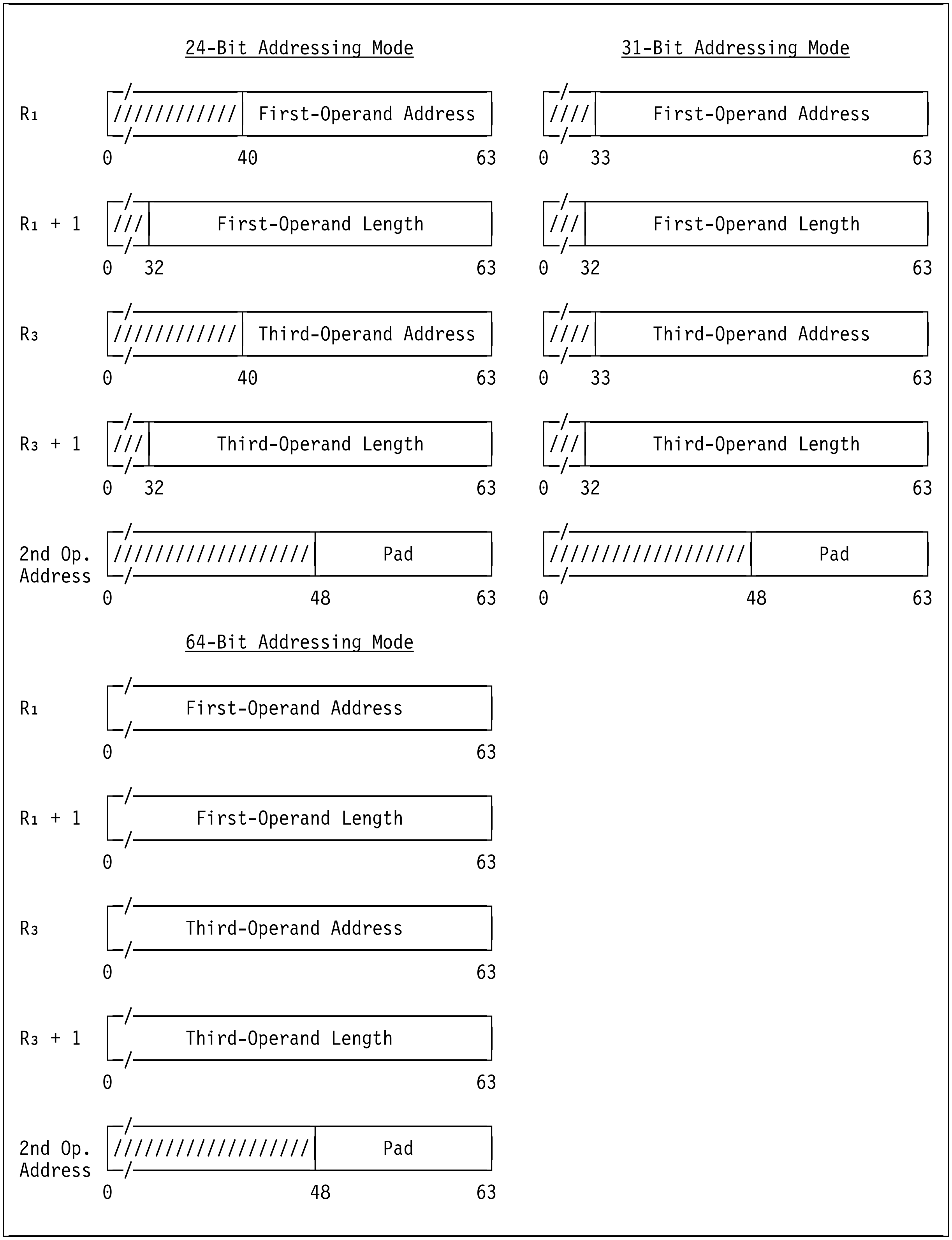


Figure 7-70. Register Contents and Second-Operand Address for MOVE LONG UNICODE

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 $D_1(L,B_1),D_2(B_2)$ [SS]

```
┌─────────┬─────────┬────┬──/──┬────┬──/──┐
│  'D1'   │    L    │ B1 │ D1  │ B2 │ D2  │
└─────────┴─────────┴────┴──/──┴────┴──/──┘
0         8        16   20    32   36    47
```

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 R_1,R_2 [RRE]

```
┌─────────────────┬─────────┬────┬────┐
│     'B255'      │//////// │ R1 │ R2 │
└─────────────────┴─────────┴────┴────┘
0                16        24   28   31
```

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:

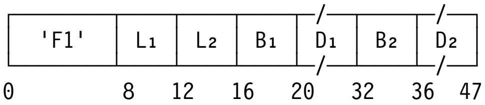
- Access (fetch, operand 2; store, operand 1)
- Specification

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:

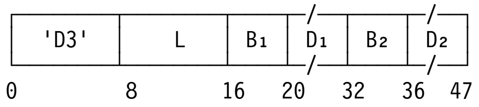
- Access (fetch, operand 2; fetch and store, operand 1)

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 $D_1(L,B_1),D_2(B_2)$ [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:

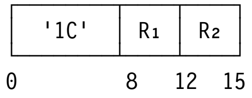
- Access (fetch, operand 2; fetch and store, operand 1)

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 R_1,R_2 [RR]



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

 ┌─────────┬────┬────┬────┬────────────┐
 │  '5C'   │ R1 │ X2 │ B2 │     D2     │
 └─────────┴────┴────┴────┴────────────┘
 0         8   12   16   20           31
```

The 32-bit first operand (the multiplicand) is multiplied by the 32-bit second-operand (the multiplier), and the 64-bit product is placed at the first-operand location.

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

Both the multiplicand and multiplier are treated as 32-bit signed binary integers. The multiplicand is in bit positions 32-63 of general register $R_1 + 1$. For MULTIPLY (MR), the multiplier is in bit positions 32-63 of general register R_2 The contents of general register R_1 and of bit positions 0-31 of general register $R_1 + 1$ and, for MR, of general register R_2 are ignored.

The product is a 64-bit signed binary integer. Bits 0-31 of the product replace bits 32-63 of general register R_1. Bits 32-63 of the product replace bits 32-63 of general register $R_1 + 1$. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged. An overflow cannot occur.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of M only)
- Specification

Programming Notes:

1. An example of the use of the MULTIPLY instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The significant part of the product usually occupies 62 bit positions or fewer. Only when two maximum 32-bit negative numbers are multiplied are 63 significant product bits formed.

MULTIPLY HALFWORD

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

 ┌─────────┬────┬────┬────┬────────────┐
 │  '4C'   │ R1 │ X2 │ B2 │     D2     │
 └─────────┴────┴────┴────┴────────────┘
 0         8   12   16   20           31
```

MULTIPLY HALFWORD IMMEDIATE

```
MHI     R1,I2     [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A7'   │ R1 │'C' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0         8   12   16               31

MGHI     R1,I2     [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A7'   │ R1 │'D' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0         8   12   16               31
```

The 32-bit or 64-bit first operand (the multiplicand) is multiplied by the 16-bit second operand (the multiplier), and the rightmost 32 or 64 bits of the product are 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 MULTIPLY HALFWORD and MULTIPLY HALFWORD IMMEDIATE (MHI), the multiplicand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R_1, and it is replaced by the rightmost 32 bits of the signed-binary-integer product. The bits to the left of the 32 rightmost bits of the product are not tested for significance; no overflow indication is given. Bits 0-31 of general register R_1 are ignored and remain unchanged.

For MULTIPLY HALFWORD IMMEDIATE (MGHI), The multiplicand is treated as a 64-bit signed binary integer in bit positions 0-63 of general register R_1, and it is replaced by the rightmost 64 bits of the signed-binary-integer product. The bits to the left of the 64 rightmost bits of the product are not tested for significance; no overflow indication is given.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of MH only)

Programming Notes:

1. An example of the use of the MULTIPLY HALFWORD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. For MH and MHI, the significant part of the product usually occupies 46 bit positions or fewer. Only when two maximum negative numbers are multiplied are 47 significant product bits formed. Since the rightmost 32 bits of the product are placed unchanged at the first-operand location, ignoring all bits to the left, the sign bit of the result may differ from the true sign of the product in the case of overflow. For a negative product, the 32 bits placed in register R_1 are the rightmost part of the product in two's-complement notation. For MGHI, the significant part of the product usually occupies 78 bit positions or fewer.

MULTIPLY LOGICAL

```
MLR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B996'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

```
MLGR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B986'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '96'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47
```

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

┌────────┬────┬────┬────┬────┬────────┬────────┐
│  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '86'  │
└────────┴────┴────┴────┴────┴────────┴────────┘
0        8    12   16   20   32       40      47
```

The 32-bit or 64-bit first operand (the multiplicand) is multiplied by the 32-bit or 64-bit second operand (the multiplier), and the 64-bit or 128-bit product is placed at the first-operand location.

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

For MULTIPLY LOGICAL (MLR, ML), both the multiplicand and the multiplier are treated as 32-bit unsigned binary integers. The multiplicand is in bit positions 32-63 of general register $R_1 + 1$. For MULTIPLY LOGICAL (MLR), the multiplier is in bit positions 32-63 of general register R_2. The contents of general register R_1 and of bit positions 0-31 of general register $R_1 + 1$ and, for MLR, of general register R_2 are ignored. The product is a 64-bit unsigned binary integer. Bits 0-31 of the product replace bits 32-63 of general register R_1, and bits 32-63 of the product replace bits 32-63 of general register $R_1 + 1$. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged. An overflow cannot occur.

For MULTIPLY LOGICAL (MLGR, MLG), the multiplicand and the multiplier are treated as 64-bit unsigned binary integers. The multiplicand is in general register $R_1 + 1$. The contents of general register R_1 are ignored. The product is a 128-bit unsigned binary integer. Bits 0-63 of the product replace the contents of general register R_1, and bits 64-127 of the product replace the contents of general register $R_1 + 1$. An overflow cannot occur.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of ML and MLG only)
- Specification

MULTIPLY SINGLE

```
MSR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B252'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

```
MSGR      R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B90C'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

```
MSGFR      R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B91C'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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

 ┌────────┬────┬────┬────┬────────────┐
 │  '71'  │ R1 │ X2 │ B2 │     D2     │
 └────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '51'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '0C'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '1C'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

The first operand (multiplicand) is multiplied by the second operand (multiplier), and the rightmost 32 or 64 bits of the product are placed at the first-operand location.

For MULTIPLY SINGLE (MSR, MS, MSY), the multiplicand, multiplier, and product are treated as 32-bit signed binary integers. The multiplicand is taken from bit positions 32-63 of general register R_1 and is replaced by the rightmost 32 bits of the signed-binary-integer product. Bits 0-31 of general register R_1 remain unchanged. For MSR, the multiplier is in bit positions 32-63 of general register R_2. The bits to the left of the 32 rightmost bits of the product are not tested for significance; no overflow indication is given.

For MULTIPLY SINGLE (MSGR, MSGFR, MSG, MSGF), the multiplicand, multiplier, and product are treated as 64-bit signed binary integers, except that, for MSGFR and MSGF, the multiplier is treated as a 32-bit signed binary integer. The multiplicand is taken from general register R_1 and is replaced by the rightmost 64 bits of the signed-binary-integer product. For MSGFR, the multiplier is in bit positions 32-63 of general register R_2. The bits to the left of the 64 rightmost bits of the product are not tested for significance; no overflow indication is given.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

The displacement for MS is treated as a 12-bit unsigned binary integer. The displacement for MSY, MSG, and MSGF is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of MS, MSY, MSG, MSGF only)
- Operation (MSY, if the long-displacement facility is not installed)

OR

```
OR      R1,R2      [RR]

 ┌────────┬────┬────┐
 │  '16'  │ R1 │ R2 │
 └────────┴────┴────┘
 0        8    12  15
```

```
OGR      R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B981'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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

 ┌────────┬────┬────┬────┬────────────┐
 │  '56'  │ R1 │ X2 │ B2 │     D2     │
 └────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31
```

```
OY     R₁,D₂(X₂,B₂)     [RXY]

 ┌────────┬────┬────┬────┬───/───┬────────┬────────┐
 │  'E3'  │ R₁ │ X₂ │ B₂ │DL₂    │  DH₂   │  '56'  │
 └────────┴────┴────┴────┴───/───┴────────┴────────┘
 0        8   12   16   20     32       40       47
```

```
OG      R₁,D₂(X₂,B₂)     [RXY]

 ┌────────┬────┬────┬────┬───/───┬────────┬────────┐
 │  'E3'  │ R₁ │ X₂ │ B₂ │DL₂    │  DH₂   │  '81'  │
 └────────┴────┴────┴────┴───/───┴────────┴────────┘
 0        8   12   16   20     32       40       47
```

```
OI     D₁(B₁),I₂        [SI]

 ┌────────┬────────┬────┬────────────┐
 │  '96'  │   I₂   │ B₁ │     D₁     │
 └────────┴────────┴────┴────────────┘
 0        8       16   20           31
```

```
OIY    D₁(B₁),I₂        [SIY]

 ┌────────┬────────┬────┬───/───┬────────┬────────┐
 │  'EB'  │   I₂   │ B₁ │DL₁    │  DH₁   │  '56'  │
 └────────┴────────┴────┴───/───┴────────┴────────┘
 0        8       16   20     32       40       47
```

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

 ┌────────┬────────┬────┬──/──┬────┬──/──┐
 │  'D6'  │   L    │ B₁ │ D₁  │ B₂ │ D₂  │
 └────────┴────────┴────┴──/──┴────┴──/──┘
 0        8       16   20    32   36   47
```

The OR of the first and second operands is placed at the first-operand location.

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.

For OR (OC), 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.

For OR (OI, OIY), the first operand is one byte in length, and only one byte is stored.

For OR (OR, O, OY), the operands are 32 bits, and for OR (OGR, OG), they are 64 bits.

The displacements for O, OI, and both operands of OC are treated as 12-bit unsigned binary integers. The displacement for OY, OIY, and OG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

Program Exceptions:

- Access (fetch, operand 2, O, OY, OG, and OC; fetch and store, operand 1, OI, OIY, and OC)
- Operation (OY and OIY, if the long-displacement facility is not installed)

Programming Notes:

1. Examples of the use of the OR instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. OR may be used to set a bit to one.
3. Accesses to the first operand of OR (OI) and OR (OC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, OR 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 in “Multiprogramming and Multiprocessing Examples” in Appendix A, “Number Representation and Instruction-Use Examples.”

OR IMMEDIATE

```
OIHH     R₁,I₂     [RI]

 ┌────────┬────┬────┬────────────────┐
 │  'A5'  │ R₁ │'8' │       I₂       │
 └────────┴────┴────┴────────────────┘
 0        8   12   16               31
```

```
OIHL     R₁,I₂     [RI]

 ┌────────┬────┬────┬────────────────┐
 │  'A5'  │ R₁ │'9' │       I₂       │
 └────────┴────┴────┴────────────────┘
 0        8   12   16               31
```

```
OILH     R1,I2      [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A5'   │ R1 │'A' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0         8   12   16               31
```

```
OILL     R1,I2      [RI]

 ┌─────────┬────┬────┬────────────────┐
 │  'A5'   │ R1 │'B' │       I2       │
 └─────────┴────┴────┴────────────────┘
 0         8   12   16               31
```

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]

 ┌─────────┬────┬────┬────┬──/─┬────┬──/─┐
 │  'F2'   │ L1 │ L2 │ B1 │ D1 │ B2 │ D2 │
 └─────────┴────┴────┴────┴──/─┴────┴──/─┘
 0         8   12   16   20   32   36  47
```

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:

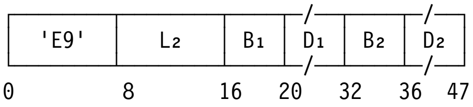
- Access (fetch, operand 2; store, operand 1)

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

'E1'	L_2	B_1	D_1	B_2	D_2	
0	8	16	20	32	36	47

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]

┌────────┬────┬────┬────┬───/───┬────┬───/───┐
│  'EE'  │ R1 │ R3 │ B2 │  D2   │ B4 │  D4   │
└────────┴────┴────┴────┴───/───┴────┴───/───┘
0         8   12   16   20     32   36   47
```

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.

The function code (FC) is in bit positions 56-63 of general register 0. The function code specifies not only the operation to be performed but also the length of the operands and whether the first-operand comparison and replacement values and the third operand or third-operand comparison and replacement values, which are referred to collectively simply as the first and third operands, are in general registers or a parameter list. The pattern of the function codes is as follows:

- A function code that is a multiple of 4 (including 0) specifies a 32-bit length with the first and third operands in bit positions 32-63 of general registers.
- A function code that is one more than a multiple of 4 specifies a 64-bit length with the first and third operands in a parameter list.
- A function code that is 2 more than a multiple of 4 specifies a 64-bit length with the first and third operands in bit positions 0-63 of general registers.
- A function code that is 3 more than a multiple of 4 specifies a 128-bit length with the first and third operands in a parameter list.

Figure 7-71 shows the function codes, operation names, and operand lengths, and also symbols that may be used to refer to the operations in discussions. For example, PLO.DCS may be used to mean PERFORM LOCKED OPERATION with function code 8. In the symbols, the letter "G" indicates a 64-bit operand length, the letter "R" indicates that some or all 64-bit operands are in general registers, and the letter "X" indicates a 128-bit operand length.

The CPU can perform all of the operations specified by the function codes listed in Figure 7-71. Function codes specifying operations that the CPU can perform are called valid. Function codes that have not been assigned to operations or that specify operations that the CPU cannot perform because the operations are not implemented (installed) are called invalid.

Bit 55 of general register 0 is the test bit (T). When bit 55 is zero, the function code in general register 0 must be valid; otherwise, a specification exception is recognized. When bit 55 is one, the condition code is set to 0 if the function code is valid or to 3 if the function code is invalid, and no other operation is performed.

Function Code	Operation	Operand Length (Bits)	Function Symbol
0	Compare and load	32	CL
1	Same as 0	64	CLG
2	Same as 0	64	CLGR
3	Same as 0	128	CLX
4	Compare and swap	32	CS
5	Same as 4	64	CSG
6	Same as 4	64	CSGR
7	Same as 4	128	CSX
8	Double compare and swap	32	DCS
9	Same as 8	64	DCSG
10	Same as 8	64	DCSGR
11	Same as 8	128	DCSX
12	Compare and swap and store	32	CSST
13	Same as 12	64	CSSTG
14	Same as 12	64	CSSTGR
15	Same as 12	128	CSSTX
16	Compare and swap and double store	32	CSDST
17	Same as 16	64	CSDSTG
18	Same as 16	64	CSDSTGR
19	Same as 16	128	CSDSTX
20	Compare and swap and triple store	32	CSTST
21	Same as 20	64	CSTSTG
22	Same as 20	64	CSTSTGR
23	Same as 20	128	CSTSTX

Figure 7-71. PERFORM LOCKED OPERATION Function Codes and Operations

Bits 32-54 of general register 0 must be all zeros; otherwise, a specification exception is recognized. When bit 55 of the register is one, this is the only exception that can be recognized. Bits 0-31 of general register 0 are ignored.

The lock to be used is represented by a program lock token (PLT) whose logical address is specified in general register 1. In the 24-bit addressing mode, the PLT address is bits 40-63 of general register 1, and bits 0-39 of the register are ignored. In the 31-bit addressing mode, the PLT address is bits 33-63 of the register, and bits 0-32 of the register are ignored. In the 64-bit addressing mode, the PLT address is bits 0-63 of the register.

The contents of general registers 0 and 1 described above are as follows:

```
GR 0
┌─/─┬────────────────────────┬─┬──────────┐
│///│000000000000000000000000│T│    FC    │
└─/─┴────────────────────────┴─┴──────────┘
0   32                        56         63
```

```
GR 1 in 24-Bit Addressing Mode
┌─/──────────┬─────────────────────────────┐
│////////////│         PLT Address         │
└─/──────────┴─────────────────────────────┘
0            40                          63
```

```
GR 1 in 31-Bit Addressing Mode
┌─/──┬─────────────────────────────────────┐
│////│             PLT Address             │
└─/──┴─────────────────────────────────────┘
0    33                                  63
```

```
GR 1 in 64-Bit Addressing Mode
┌──────────────────────────────────────────┐
│               PLT Address                │
└──────────────────────────────────────────┘
0                                        63
```

For the even-numbered function codes, including 0, the first-operand comparison value is in general register R_1. For the even-numbered function codes beginning with 4, the first-operand replacement value is in general register $R_1 + 1$, and R_1 designates an even-odd pair of registers and must designate an even-numbered register; otherwise, a specification exception is recognized. For function codes 0 and 2, R_1 can be even or odd.

For function codes 0, 2, 12, and 14, the third operand is in general register R_3, and R_3 can be even or odd.

For function codes 8 and 10, the third-operand comparison value is in general register R_3, the third-operand replacement value is in general register $R_3 + 1$, and R_3 designates an even-odd pair of registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For all function codes, the B_2 and D_2 fields of the instruction specify the second-operand address.

For function codes 0, 2, 8, 10, 12, and 14, the B_4 and D_4 fields of the instruction specify the fourth-operand address.

For function codes 1, 3, 5, 7, 9, 11, 13, 15, and 16-23, the B_4 and D_4 fields of the instruction specify the address of a parameter list that is used by the instruction, and this address is not called the fourth-operand address. The parameter list contains odd-numbered operands, including comparison and replacement values, and addresses of even-numbered operands other than the second operand. In the access-register mode, the parameter list also contains access-list-entry tokens (ALETs) associated with the even-numbered-operand addresses.

In the access-register mode, for function codes that cause use of a parameter list containing an ALET, R_3 must not be zero; otherwise, a specification exception is recognized.

The rules about R_1 and R_3, and the use of the address specified by B_4 and D_4, are summarized in Figure 7-72 on page 7-147.

Function Codes	Operation	R_1	R_3	$D_4(B_4)$
0 and 2	Compare and load	EO	EO	Op4a
1 and 3	Compare and load	-	NZ	PLa
4 and 6	Compare and swap	E	-	-
5 and 7	Compare and swap	-	-	PLa
8 and 10	Double compare and swap	E	E	Op4a
9 and 11	Double compare and swap	-	NZ	PLa
12 and 14	Compare and swap and store	E	EO	Op4a
13 and 15	Compare and swap and store	-	NZ	PLa
16 and 18	Compare and swap and double store	E	NZ	PLa
17 and 19	Compare and swap and double store	-	NZ	PLa
20 and 22	Compare and swap and triple store	E	NZ	PLa
21 and 23	Compare and swap and triple store	-	NZ	PLa

Explanation:

-	Ignored.
E	Must be even.
EO	Can be even or odd.
NZ	Must be nonzero in the access-register mode. Ignored otherwise.
Op4a	$D_4(B_4)$ is operand-4 address.
PLa	$D_4(B_4)$ is parameter-list address.

Figure 7-72. Register Rules and $D_4(B_4)$ Usage for PERFORM LOCKED OPERATION

Figure 7-73 on page 7-148 shows the locations of the operands (including operand comparison and replacement values), operand addresses, and parameter-list address used by the instruction.

Operand addresses in a parameter list, if used, are in doublewords in the list. In the 24-bit addressing mode, an operand address is bits 40-63 of a doubleword, and bits 0-39 of the doubleword are ignored. In the 31-bit addressing mode, an operand address is bits 33-63 of a doubleword, and bits 0-32 of the doubleword are ignored. In the 64-bit addressing mode, an operand address is bits 0-63 of a doubleword.

In the access-register mode, access register 1 specifies the address space containing the program lock token (PLT), access register B_2 specifies the address space containing the second operand, and access register B_4 specifies the address space containing a fourth operand or a parameter list as shown in Figure 7-73 on page 7-148. Also, for an operand whose address is in the parameter list, an access-list-entry token (ALET) is in the list along with the address and is used in the access-register mode to specify the address space containing the operand.

In the access-register mode, if an access exception or PER storage-alteration event is recognized for an operand whose address is in the parameter list, the associated ALET in the parameter list is loaded into access register R_3 when the exception or event is recognized. Then, during the resulting program interruption, if a value is due to be stored as the exception access identification at real location 160 or the PER access identification at real location 161, R_3 is stored. If the instruction execution is completed without the recognition of an exception or event, the contents of access register R_3 are unpredictable. When not in the access-register mode, or when a parameter list containing an ALET is not used, the contents of access register R_3 remain unchanged.

The even-numbered (2, 4, 6, and 8) storage operands must be designated on an integral boundary, which is a word boundary for function codes that are a multiple of 4, a doubleword boundary for function codes that are one or 2 more than a multiple of 4, or a quadword boundary for function codes that are 3 more than a multiple of 4. A parameter list, if used, must be designated on a doubleword boundary. Otherwise, a specification exception is recognized. The program-lock-token (PLT) address in general register 1 does not have a boundary-alignment requirement.

All unused fields in a parameter list should contain all zeros; otherwise, the program may not operate compatibly in the future.

A serialization operation is performed immediately after the lock is obtained and again immediately before it is released. However, values fetched from the parameter list before the lock is obtained are not necessarily refetched. A serialization operation is not performed if the test bit, bit 55 of general register 0, is one.

In the following figures showing the parameter lists for the different function codes, the offsets shown on the left are byte values.

Function Codes[1]	Operation	Op1c	Op1r	Op2a	Op3 or Op3c	Op3r	Op4a	Op5 and Op6a	Op7 and Op8a	PLa
0 and 2	Compare and load	R_1	-	$D_2(B_2)$	R_3		$D_4(B_4)$	-	-	-
1 and 3	Compare and load	PL	-	$D_2(B_2)$	PL		PL	-	-	$D_4(B_4)$
4 and 6	Compare and swap	R_1	R_1+1	$D_2(B_2)$	-		-	-	-	-
5 and 7	Compare and swap	PL	PL	$D_2(B_2)$	-		-	-	-	$D_4(B_4)$
8 and 10	Double compare and swap	R_1	R_1+1	$D_2(B_2)$	R_3	R_3+1	$D_4(B_4)$	-	-	-
9 and 11	Double compare and swap	PL	PL	$D_2(B_2)$	PL	PL	PL	-	-	$D_4(B_4)$
12 and 14	Compare and swap and store	R_1	R_1+1	$D_2(B_2)$	R_3		$D_4(B_4)$	-	-	-
13 and 15	Compare and swap and store	PL	PL	$D_2(B_2)$	PL		PL	-	-	$D_4(B_4)$
16 and 18	Compare and swap and double store	R_1	R_1+1	$D_2(B_2)$	PL		PL	PL	-	$D_4(B_4)$
17 and 19	Compare and swap and double store	PL	PL	$D_2(B_2)$	PL		PL	PL	-	$D_4(B_4)$
20 and 22	Compare and swap and triple store	R_1	R_1+1	$D_2(B_2)$	PL		PL	PL	PL	$D_4(B_4)$
21 and 23	Compare and swap and triple store	PL	PL	$D_2(B_2)$	PL		PL	PL	PL	$D_4(B_4)$

Explanation:

[1] For function codes that are a multiple of 4 (including 0) or one more than a multiple of 4, operands in general registers are in bit positions 32-63 of the registers, and bits 0-31 of the registers are ignored and remain unchanged. For function codes that are two more than a multiple of 4, operands in general registers are in bit positions 0-63 of the registers.

- Operand, value, or address is not used in the operation.
OpNc Operand-N comparison value.
OpNr Operand-N replacement value.
OpNa Operand-N address.
PL Operand, value, or address is in the parameter list.
PLa Parameter-list address.

Figure 7-73. Operand and Address Locations for PERFORM LOCKED OPERATION

Function Codes 0-3 (Compare and Load)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-73.

The parameter list used for function code 1 has the following format:

Parameter List for Function Code 1

0		
8	Operand-1 Comparison Value	
16		
24		
32		
40	Operand 3	
48		
56		
64		Operand-4 ALET
72	Operand-4 Address	

The parameter list used for function code 3 has the following format:

Parameter List for Function Code 3

0	Operand-1 Comparison Value	
8	Operand-1 Comp. Value (continued)	
16		
24		
32	Operand 3	
40	Operand 3 (continued)	
48		
56		
64		Operand-4 ALET
72	Operand-4 Address	

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the third operand is replaced by the fourth operand, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Function Codes 4-7 (Compare and Swap)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-73 on page 7-148.

The parameter list used for function code 5 has the following format:

Parameter List for Function Code 5

0	
8	Operand-1 Comparison Value
16	
24	Operand-1 Replacement Value

The parameter list used for function code 7 has the following format:

Parameter List for Function Code 7

0	Operand-1 Comparison Value
8	Operand-1 Comp. Value (continued)
16	Operand-1 Replacement Value
24	Operand-1 Repl. Value (continued)

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Function Codes 8-11 (Double Compare and Swap)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-73 on page 7-148.

The parameter list used for function code 9 has the following format:

Parameter List for Function Code 9

0		
8	Operand-1 Comparison Value	
16		
24	Operand-1 Replacement Value	
32		
40	Operand-3 Comparison Value	
48		
56	Operand-3 Replacement Value	
64		Operand-4 ALET
72	Operand-4 Address	

The parameter list used for function code 11 has the following format:

Parameter List for Function Code 11

0	Operand-1 Comparison Value	
8	Operand-1 Comp. Value (continued)	
16	Operand-1 Replacement Value	
24	Operand-1 Repl. Value (continued)	
32	Operand-3 Comparison Value	
40	Operand-3 Comp. Value (continued)	
48	Operand-3 Replacement Value	
56	Operand-3 Repl. Value (continued)	
64		Operand-4 ALET
72	Operand-4 Address	

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the third-operand comparison value is compared to the fourth operand. When the third-operand comparison value is equal to the fourth operand (after the first-operand comparison value has been found equal to the second operand), the first-operand replacement value is stored at the second-operand location, the third-operand replacement value is stored at the fourth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

When the third-operand comparison value is not equal to the fourth operand (after the first-operand comparison value has been found equal to the second operand), the third-operand comparison value is replaced by the fourth operand, and condition code 2 is set.

Function Codes 12-15 (Compare and Swap and Store)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-73 on page 7-148.

The parameter list used for function code 13 has the following format:

Parameter List for Function Code 13

0		
8	Operand-1 Comparison Value	
16		
24	Operand-1 Replacement Value	
32		
40		
48		
56	Operand 3	
64		Operand-4 ALET
72	Operand-4 Address	

The parameter list used for function code 15 has the following format:

Parameter List for Function Code 15

0	Operand-1 Comparison Value	
8	Operand-1 Comp. Value (continued)	
16	Operand-1 Replacement Value	
24	Operand-1 Repl. Value (continued)	
32		
40		
48	Operand 3	
56	Operand 3 (continued)	
64		Operand-4 ALET
72	Operand-4 Address	

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Function Codes 16-19 (Compare and Swap and Double Store)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-73 on page 7-148.

The parameter list used for function code 16 has the following format:

Parameter List for Function Code 16

0		
8		
16		
24		
32		
40		
48		
56		Operand 3
64		Operand-4 ALET
72	Operand-4 Address	
80		
88		Operand 5
96		Operand-6 ALET
104	Operand-6 Address	

The parameter list used for function code 17 has the following format:

Parameter List for Function Code 17

0		
8	Operand-1 Comparison Value	
16		
24	Operand-1 Replacement Value	
32		
40		
48		
56	Operand 3	
64		Operand-4 ALET
72	Operand-4 Address	
80		
88	Operand 5	
96		Operand-6 ALET
104	Operand-6 Address	

The parameter list used for function code 18 has the following format:

Parameter List for Function Code 18

0		
8		
16		
24		
32		
40		
48		
56	Operand 3	
64		Operand-4 ALET
72	Operand-4 Address	
80		
88	Operand 5	
96		Operand-6 ALET
104	Operand-6 Address	

The parameter list used for function code 19 has the following format:

Parameter List for Function Code 19

0	Operand-1 Comparison Value	
8	Operand-1 Comp. Value (continued)	
16	Operand-1 Replacement Value	
24	Operand-1 Repl. Value (continued)	
32		
40		
48	Operand 3	
56	Operand 3 (continued)	
64		Operand-4 ALET
72	Operand-4 Address	
80	Operand 5	
88	Operand 5 (continued)	
96		Operand-6 ALET
104	Operand-6 Address	

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, the fifth operand is stored at the sixth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Function Codes 20-23 (Compare and Swap and Triple Store)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-73 on page 7-148.

The parameter list used for function code 20 has the following format:

Parameter List for Function Code 20

0		
8		
16		
24		
32		
40		
48		
56		Operand 3
64		Operand-4 ALET
72	Operand-4 Address	
80		
88		Operand 5
96		Operand-6 ALET
104	Operand-6 Address	
112		
120		Operand 7
128		Operand-8 ALET
136	Operand-8 Address	

The parameter list used for function code 21 has the following format:

Parameter List for Function Code 21

0		
8	Operand-1 Comparison Value	
16		
24	Operand-1 Replacement Value	
32		
40		
48		
56	Operand 3	
64		Operand-4 ALET
72	Operand-4 Address	
80		
88	Operand 5	
96		Operand-6 ALET
104	Operand-6 Address	
112		
120	Operand 7	
128		Operand-8 ALET
136	Operand-8 Address	

The parameter list used for function code 22 has the following format:

Parameter List for Function Code 22

0		
8		
16		
24		
32		
40		
48		
56	Operand 3	
64		Operand-4 ALET
72	Operand-4 Address	
80		
88	Operand 5	
96		Operand-6 ALET
104	Operand-6 Address	
112		
120	Operand 7	
128		Operand-8 ALET
136	Operand-8 Address	

The parameter list used for function code 23 has the following format:

Parameter List for Function Code 23

0	Operand-1 Comparison Value	
8	Operand-1 Comp. Value (continued)	
16	Operand-1 Replacement Value	
24	Operand-1 Repl. Value (continued)	
32		
40		
48	Operand 3	
56	Operand 3 (continued)	
64		Operand-4 ALET
72	Operand-4 Address	
80	Operand 5	
88	Operand 5 (continued)	
96		Operand-6 ALET
104	Operand-6 Address	
112	Operand 7	
120	Operand 7 (continued)	
128		Operand-8 ALET
136	Operand-8 Address	

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, the fifth operand is stored at the sixth-operand location, the seventh operand is stored at the eighth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Locking

A lock is obtained at the beginning of the operation and released at the end of the operation. The lock obtained is represented by a program lock token (PLT) whose logical address is specified in general register 1 as already described.

A PLT is a value produced by a model-dependent transformation of the PLT logical address. Depending on the model, the PLT may be derived directly from the PLT logical address or, when DAT is on, from the real address that results from transformation of the PLT logical address by DAT. If DAT is used, access-register translation (ART) precedes DAT in the access-register mode.

A PLT selects one of a model-dependent number of locks within the configuration. Programs being executed by different CPUs can be assured of specifying the same lock only by specifying PLT logical addresses that are the same and that can be transformed to the same real address by the different CPUs.

Since a model may or may not use ART and DAT when forming a PLT, access-exception conditions that can be encountered during ART and DAT may or may not be recognized as exceptions. There is no accessing of a location designated by a PLT, but an addressing exception may be recognized for the location. A protection exception is not recognized for any reason during processing of a PLT logical address.

The CPU can hold one lock at a time.

When PERFORM LOCKED OPERATION is executed by this CPU and is to use a lock that is already held by another CPU due to the execution of a PERFORM LOCKED OPERATION instruction by the other CPU, the execution by this CPU is delayed until the lock is no longer held. An excessive delay can be caused only by a machine malfunction and is a machine-check condition.

The order in which multiple requests for the same lock are satisfied is undefined.

A nonrecoverable failure of a CPU while holding a lock may result in a machine check, entry into the check-stop state, or system check stop. The machine check is processing backup if all operands are undamaged or processing damage if register operands are damaged. If a machine check or the check-stop state is the result, either no storage operands have been changed or else all storage operands that were due to be changed have been correctly changed, and, in either case, the lock has been released. If the storage operands are not in either their correct original state or their correct final state, the result is system check stop.

Storage-Operand References

The accesses to the even-numbered storage operands appear to be word concurrent, as observed by other CPUs, for function codes that are a multiple of 4 or doubleword concurrent for function codes that are one, 2, or 3 more than a multiple of 4. The accesses to the doublewords in the parameter list appear to be doubleword concurrent, as observed by other CPUs, regardless of the function code.

As observed by other CPUs, all storage operands may be tested for access exceptions before a lock is obtained. (A channel program cannot observe a lock.)

As observed by other CPUs, in all operations except the compare-and-swap operation (which does not have a fourth operand), the fourth operand is accessed while the lock is held only if a comparison of the first-operand comparison value to the second operand while the lock is held has indicated equality. In these operations, the fourth operand is accessed before the lock is held only if a comparison of the first-operand comparison value to the second operand has indicated equality and only if, when DAT is on, an INVALIDATE PAGE TABLE ENTRY instruction executed by another CPU after the fetch of the second operand will not be the cause of a page-translation exception recognized for the fourth operand, which it will if it sets to one the page-invalid bit in the page-table entry for the fourth operand when this CPU does not have a TLB entry corresponding to that page-table entry. In the compare-and-swap-and-double-store and compare-and-swap-and-triple-store operations, the sixth operand, and also the eighth operand in the triple-store operation, are treated the same as the fourth operand described above. The reason for this specification about INVALIDATE PAGE TABLE ENTRY is given in programming note 6 on page 7-157.

Provided that accessing of an operand is not prohibited as described in the preceding paragraph, store-type access exceptions may be recognized for the operand even when a store does not occur because of the results of a comparison. A storage-alteration PER event is recognized, and a change bit is set, only if a store occurs.

When a comparison is made between an operand comparison value and an operand before the lock is obtained and indicates inequality, the lock still is obtained. The condition code is set only as a result of a comparison made while the lock is held. When condition code 1 or 2 is set, the first-operand comparison value or third-operand comparison value is replaced only by means of a fetch of the second operand or fourth operand, respectively, made while the lock is held, as observed by other CPUs.

In those cases when a store is performed to the second-operand location and one or more of the fourth-, sixth-, and eighth-operand locations, the store to the second-operand location is always performed last, as observed by other CPUs and by channel programs.

Stores into the parameter list may be performed while the lock is held or after it has been released.

A serialization operation is performed immediately after the lock is obtained and again immediately before it is released. However, values fetched from the parameter list before the lock is obtained are not necessarily refetched. A serialization operation is not performed if the test bit, bit 55 of general register 0, is one.

Access exceptions may be recognized for parameter-list locations even when the locations are not required in the operation. The locations are those beginning at offset 0 and extending up through the last location defined for the function code used.

For the compare-and-load and compare-and-swap operations, the operation is suppressed on all addressing and protection exceptions.

When a nonrecoverable failure of a CPU while holding a lock results in a machine check or entry into the check-stop state, either no storage operands have been changed or else all storage operands that were due to be changed have been correctly changed. The latter may be accomplished by repeating stores that were performed successfully before the failure. Therefore, there may be two single-access store references (possibly the store part of an update reference and then a store reference) to the store-type operands, with the first value stored equal to the second value stored.

Resulting Condition Code:

When test bit is zero:

0 All comparisons equal; replacement value or values stored or loaded
1 First-operand comparison not equal; first-operand comparison value replaced
2 -- (all operations except double compare and swap)
2 First-operand comparison equal but third-operand comparison not equal; third-operand comparison value replaced (double compare and swap)
3 --

When test bit is one:

0 Function code valid
1 --
2 --
3 Function code invalid

Program Exceptions:

- Access (for all function codes, fetch, except addressing and protection for PLT location, program lock token, model-dependent; for all function codes, fetch and store, operand 2; for odd-numbered function codes, fetch and store, parameter list; for function codes 16, 18, 20, and 22, fetch, parameter list; for function codes 0-3, fetch, operand 4; for function codes 8-11, fetch and store, operand 4; for function codes 12-23, store, operand 4; for function codes 16-23, store, operand 6; for function codes 20-23, store, operand 8)
- Specification

Programming Notes:

1. An example of the use of the PERFORM LOCKED OPERATION instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When the contents of storage locations are changed by PERFORM LOCKED OPERATION instructions that are executed concurrently by different CPUs and that use the same lock, the changes to operands not in the parameter list will be completed by one of the CPUs before they are begun by the other CPU, depending on which CPU first obtains the lock.

3. The compare-and-swap functions of PERFORM LOCKED OPERATION are not performed by means of interlocked-update references. Concurrent store references by another CPU to the storage operands, even if they are interlocked-update references, will interfere unpredictably, in terms of the resulting register and storage contents, with the intended operation of PERFORM LOCKED OPERATION. All changes to the contents of the storage locations must be made by PERFORM LOCKED OPERATION instructions that use the same lock, if predictable storage results are to be obtained.

4. Because a nonrecoverable failure of a CPU while executing PERFORM LOCKED OPERATION may cause two stores of the same value to a store-type operand, a concurrent store made by another CPU to the same operand but not by executing PERFORM LOCKED OPERATION may be lost.

5. When programs in different address spaces are using the same lock when DAT is on, the programs must ensure that they are using PLT logical addresses that are the same and that will be translated to the same real address regardless of the address space in which a translation occurs. Otherwise, the programs may in fact use different locks.

6. The section “Storage-Operand References” on page 7-156 contains a specification concerning the INVALIDATE PAGE TABLE ENTRY (IPTE) instruction. The need for the specification is shown by the following example that is possible without the specification.

   a. CPU 1 begins to execute a PERFORM LOCKED OPERATION instruction with function code 8, which is referred to as PLO.DCS. Operand 2 is a location, Qtail, containing the address (the first-operand comparison value) of the last element, element X, on a queue, and operand 4 is a location in that element containing the address (0, the third-operand comparison value) of the next (nonexisting) element on the queue. The purpose of the PLO instruction is to enqueue an element by placing the address of the element (the first-operand and third-operand replacement values) in both operand 2 and operand 4. With the lock not held, the PLO instruction fetches operand 2 and compares it, with an equal result, to the first-operand comparison value.

b. CPU 2 completely executes a PLO.DCS instruction to dequeue element X, which is the only element on the queue, from the queue. The PLO instruction stores 0 in Qtail and also in Qhead, which is a location containing the address of the first element on the queue. The program on CPU 2 processes the dequeued element and then invokes the freemain service of the control program to deallocate the storage containing the element. The freemain service uses IPTE to set the page-invalid bit to one in the page-table entry for the page containing element X. The IPTE instruction immediately sets the page-invalid bit to one, and then it waits for the signal that all other CPUs have cleared their TLBs of entries corresponding to the page.

c. CPU 1 attempts to fetch operand 4. CPU 1 does not have a TLB entry for the operand-4 page. CPU 1 signals CPU 2 that the CPU 2 IPTE instruction may proceed.

d. CPU 2 completes its IPTE instruction. The program on CPU 2 sets a software bit in the page-table entry to one to indicate that the page has been freemained and that, therefore, a reference to the page should result in presentation by the control program of an addressing exception to the program making the reference.

e. CPU 1 attempts to do DAT for operand 4 and sees that the page-invalid bit is one. CPU 1 performs a program interruption indicating a page-translation exception. The exception handler sees that the software bit indicating freemained is one, and it presents an addressing exception to the CPU 1 program, which causes an abend of the program.

If CPU 1 had had a TLB entry for the page, its PLO instruction would not have been interrupted, and the comparison of the first-operand comparison value to the second operand while the lock was held would indicate that CPU 2 had changed the second operand. The PLO instruction would set condition code 1. If CPU 1 did not have a TLB entry but IPTE could not set the page-invalid bit to one while CPU 1 was executing an instruction, CPU 1 could successfully translate the operand-4 address and, again, discover while the lock was held that operand 2 had changed. The case when operand 2 points to element X but the freemained bit for the element-X page is one is a programming error.

7. Figure 7-74 on page 7-159 summarizes the results of the operation.

Op1c=Op2	Op3c=Op4	Cond Code	Action
Function Codes 0-3 (Compare and Load)			
No	-	1	Op2 → Op1c
Yes	-	0	Op4 → Op3
Function Codes 4-7 (Compare and Swap)			
No	-	1	Op2 → Op1c
Yes	-	0	Op1r → Op2
Function Codes 8-11 (Double Compare and Swap)			
No	-	1	Op2 → Op1c
Yes	No	2	Op4 → Op3c
Yes	Yes	0	Op1r → Op2 Op3r → Op4
Function Codes 12-15 (Compare and Swap and Store)			
No	-	1	Op2 → Op1c
Yes	-	0	Op1r → Op2 Op3 → Op4
Function Codes 16-19 (Compare and Swap and Double Store)			
No	-	1	Op2 → Op1c
Yes	-	0	Op1r → Op2 Op3 → Op4 Op5 → Op6
Function Codes 20-23 (Compare and Swap and Triple Store)			
No	-	1	Op2 → Op1c
Yes	-	0	Op1r → Op2 Op3 → Op4 Op5 → Op6 Op7 → Op8

Explanation:

-	Not applicable.
OpNc	Operand-N comparison value.
OpNr	Operand-N replacement value.

Figure 7-74. Summary of PERFORM LOCKED OPERATION Results

ROTATE LEFT SINGLE LOGICAL

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

 ┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
 │  'EB'   │ R1 │ R3 │ B2 │DL2 │   DH2   │  '1D'   │
 └─────────┴────┴────┴────┴────┴─────────┴─────────┘
 0         8   12   16   20   32        40        47

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

 ┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
 │  'EB'   │ R1 │ R3 │ B2 │DL2 │   DH2   │  '1C'   │
 └─────────┴────┴────┴────┴────┴─────────┴─────────┘
 0         8   12   16   20   32        40        47
```

The 32-bit or 64-bit third operand is rotated left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R_3. For ROTATE LEFT SINGLE LOGICAL (RLL), bits 0-31 of general registers R_1 and R_3 remain unchanged.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be rotated. The remainder of the address is ignored.

For RLL, the first and third operands are in bit positions 32-63 of general registers R_1 and R_3, respectively. For RLLG, the operands are in bit positions 0-63 of the registers.

All 32 or 64 bits of the third operand participate in a left shift. Each bit shifted out of the leftmost bit position of the operand reenters in the rightmost bit position of the operand.

Condition Code: The code remains unchanged.

Program Exceptions: None.

SEARCH STRING

```
SRST     R₁,R₂      [RRE]
```

'B25E'	////////	R_1	R_2
0	16	24	28 31

The second operand is searched until a specified character is found, the end of the second operand is reached, or a CPU-determined number of bytes have been searched, whichever occurs first. The CPU-determined number is at least 256. The result is indicated in the condition code.

The location of the leftmost byte of the second operand is designated by the contents of general register R_2. The location of the first byte after the second operand is designated by the contents of general register R_1.

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.

In the access-register mode, the address space containing the second operand is specified only by means of access register R_2. The contents of access register R_1 are ignored.

The character for which the search occurs 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 specified character has been found in the second operand, the address of the next second-operand byte to be examined equals the address in general register R_1, or a CPU-determined number of second-operand bytes have been examined, whichever occurs first. The CPU-determined number is at least 256. When the specified character is found, condition code 1 is set. When the address of the next second-operand byte to be examined equals the address in general register R_1, condition code 2 is set. When a CPU-determined number of second-operand bytes have been examined, condition code 3 is set. When the CPU-determined number of second-operand bytes have been examined and the address of the next second-operand byte is in general register R_1, it is unpredictable whether condition code 2 or 3 is set.

When condition code 1 is set, the address of the specified character found in the second 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 second operand is placed in general register R_2, and the contents of general register R_1 remain unchanged. When condition code 2 is set, the contents of general registers R_1 and R_2 remain unchanged. 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 second operand are recognized only for that portion of the operand that is necessarily examined.

The storage-operand-consistency rules are the same as for the COMPARE LOGICAL LONG instruction.

Resulting Condition Code:

0 --
1 Specified character found; general register R_1 updated with address of character; general register R_2 unchanged
2 Specified character not found in entire second operand; general registers R_1 and R_2 unchanged
3 CPU-determined number of bytes searched; general register R_1 unchanged; general register R_2 updated with address of next byte

Program Exceptions:

- Access (fetch, operand 2)
- Specification

Programming Notes:

1. Examples of the use of the SEARCH STRING instruction are 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 search. The program need not determine the number of bytes that were searched.

3. When the address in general register R_1 equals the address in general register R_2, condition code 2 is set immediately, and access exceptions are not recognized. When the address in general register R_1 is less than the address in general register R_2, condition code 2 can be set only if the operand wraps around from the top of storage to location 0.

4. R_1 or R_2 may be zero, in which case general register 0 is treated as containing an address and also the specified character.

5. When it is desired to search a string of unknown length for its ending character, and assuming that (1) the string does not start below location 256 (or below location 1 if the ending character is 00 hex), (2) the string does not wrap around to location 0, and (3) the specified character in general register 0 need not be preserved, then R_1 can be zero in order to have SEARCH STRING use only two general registers instead of three.

SET ACCESS

```
SAR     R1,R2     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B24E'     │////////│ R1 │ R2 │
└────────────────┴────────┴────┴────┘
0                16       24   28  31
```

The contents of bit positions 32-63 of general register R_2 are placed in access register R_1.

Condition Code: The code remains unchanged.

Program Exceptions: None.

SET ADDRESSING MODE

```
SAM24     [E]

┌────────────────┐
│     '010C'     │
└────────────────┘
0               15

SAM31     [E]

┌────────────────┐
│     '010D'     │
└────────────────┘
0               15

SAM64     [E]

┌────────────────┐
│     '010E'     │
└────────────────┘
0               15
```

The addressing mode is set by setting the extended-addressing-mode bit, bit 31 of the current PSW, and the basic-addressing-mode bit, bit 32 of the current PSW, as follows:

Instruction	PSW Bit 31	PSW Bit 32	Resulting Addressing Mode
SAM24	0	0	24-bit
SAM31	0	1	31-bit
SAM64	1	1	64-bit

The instruction address in the PSW is updated under the control of the new addressing mode, as follows. The value 2 (the instruction length) is added to the contents of bit positions 64-127 of the PSW, or the value 4 is added if the instruction is the target of EXECUTE. In either case, a carry out of bit position 0 is ignored. Then bits 64-103 of the PSW are set to zeros if the new addressing mode is the 24-bit mode, or bits 64-96 are set to zeros if the new addressing mode is the 31-bit mode.

The instruction is completed only if the new addressing mode and the unupdated instruction address in the PSW are a valid combination. When the new addressing mode is to be the 24-bit mode, bits 64-103 of the unupdated PSW must be all zeros, or, when the new addressing mode is to be the 31-bit mode, bits 64-96 of the unupdated PSW must be all zeros; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Specification (SAM24 and SAM31 only)
- Trace

Programming Note: Checking the unupdated instruction address prevents completion in two major cases: the instruction is located at address 2^{24} or above and the new addressing mode is to be the 24-bit mode, or the instruction is located at address 2^{31} or above and the new addressing mode is to be the 24-bit or 31-bit mode. In these cases, if the instruction were completed, the updating of the instruction address under the control of the new addressing mode would cause one or more leftmost bits of the address to be set to zeros, which would cause the next instruction to be fetched from other than the next sequential location. This action is sometimes called a “wild branch.” A wild branch still can occur if the instruction is located at $2^{24} - 2$ or $2^{31} - 2$, or at $2^{24} - 4$ or $2^{31} - 4$ if EXECUTE is used.

SET PROGRAM MASK

SPM R_1 [RR]

'04'	R_1	////
0	8	12 15

The first operand is used to set the condition code and the program mask of the current PSW.

Bits 34 and 35 of general register R_1 replace the condition code, and bits 36-39 replace the program mask. Bits 0-33 and 40-63 of general register R_1 are ignored.

Condition Code: The code is set as specified by bits 34 and 35 of general register R_1.

Program Exceptions: None.

Programming Notes:

1. Bits 34-39 of the general register may have been loaded from the PSW by execution of BRANCH AND LINK in the 24-bit addressing mode or by execution of INSERT PROGRAM MASK in either the 24-bit or 31-bit addressing mode.
2. SET PROGRAM MASK permits setting of the condition code and the mask bits in either the problem state or the supervisor state.
3. The program should take into consideration that the setting of the program mask can have a significant effect on subsequent execution of the program. Not only do the four mask bits control whether the corresponding interruptions occur, but the exponent-underflow and significance masks also determine the result which is obtained.

SHIFT LEFT DOUBLE

SLDA $R_1,D_2(B_2)$ [RS]

'8F'	R_1	////	B_2	D_2
0	8	12	16	20 31

The 63-bit numeric part of the signed first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register R_1 followed on the right by bits 32-63 of general register $R_1 + 1$.

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

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

The first operand is treated as a 64-bit signed binary integer. The sign bit of the first operand, bit 32 of the even-numbered register, remains unchanged. Bit position 32 of the odd-numbered register contains a numeric bit, which participates in the shift in the same manner as the other numeric bits. Zeros are supplied to the vacated bit positions on the right. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged.

If one or more bits unlike the sign bit are shifted out of bit position 33 of the even-numbered register, an overflow occurs, and condition code 3 is set. If the fixed-point-overflow mask bit 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
- Specification

Programming Notes:

1. An example of the use of the SHIFT LEFT DOUBLE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The eight shift instructions that are in both ESA/390 and z/Architecture provide the following three pairs of alternatives for 32 bits in one general register or, for double, in each of two general registers: left or right, single or double, and signed or logical. The four additional shift instructions in z/Architecture provide left or right, signed or logical shifts of 64 bits in one general register. The signed shifts differ from the logical shifts in that, in the signed shifts, overflow is recognized, the condition code is set, and the leftmost bit participates as a sign.
3. A zero shift amount in the two signed double-shift operations provides a double-length sign and magnitude test.
4. The base register participating in the generation of the second-operand address permits indirect specification of the shift amount by means of placement of the shift amount in the base register. A zero in the B_2 field indicates the absence of indirect shift specification.

SHIFT LEFT DOUBLE LOGICAL

```
SLDL   R1,D2(B2)        [RS]

┌─────────┬────┬────┬────┬────────────┐
│  '8D'   │ R1 │////│ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8    12   16   20          31
```

The 64-bit first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register R_1 followed on the right by bits 32-63 of general register $R_1 + 1$.

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

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

All 64 bits of the first operand participate in the shift. Bits shifted out of bit position 32 of the even-numbered register are not inspected and are lost. Zeros are supplied to the vacated bit positions on the right. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:

- Specification

SHIFT LEFT SINGLE

```
SLA    R1,D2(B2)        [RS]

┌─────────┬────┬────┬────┬────────────┐
│  '8B'   │ R1 │////│ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8    12   16   20          31
```

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

                              /
  ┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
| │  'EB'   │ R1 │ R3 │ B2 │DL2 │   DH2   │  '0B'   │
  └─────────┴────┴────┴────┴────┴─────────┴─────────┘
                              /
  0         8    12   16   20   32        40       47
```

For SHIFT LEFT SINGLE (SLA), the 31-bit numeric part of the signed first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R_1 remain unchanged.

For SHIFT LEFT SINGLE (SLAG), the 63-bit numeric part of the signed third operand is shifted left the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. The third operand remains unchanged in general register R_3.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SLA, the first operand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R_1. The sign of the first operand remains unchanged. All 31 numeric bits of the operand participate in the left shift.

For SLAG, the first and third operands are treated as 64-bit signed binary integers in bit positions 0-63 of general registers R_1 and R_3, respectively. The sign of the first operand is set equal to the sign of the third operand. All 63 numeric bits of the third operand participate in the left shift.

For SLA or SLAG, zeros are supplied to the vacated bit positions on the right.

If one or more bits unlike the sign bit are shifted out of bit position 33, for SLA, or 1, for SLAG, an overflow occurs, and condition code 3 is set. If the fixed-point-overflow mask bit 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:

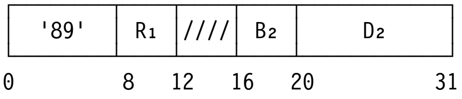
- Fixed-point overflow

Programming Notes:

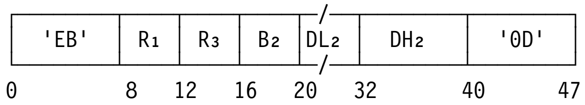
1. An example of the use of the SHIFT LEFT SINGLE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. For SHIFT LEFT SINGLE (SLA), for numbers with a value greater than or equal to -2^{30} and less than 2^{30}, a left shift of one bit position is equivalent to multiplying the number by 2. For SHIFT LEFT SINGLE (SLAG), the comparable values are -2^{62} and 2^{62}.
3. For SHIFT LEFT SINGLE (SLA), shift amounts from 31 to 63 cause the entire numeric part to be shifted out of the register, leaving a result of the maximum negative number or zero, depending on whether or not the initial contents were negative. For SHIFT LEFT SINGLE (SLAG), a shift amount of 63 causes the same effect.

SHIFT LEFT SINGLE LOGICAL

```
SLL    R₁,D₂(B₂)        [RS]
```



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



For SHIFT LEFT SINGLE LOGICAL (SLL), the 32-bit first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R_1 remain unchanged.

For SHIFT LEFT SINGLE LOGICAL (SLLG), the 64-bit third operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R_3.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SLL, the first operand is in bit positions 32-63 of general register R_1. All 32 bits of the operand participate in the left shift.

For SLLG, the first and third operands are in bit positions 0-63 of general registers R_1 and R_3, respectively. All 64 bits of the third operand participate in the left shift.

For SLL or SLLG, zeros are supplied to the vacated bit positions on the right.

Condition Code: The code remains unchanged.

Program Exceptions: None.

SHIFT RIGHT DOUBLE

SRDA $R_1,D_2(B_2)$ [RS]

'8E'	R_1	////	B_2	D_2	
0	8	12	16	20	31

The 63-bit numeric part of the signed first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register R_1 followed on the right by bits 32-63 of general register $R_1 + 1$.

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

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

The first operand is treated as a 64-bit signed binary integer. The sign bit of the first operand, bit 32 of the even-numbered register, remains unchanged. Bit position 32 of the odd-numbered register contains a numeric bit, which participates in the shift in the same manner as the other numeric bits. Bits shifted out of bit position 63 of the odd-numbered register are not inspected and are lost. Bits equal to the sign are supplied to the vacated bit positions on the left. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged.

Resulting Condition Code:

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

Program Exceptions:

- Specification

SHIFT RIGHT DOUBLE LOGICAL

SRDL $R_1,D_2(B_2)$ [RS]

'8C'	R_1	////	B_2	D_2	
0	8	12	16	20	31

The 64-bit first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register R_1 followed on the right by bits 32-63 of general register $R_1 + 1$.

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

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

All 64 bits of the first operand participate in the shift. Bits shifted out of bit position 63 of the odd-numbered register are not inspected and are lost. Zeros are supplied to the vacated bit positions on the left. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:

- Specification

SHIFT RIGHT SINGLE

```
SRA    R1,D2(B2)         [RS]

┌─────────┬────┬────┬────┬────────────┐
│  '8A'   │ R1 │////│ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8    12   16   20          31
```

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

┌─────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'   │ R1 │ R3 │ B2 │DL2 │  DH2   │  '0A'  │
└─────────┴────┴────┴────┴────┴────────┴────────┘
0         8    12   16   20   32       40      47
```

For SHIFT RIGHT SINGLE (SRA), The 31-bit numeric part of the signed first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-32 of general register R$_1$ remain unchanged.

For SHIFT RIGHT SINGLE (SRAG), the 63-bit numeric part of the signed third operand is shifted right the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. The third operand remains unchanged in general register R$_3$.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SRA, The first operand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R$_1$. The sign of the first operand remains unchanged. All 31 numeric bits of the operand participate in the right shift.

For SRAG, the first and third operands are treated as 64-bit signed binary integers in bit positions 0-63 of general registers R$_1$ and R$_3$, respectively. The sign of the first operand is set equal to the sign of the third operand. All 63 numeric bits of the third operand participate in the right shift.

For SRA or SRAG, bits shifted out of bit position 63 are not inspected and are lost. Bits equal to the sign are supplied to the vacated bit positions on the left.

Resulting Condition Code:

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

Program Exceptions: None.

Programming Notes:

1. A right shift of one bit position is equivalent to division by 2 with rounding downward. When an even number is shifted right one position, the result is equivalent to dividing the number by 2. When an odd number is shifted right one position, the result is equivalent to dividing the *next lower* number by 2. For example, +5 shifted right by one bit position yields +2, whereas -5 yields -3.
2. For SHIFT RIGHT SINGLE (SRA), shift amounts from 31 to 63 cause the entire numeric part to be shifted out of the register, leaving a result of -1 or zero, depending on whether or not the initial contents were negative. For SHIFT RIGHT SINGLE (SRAG), a shift amount of 63 causes the same effect.

SHIFT RIGHT SINGLE LOGICAL

```
SRL    R1,D2(B2)         [RS]

┌─────────┬────┬────┬────┬────────────┐
│  '88'   │ R1 │////│ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8    12   16   20          31
```

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

┌─────────┬────┬────┬────┬────┬────────┬────────┐
│  'EB'   │ R1 │ R3 │ B2 │DL2 │  DH2   │  '0C'  │
└─────────┴────┴────┴────┴────┴────────┴────────┘
0         8    12   16   20   32       40      47
```

For SHIFT RIGHT SINGLE LOGICAL (SRL), the 32-bit first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R$_1$ remain unchanged.

For SHIFT RIGHT SINGLE LOGICAL (SRLG), the 64-bit third operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R$_3$.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SRL, the first operand is in bit positions 32-63 of general register R_1. All 32 bits of the operand participate in the right shift.

For SRLG, the first and third operands are in bit positions 0-63 of general registers R_1 and R_3, respectively. All 64 bits of the third operand participate in the right shift.

For SRL or SRLG, bits shifted out of bit position 63 are not inspected and are lost. Zeros are supplied to the vacated bit positions on the left.

Condition Code: The code remains unchanged.

Program Exceptions: None.

STORE

ST $R_1,D_2(X_2,B_2)$ [RX]

```
┌─────────┬────┬────┬────┬────────────┐
│  '50'   │ R1 │ X2 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8    12   16   20           31
```

STY $R_1,D_2(X_2,B_2)$ [RXY]

```
┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '50'   │
└─────────┴────┴────┴────┴────┴─────────┴─────────┘
0         8    12   16   20   32        40       47
```

STG $R_1,D_2(X_2,B_2)$ [RXY]

```
┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
│  'E3'   │ R1 │ X2 │ B2 │DL2 │   DH2   │  '24'   │
└─────────┴────┴────┴────┴────┴─────────┴─────────┘
0         8    12   16   20   32        40       47
```

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

For STORE (ST, STY), the operands are 32 bits, and, for STORE (STG), the operands are 64 bits.

The displacement for ST is treated as a 12-bit unsigned binary integer. The displacement for STY and STG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

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

STORE ACCESS MULTIPLE

STAM $R_1,R_3,D_2(B_2)$ [RS]

```
┌─────────┬────┬────┬────┬────────────┐
│  '9B'   │ R1 │ R3 │ B2 │     D2     │
└─────────┴────┴────┴────┴────────────┘
0         8    12   16   20           31
```

STAMY $R_1,D_2(X_2,B_2)$ [RSY]

```
┌─────────┬────┬────┬────┬────┬─────────┬─────────┐
│  'EB'   │ R1 │ R3 │ B2 │DL2 │   DH2   │  '9B'   │
└─────────┴────┴────┴────┴────┴─────────┴─────────┘
0         8    12   16   20   32        40       47
```

The contents of the set of access registers starting with access register R_1 and ending with access register R_3 are stored at the locations designated by the second-operand address.

The storage area where the contents of the access registers are placed 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 contents of the access registers are stored 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 contents of the access registers remain unchanged.

The displacement for STAM is treated as a 12-bit unsigned binary integer. The displacement for STAMY 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 (store, operand 2)
- Operation (STAMY, if the long-displacement facility is not installed)
- Specification

STORE CHARACTER

STC $R_1,D_2(X_2,B_2)$ [RX]

'42'	R_1	X_2	B_2	D_2
0	8	12	16	20 31

STCY $R_1,D_2(X_2,B_2)$ [RXY]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'72'
0	8	12	16	20	32	40 47

Bits 56-63 of general register R_1 are placed unchanged at the second-operand location. The second operand is one byte in length.

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

Condition Code: The code remains unchanged.

Program Exceptions:

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

STORE CHARACTERS UNDER MASK

STCM $R_1,M_3,D_2(B_2)$ [RS]

'BE'	R_1	M_3	B_2	D_2
0	8	12	16	20 31

STCMY $R_1,M_3,D_2(B_2)$ [RSY]

'EB'	R_1	M_3	B_2	DL_2	DH_2	'2D'
0	8	12	16	20	32	40 47

STCMH $R_1,M_3,D_2(B_2)$ [RSY]

'EB'	R_1	M_3	B_2	DL_2	DH_2	'2C'
0	8	12	16	20	32	40 47

Bytes selected from general register R_1 under control of a mask are placed at contiguous byte locations beginning at the second-operand address.

The contents of the M_3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R_1. For STORE CHARACTERS UNDER MASK (STCM, STCMY), the four bytes to which the mask bits correspond are in bit positions 32-63 of general register R_1. For STORE CHARACTERS UNDER MASK (STCMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The bytes corresponding to ones in the mask are placed in the same order at successive and contiguous storage locations beginning at the second-operand address. When the mask is not zero, the length of the second operand is equal to the number of ones in the mask. The contents of the general register remain unchanged.

When the mask is not zero, exceptions associated with storage-operand accesses are recognized only for the number of bytes specified by the mask.

When the mask is zero, the single byte designated by the second-operand address remains unchanged; however, on some models, the contents may be fetched and subsequently stored back unchanged at the same storage location. This update appears to be an interlocked-update reference as observed by other CPUs.

The displacement for STCM is treated as a 12-bit unsigned binary integer. The displacement for STCMY and STCMH is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

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

Programming Notes:

1. An example of the use of the STORE CHARACTERS UNDER MASK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. STORE CHARACTERS UNDER MASK (STCM, STCMY), with a mask of 0111 may

be used to store a three-byte address, for example, in modifying the address in a CCW.

3. STORE CHARACTERS UNDER MASK (STCM, STCMY) with a mask of 1111, 0011, or 0001 performs the same function as STORE (ST), STORE HALFWORD, or STORE CHARACTER, respectively. However, on most models, the performance of STORE CHARACTERS UNDER MASK is slower.
4. Using STORE CHARACTERS UNDER MASK with a zero mask should be avoided since this instruction, depending on the model, may perform a fetch and store of the single byte designated by the second-operand address. This reference is not interlocked against accesses by channel programs. In addition, it may cause any of the following to occur for the byte designated by the second-operand address: a PER storage-alteration event may be recognized; access exceptions may be recognized; and, provided no access exceptions exist, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.

STORE CLOCK

```
STCK   D2(B2)          [S]

┌───────────────┬────┬──────────┐
│    'B205'     │ B2 │    D2    │
└───────────────┴────┴──────────┘
0              16   20         31
```

The current value of bits 0-63 of the TOD clock is stored in the eight-byte field designated by the second-operand address, provided the clock is in the set, stopped, or not-set state.

When the clock is stopped, zeros are stored in positions to the right of the rightmost bit position that is incremented when the clock is running. When the value of a running clock is stored, nonzero values may be stored in positions to the right of the rightmost incremented bit; this is to ensure that a unique value is stored.

Zeros are stored for the rightmost bit positions that are not provided by the clock.

Zeros are stored at the operand location when the clock is in the error state or the not-operational state.

The quality of the clock value stored by the instruction is indicated by the resultant condition-code setting.

A serialization function is performed before the value of the clock is fetched and again after the value is placed in storage.

Resulting Condition Code:

0	Clock in set state
1	Clock in not-set state
2	Clock in error state
3	Clock in stopped state or not-operational state

Program Exceptions:

- Access (store, operand 2)

Programming Notes:

1. Bit position 31 of the clock is incremented every 1.048576 seconds; hence, for timing applications involving human responses, the leftmost clock word may provide sufficient resolution.
2. Condition code 0 normally indicates that the clock has been set by the control program. Accordingly, the value may be used in elapsed-time measurements and as a valid time-of-day and calendar indication. Condition code 1 indicates that the clock value is the elapsed time since the power for the clock was turned on. In this case, the value may be used in elapsed-time measurements but is not a valid time-of-day indication. Condition codes 2 and 3 mean that the value provided by STORE CLOCK cannot be used for time measurement or indication.
3. Condition code 3 indicates that the clock is in either the stopped state or the not-operational state. These two states can normally be distinguished because an all-zero value is stored when the clock is in the not-operational state.
4. If a problem program written for z/Architecture is to be executed also on a system in the System/370 mode, then the program should take into account that, in the System/370 mode, the value stored when the condition code is 2 is not necessarily zero.

STORE CLOCK EXTENDED

STCKE $D_2(B_2)$ [S]

'B278'	B_2	D_2
0	16	20 31

The current value of bits 0-103 of the TOD clock is stored in byte positions 1-13 of the sixteen-byte field designated by the second-operand address, provided the clock is in the set, stopped, or not-set state. Zeros are stored in byte position 0. The TOD programmable field, bits 16-31 of the TOD programmable register, is stored in byte positions 14 and 15.

The operand just described has the following format:

Zeros	TOD Clock	Programmable Field
0	8	112 127

When the clock is stopped, zeros are stored in the clock value in positions to the right of the rightmost bit position that is incremented when the clock is running. The programmable field still is stored.

When the value of a running clock is stored, the value in bit positions 64-103 of the clock (bit positions 72-111 of the storage operand) is always nonzero; this ensures that values stored by STORE CLOCK EXTENDED are unique when compared with values stored by STORE CLOCK and extended with zeros.

Zeros are stored at the operand location when the clock is in the error state or the not-operational state.

The quality of the clock value stored by the instruction is indicated by the resultant condition-code setting.

A serialization function is performed before the value of the clock is fetched and again after the value is placed in storage.

Resulting Condition Code:

0	Clock in set state
1	Clock in not-set state
2	Clock in error state
3	Clock in stopped state or not-operational state

Program Exceptions:

- Access (store, operand 2)

Programming Notes:

1. Condition code 0 normally indicates that the clock has been set by the control program. Accordingly, the value may be used in elapsed-time measurements and as a valid time-of-day and calendar indication. Condition code 1 indicates that the clock value is the elapsed time since the power for the clock was turned on. In this case, the value may be used in elapsed-time measurements but is not a valid time-of-day indication. Condition codes 2 and 3 mean that the value provided by STORE CLOCK EXTENDED cannot be used for time measurement or indication.

2. Programming notes beginning on page 4-39 show hex values related to the value of the TOD clock as it is stored by the STORE CLOCK instruction. Notes 3-5, below, are repetitions of those notes except with the text and hex values adjusted so they apply to bits 0-71 of the value stored by STORE CLOCK EXTENDED.

3. The following chart shows the time interval between instants at which various bits of the TOD-clock value stored by STORE CLOCK EXTENDED are stepped. This time value may also be considered as the weighted time value that the bit, when one, represents. The bit numbers are those of the STORE CLOCK EXTENDED operand.

STCKE Bit	Stepping Interval			
	Days	Hours	Min.	Seconds
59				0.000 001
55				0.000 016
51				0.000 256
47				0.004 096
43				0.065 536
39				1.048 576
35				16.777 216
31			4	28.435 456
27		1	11	34.967 296
23		19	5	19.476 736
19	12	17	25	11.627 776
15	203	14	43	6.044 416
11	3257	19	29	36.710 656

4. The following chart shows the setting of bits 0-63 of the STORE CLOCK EXTENDED operand for 00:00:00 (0 am), UTC time, for several dates: January 1, 1900, January 1, 1972, and for that instant in time just after each of the 22 leap seconds that have occurred through January, 1999. Each of these leap seconds was inserted in the UTC time scale beginning at 23:59:60 UTC of the day previous to the one listed and ending at 00:00:00 UTC of the day listed.

Year	Mth	Day	Leap Sec.	STCKE Value (Hex) Bits 0-63
1900	1	1		0000 0000 0000 0000
1972	1	1		0081 26D6 0E46 0000
1972	7	1	1	0082 0BA9 811E 2400
1973	1	1	2	0082 F300 AEE2 4800
1974	1	1	3	0084 BDE9 7114 6C00
1975	1	1	4	0086 88D2 3346 9000
1976	1	1	5	0088 53BA F578 B400
1977	1	1	6	008A 1FE5 9520 D800
1978	1	1	7	008B EACE 5752 FC00
1979	1	1	8	008D B5B7 1985 2000
1980	1	1	9	008F 809F DBB7 4400
1981	7	1	10	0092 305C 0FCD 6800
1982	7	1	11	0093 FB44 D1FF 8C00
1983	7	1	12	0095 C62D 9431 B000
1985	7	1	13	0099 5D40 F517 D400
1988	1	1	14	009D DA69 A557 F800
1990	1	1	15	00A1 717D 063E 1C00
1991	1	1	16	00A3 3C65 C870 4000
1992	7	1	17	00A5 EC21 FC86 6400
1993	7	1	18	00A7 B70A BEB8 8800
1994	7	1	19	00A9 81F3 80EA AC00
1996	1	1	20	00AC 3433 6FEC D000
1997	7	1	21	00AE E3EF A402 F400
1999	1	1	22	00B1 962F 9305 1800

5. The stepping value of TOD-clock bit position 63, if implemented, is 2^{-12} microseconds, or approximately 244 picoseconds. This value is called a clock unit.

   The following chart shows various time intervals in clock units expressed in hexadecimal notation. The chart shows the values stored in bit positions 0-71 of the STORE CLOCK EXTENDED operand. Bit 71 of the operand represents a clock unit.

Interval	Clock Units (Hex) Bits 0-71
1 microsecond	0010 00
1 millisecond	3E80 00
1 second	00F4 2400 00
1 minute	3938 7000 00
1 hour	000D 693A 4000 00
1 day	0141 DD76 0000 00
365 days	0001 CAE8 C13E 0000 00
366 days	0001 CC2A 9EB4 0000 00
1,461 days*	0007 2CE4 E26E 0000 00

* Number of days in four years, including a leap year. Note that the year 1900 was not a leap year. Thus, the four-year span starting in 1900 has only 1,460 days.

STORE HALFWORD

STH $R_1,D_2(X_2,B_2)$ [RX]

```
┌──────────┬─────┬─────┬─────┬──────────────┐
│   '40'   │ R1  │ X2  │ B2  │      D2      │
└──────────┴─────┴─────┴─────┴──────────────┘
0          8    12    16    20            31
```

STHY $R_1,D_2(X_2,B_2)$ [RXY]

```
┌──────────┬─────┬─────┬─────┬─────/─────────────┬──────────┐
│   'E3'   │ R1  │ X2  │ B2  │ DL2 │     DH2     │   '70'   │
└──────────┴─────┴─────┴─────┴─────/─────────────┴──────────┘
0          8    12    16    20    32            40         47
```

Bits 48-63 of general register R_1 are placed unchanged at the second-operand location. The second operand is two bytes in length.

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

Condition Code: The code remains unchanged.

Program Exceptions:

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

STORE MULTIPLE

STM $R_1,R_3,D_2(B_2)$ [RS]

```
┌──────────┬─────┬─────┬─────┬──────────────┐
│   '90'   │ R1  │ R3  │ B2  │      D2      │
└──────────┴─────┴─────┴─────┴──────────────┘
0          8    12    16    20            31
```

STMY $R_1,D_2(X_2,B_2)$ [RSY]

```
┌──────────┬─────┬─────┬─────┬─────/─────────────┬──────────┐
│   'EB'   │ R1  │ R3  │ B2  │ DL2 │     DH2     │   '90'   │
└──────────┴─────┴─────┴─────┴─────/─────────────┴──────────┘
0          8    12    16    20    32            40         47
```

STMG $R_1,R_3,D_2(B_2)$ [RSY]

```
┌──────────┬─────┬─────┬─────┬─────/─────────────┬──────────┐
│   'EB'   │ R1  │ R3  │ B2  │ DL2 │     DH2     │   '24'   │
└──────────┴─────┴─────┴─────┴─────/─────────────┴──────────┘
0          8    12    16    20    32            40         47
```

The contents of bit positions of the set of general registers starting with general register R_1 and ending with general register R_3 are placed in the storage area beginning at the location designated by the second-operand address and continuing through as many locations as needed.

For STORE MULTIPLE (STM, STMY), the contents of bit positions 32-63 of the general registers are stored in successive four-byte fields beginning at the second-operand address. For STORE MULTIPLE (STMG), the contents of bit positions 0-63 of the general registers are stored in successive eight-byte fields beginning at the second-operand address.

The general registers are stored 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 STM is treated as a 12-bit unsigned binary integer. The displacement for STMY and STMG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

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

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

STORE MULTIPLE HIGH

STMH $R_1,R_3,D_2(B_2)$ [RSY]

```
┌──────────┬─────┬─────┬─────┬─────/─────────────┬──────────┐
│   'EB'   │ R1  │ R3  │ B2  │ DL2 │     DH2     │   '26'   │
└──────────┴─────┴─────┴─────┴─────/─────────────┴──────────┘
0          8    12    16    20    32            40         47
```

The contents of 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 placed in the storage area beginning at the location designated by the second-operand address and continuing through as many locations as needed, that is, the contents of bit positions 0-31 are stored in successive four-byte fields beginning at the second-operand address. Bits 32-63 of the registers are ignored.

The general registers are stored 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 (store, 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.

STORE PAIR TO QUADWORD

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

 ┌────────┬────┬────┬────┬/───┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '8E'  │
 └────────┴────┴────┴────┴/───┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

The quadword first operand is stored at the second-operand location. The store at the second-operand location appears to be quadword concurrent as observed by other CPUs. The left doubleword of the first operand is in general register R_1, and the right doubleword is in general register $R_1 + 1$.

The R_1 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 (store, operand 2)
- Specification

Programming Notes:

1. The STORE MULTIPLE (STM or STMG) instruction does not necessarily provide quadword-concurrent access.

2. The performance of STORE PAIR TO QUADWORD on some models may be significantly slower than that of STORE MULTIPLE (STMG). Unless quadword consistency is required, STMG should be used instead of STPQ.

STORE REVERSED

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

 ┌────────┬────┬────┬────┬/───┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '3F'  │
 └────────┴────┴────┴────┴/───┴────────┴────────┘
 0        8    12   16   20   32       40      47

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

 ┌────────┬────┬────┬────┬/───┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '3E'  │
 └────────┴────┴────┴────┴/───┴────────┴────────┘
 0        8    12   16   20   32       40      47

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

 ┌────────┬────┬────┬────┬/───┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '2F'  │
 └────────┴────┴────┴────┴/───┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

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

For STORE REVERSED (STRVH), the first operand is two bytes in bit positions 48-63 of general register R_1. For STORE REVERSED (STRV), the first operand is four bytes in bit positions 32-63 of general register R_1. For STORE REVERSED (STRVG), the first operand is eight bytes in bit positions 0-63 of general register R_1.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (store, operand 2)

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. The storage-operand references of STORE REVERSED may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

SUBTRACT

```
SR     R₁,R₂     [RR]

┌────────┬────┬────┐
│  '1B'  │ R₁ │ R₂ │
└────────┴────┴────┘
0        8    12   15
```

```
SGR     R₁,R₂     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B909'     │////////│ R₁ │ R₂ │
└────────────────┴────────┴────┴────┘
0                16       24   28   31
```

```
SGFR     R₁,R₂     [RRE]

┌────────────────┬────────┬────┬────┐
│     'B919'     │////////│ R₁ │ R₂ │
└────────────────┴────────┴────┴────┘
0                16       24   28   31
```

```
S       R₁,D₂(X₂,B₂)     [RX]

┌────────┬────┬────┬────┬────────────┐
│  '5B'  │ R₁ │ X₂ │ B₂ │     D₂     │
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20           31
```

```
SY      R₁,D₂(X₂,B₂)     [RXY]

┌────────┬────┬────┬────┬────/────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '5B'  │
└────────┴────┴────┴────┴────/────────┴────────┘
0        8    12   16   20   32       40       47
```

```
SG      R₁,D₂(X₂,B₂)     [RXY]

┌────────┬────┬────┬────┬────/────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '09'  │
└────────┴────┴────┴────┴────/────────┴────────┘
0        8    12   16   20   32       40       47
```

```
SGF     R₁,D₂(X₂,B₂)     [RXY]

┌────────┬────┬────┬────┬────/────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '19'  │
└────────┴────┴────┴────┴────/────────┴────────┘
0        8    12   16   20   32       40       47
```

The second operand is subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT (SR, S, SY), the operands and the difference are treated as 32-bit signed binary integers. For SUBTRACT (SGR, SG), they are treated as 64-bit signed binary integers. For SUBTRACT (SGFR, SGF), the second operand is treated as a 32-bit signed binary integer, and the first operand and the difference 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 S is treated as a 12-bit unsigned binary integer. The displacement for SY, SG, and SGF 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 S, SY, SG, and SGF only)
- Fixed-point overflow
- Operation (SY, if the long-displacement facility is not installed)

Programming Notes:

1. For SR and SGR, when R₁ and R₂ designate the same register, subtracting is equivalent to clearing the register.

2. Subtracting a maximum negative number from itself gives a zero result and no overflow.

SUBTRACT HALFWORD

```
SH      R₁,D₂(X₂,B₂)     [RX]

┌────────┬────┬────┬────┬────────────┐
│  '4B'  │ R₁ │ X₂ │ B₂ │     D₂     │
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20           31
```

```
SHY     R₁,D₂(X₂,B₂)     [RXY]

┌────────┬────┬────┬────┬────/────────┬────────┐
│  'E3'  │ R₁ │ X₂ │ B₂ │DL₂ │  DH₂   │  '7B'  │
└────────┴────┴────┴────┴────/────────┴────────┘
0        8    12   16   20   32       40       47
```

The second operand is subtracted from the first operand, and the difference 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. The first operand and the difference are treated as 32-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 SH is treated as a 12-bit unsigned binary integer. The displacement for SHY 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)
- Fixed-point overflow
- Operation (SHY, if the long-displacement facility is not installed)

Programming Note: The function of a SUBTRACT HALFWORD IMMEDIATE instruction, which is an instruction not provided, can be obtained by using an ADD HALFWORD IMMEDIATE instruction with a negative I_2 field.

SUBTRACT LOGICAL

```
SLR    R1,R2      [RR]

 ┌────────┬────┬────┐
 │  '1F'  │ R1 │ R2 │
 └────────┴────┴────┘
 0        8   12   15
```

```
SLGR     R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B90B'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

```
SLGFR    R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B91B'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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

 ┌────────┬────┬────┬────┬────────────┐
 │  '5F'  │ R1 │ X2 │ B2 │     D2     │
 └────────┴────┴────┴────┴────────────┘
 0        8   12   16   20           31
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '5F'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8   12   16   20   32       40       47
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '0B'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8   12   16   20   32       40       47
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '1B'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8   12   16   20   32       40       47
```

The second operand is subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT LOGICAL (SLR, SL, SLY), the operands and the difference are treated as 32-bit unsigned binary integers. For SUBTRACT LOGICAL (SLGR, SLG), they are treated as 64-bit unsigned binary integers. For SUBTRACT LOGICAL (SLGFR, SLGF) the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the difference are treated as 64-bit unsigned binary integers.

The displacement for SL is treated as a 12-bit unsigned binary integer. The displacement for SLY, SLG, and SLGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 --
1 Result not zero; borrow
2 Result zero; no borrow
3 Result not zero; no borrow

Program Exceptions:

- Access (fetch, operand 2 of SL, SLY, SLG, and SLGF only)
- Operation (SLY, if the long-displacement facility is not installed)

Programming Notes:

1. Logical subtraction is performed by adding the one's complement of the second operand and a value of one to the first operand. The use of the one's complement and the value of one instead of the two's complement of the second operand results in a carry when the second operand is zero.

2. SUBTRACT LOGICAL differs from SUBTRACT only in the meaning of the condition code and in the absence of the interruption for overflow.

3. A zero difference is always accompanied by a carry out of bit position 0 for SLGR, SLGFR, SLG, and SLGF or bit position 32 for SLR, SL, and SLY, and, therefore, no borrow.

4. The condition-code setting for SUBTRACT LOGICAL can also be interpreted as indicating the presence or absence of a carry, as follows:

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

SUBTRACT LOGICAL WITH BORROW

```
SLBR    R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B999'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

```
SLBGR   R1,R2      [RRE]

 ┌────────────────┬────────┬────┬────┐
 │     'B989'     │////////│ R1 │ R2 │
 └────────────────┴────────┴────┴────┘
 0                16       24   28  31
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '99'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

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

 ┌────────┬────┬────┬────┬────┬────────┬────────┐
 │  'E3'  │ R1 │ X2 │ B2 │DL2 │  DH2   │  '89'  │
 └────────┴────┴────┴────┴────┴────────┴────────┘
 0        8    12   16   20   32       40      47
```

The second operand and the borrow are subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT LOGICAL WITH BORROW (SLBR, SLB), the operands, the borrow, and the difference are treated as 32-bit unsigned binary integers. For SUBTRACT LOGICAL WITH BORROW (SLBGR, SLBG), they are treated as 64-bit unsigned binary integers.

Resulting Condition Code:

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

Program Exceptions:

- Access (fetch, operand 2 of SLB and SLBG only)

Programming Notes:

1. A borrow is represented by a zero 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 zero by an execution of a SUBTRACT LOGICAL or SUBTRACT LOGICAL WITH BORROW instruction that produces a borrow into the leftmost bit position of the 32-bit or 64-bit result.

2. Logical subtraction with borrow is performed by adding the one's complement of the second operand and bit 18 of the current PSW to the first operand. Therefore, when bit 18 is one, indicating no borrow, the addition is the same as for SUBTRACT LOGICAL.

3. Condition code zero is set for SUBTRACT LOGICAL WITH BORROW (SLBR, SLB), when the maximum 32-bit unsigned binary integer, $2^{32}-1$, is subtracted from zero when PSW bit 18 indicates a borrow. For SUBTRACT LOGICAL WITH BORROW (SLBGR, SLBG) condition code zero is set when the maximum 64-bit unsigned binary integer, $2^{64}-1$, is subtracted from zero when PSW bit 18 indicates a borrow.

4. SUBTRACT and SUBTRACT LOGICAL may provide better performance than SUBTRACT LOGICAL WITH BORROW, depending on the model.

SUPERVISOR CALL

```
SVC    I          [RR]

 ┌────────┬────────┐
 │  '0A'  │   I    │
 └────────┴────────┘
0         8       15
```

The instruction causes a supervisor-call interruption, with the I field of the instruction providing the rightmost byte of the interruption code.

Bits 8-15 of the instruction, with eight zeros appended on the left, are placed in the supervisor-call interruption code that is stored in the course of the interruption. See “Supervisor-Call Interruption” on page 6-47.

A serialization and checkpoint-synchronization function is performed.

Condition Code: The code remains unchanged and is saved as part of the old PSW. A new condition code is loaded as part of the supervisor-call interruption.

Program Exceptions: None.

TEST ADDRESSING MODE

```
TAM     [E]

 ┌────────────────┐
 │     '010B'     │
 └────────────────┘
0                15
```

The extended-addressing-mode bit and basic-addressing-mode bit, bits 31 and 32 of the current PSW, respectively, are tested, and the result is indicated in the condition code.

Resulting Condition Code:

0 PSW bits 31 and 32 zeros (indicating 24-bit addressing mode)
1 PSW bit 31 zero and bit 32 one (indicating 31-bit addressing mode)
2 --
3 PSW bits 31 and 32 ones (indicating 64-bit addressing mode)

Program Exceptions: None.

Programming Note: The case when PSW bit 31 is one and bit 32 is zero causes an early PSW specification exception to be recognized.

TEST AND SET

```
TS     D2(B2)          [S]

 ┌────────┬────────┬────┬────────────┐
 │  '93'  │////////│ B2 │     D2     │
 └────────┴────────┴────┴────────────┘
0         8       16   20           31
```

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the byte is set to all ones.

Bits 8-15 of the instruction are ignored.

The byte in storage is set to all ones as it is fetched for the testing of bit 0. This update appears to be an interlocked-update reference as observed by other CPUs.

A serialization function is performed before the byte is fetched and again after the storing of all ones.

Resulting Condition Code:

0 Leftmost bit zero
1 Leftmost bit one
2 --
3 --

Program Exceptions:

- Access (fetch and store, operand 2)

Programming Notes:

1. TEST AND SET may be used for controlled sharing of a common storage area by programs operating on different CPUs. This instruction is provided primarily for compatibility with programs written for System/360. The instructions COMPARE AND SWAP and COMPARE DOUBLE AND SWAP provide functions which are more suitable for sharing among programs on a single CPU or for programs that may be interrupted. See the description of these instructions and the associated programming notes for details.

2. TEST AND SET does not interlock against storage accesses by channel programs. Therefore, the instruction should not be used to update a location into which a channel program may store, since the channel-program data may be lost.

TEST UNDER MASK (TEST UNDER MASK HIGH, TEST UNDER MASK LOW)

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

 ┌─────────┬─────────┬────┬────────────┐
 │  '91'   │   I2    │ B1 │     D1     │
 └─────────┴─────────┴────┴────────────┘
 0         8        16   20           31
```

```
TMY    D1(B1),I2        [SIY]
                              /
 ┌─────────┬─────────┬────┬────┬─────────┬─────────┐
 │  'EB'   │   I2    │ B1 │DL1 │   DH1   │  '51'   │
 └─────────┴─────────┴────┴────┴─────────┴─────────┘
                              /
 0         8        16   20   32        40        47
```

```
TMHH     R1,I2      [RI]

 ┌─────────┬────┬────┬─────────────────┐
 │  'A7'   │ R1 │'2' │       I2        │
 └─────────┴────┴────┴─────────────────┘
 0         8   12   16                31
```

```
TMHL     R1,I2      [RI]

 ┌─────────┬────┬────┬─────────────────┐
 │  'A7'   │ R1 │'3' │       I2        │
 └─────────┴────┴────┴─────────────────┘
 0         8   12   16                31
```

```
TMLH or TMH     R1,I2     [RI]

 ┌─────────┬────┬────┬─────────────────┐
 │  'A7'   │ R1 │'0' │       I2        │
 └─────────┴────┴────┴─────────────────┘
 0         8   12   16                31
```

```
TMLL or TML     R1,I2     [RI]

 ┌─────────┬────┬────┬─────────────────┐
 │  'A7'   │ R1 │'1' │       I2        │
 └─────────┴────┴────┴─────────────────┘
 0         8   12   16                31
```

A mask is used to select bits of the first operand, and the result is indicated in the condition code.

TEST UNDER MASK is a new name of, and TMLH and TMLL are new mnemonics for, the ESA/390 instructions TEST UNDER MASK HIGH (TMH) and TEST UNDER MASK LOW (TML), respectively.

In TEST UNDER MASK (TM, TMY), the byte of immediate data, I_2, is used as an eight-bit mask. The bits of the mask are made to correspond one for one with the bits of the byte in storage designated by the first-operand address.

A mask bit of one indicates that the storage bit is to be tested. When the mask bit is zero, the storage bit is ignored. When all storage bits thus selected are zero, condition code 0 is set. Condition code 0 is also set when the mask is all zeros. When the selected bits are all ones, condition code 3 is set; otherwise, condition code 1 is set.

Access exceptions associated with the storage operand are recognized for one byte even when the mask is all zeros.

In TEST UNDER MASK (TMHH, TMHL, TMLH, TMLL), The contents of the I_2 field are used as a 16-bit mask. For each instruction, the bits of the mask are made to correspond one for one with 16 bits of the first operand as follows:

Instruction	Bits Tested
TMHH	0-15
TMHL	16-31
TMLH (or TMH)	32-47
TMLL (or TML)	48-63

A mask bit of one indicates that the first-operand bit is to be tested. When the mask bit is zero, the first-operand bit is ignored. When all first-operand bits thus selected are zero, condition code 0 is set. Condition code 0 is also set when the mask is all zeros. When the selected bits are mixed zeros and ones, condition code 1 is set if the leftmost selected bit is zero, or condition code 2 is set if the leftmost selected bit is one. When the selected bits are all ones, condition code 3 is set.

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

Resulting Condition Code:

0 Selected bits all zeros; or mask bits all zeros
1 Selected bits mixed zeros and ones (TM and TMY only)
1 Selected bits mixed zeros and ones, and leftmost is zero (TMHH, TMHL, TMLH, TMLL)
2 -- (TM and TMY only)
2 Selected bits mixed zeros and ones, and leftmost is one (TMHH, TMHL, TMLH, TMLL)
3 Selected bits all ones

Program Exceptions:

- Access (fetch, operand 1, TM and TMY only)
- Operation (TMY, if the long-displacement facility is not installed)

Programming Notes:

1. An example of the use of the TEST UNDER MASK (TM) instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When the mask for TMHH, TMHL, TMLH, or TMLL selects exactly two bits, the two selected bits effectively are loaded into the condition code.

TRANSLATE

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

┌─────────┬─────────┬────┬───/──┬────┬──/──┐
│  'DC'   │    L    │ B1 │  D1  │ B2 │ D2  │
└─────────┴─────────┴────┴───/──┴────┴──/──┘
0         8         16   20     32   36   47
```

The bytes of the first operand are used as eight-bit arguments to reference a list designated by the second-operand address. Each function byte selected from the list replaces the corresponding argument in the first operand.

The L field specifies the length of only the first operand.

The bytes of the first operand are selected one by one for translation, proceeding left to right. Each argument byte is added to the initial second-operand address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned binary integer and extended with zeros on the left. The sum is used as the address of the function byte, which then replaces the original argument byte.

The operation proceeds until the first-operand field is exhausted. The list is not altered unless an overlap occurs.

When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the corresponding function byte.

Access exceptions are recognized only for those bytes in the second operand which are actually required.

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 TRANSLATE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. TRANSLATE may be used to convert data from one code to another code.
3. The instruction may also be used to rearrange data. This may be accomplished by placing a pattern in the destination area, by designating the pattern as the first operand of TRANSLATE, and by designating the data that is to be rearranged as the second operand. Each byte of the pattern contains an eight-bit number specifying the byte destined for this position. Thus, when the instruction is executed, the pattern selects the bytes of the second operand in the desired order.

4. Because each eight-bit argument byte is added to the initial second-operand address to obtain the address of a function byte, the list may contain 256 bytes. In cases where it is known that not all eight-bit argument values will occur, it is possible to reduce the size of the list.

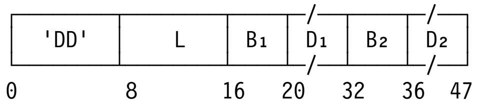
5. Significant performance degradation is possible when, with DAT on, the second-operand address of TRANSLATE designates a location that is less than 256 bytes to the left of a 4K-byte boundary. This is because the machine may perform a trial execution of the instruction to determine if the second operand actually crosses the boundary.

6. The fetch and subsequent store accesses to a particular byte in the first-operand field 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” on page A-1.

7. The storage-operand references of TRANSLATE may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

TRANSLATE AND TEST

TRT $D_1(L,B_1),D_2(B_2)$ [SS]



The bytes of the first operand are used as eight-bit arguments to select function bytes from a list designated by the second-operand address. The first nonzero function byte is inserted in general register 2, and the related argument address in general register 1.

The L field specifies the length of only the first operand.

The bytes of the first operand are selected one by one for translation, proceeding left to right. The first operand remains unchanged in storage. Calculation of the address of the function byte is performed as in the TRANSLATE instruction. The function byte retrieved from the list is inspected for a value of zero.

When the function byte is zero, the operation proceeds with the next byte of the first operand. When the first-operand field is exhausted before a nonzero function byte is encountered, the operation is completed by setting condition code 0. The contents of general registers 1 and 2 remain unchanged.

When the function byte is nonzero, the operation is completed by inserting the function byte in general register 2 and the related argument address in general register 1. This address points to the argument byte last translated. The function byte replaces bits 56-63 of general register 2, and bits 0-55 of this register remain unchanged. In the 24-bit addressing mode, the address replaces bits 40-63 of general register 1, and bits 0-39 of this register remain unchanged. In the 31-bit addressing mode, the address replaces bits 33-63 of general register 1, bit 32 of this register is set to zero, and bits 0-31 of the register remain unchanged. In the 64-bit addressing mode, the address replaces bits 0-63 of general register 1.

When the function byte is nonzero, either condition code 1 or 2 is set, depending on whether the argument byte is the rightmost byte of the first operand. Condition code 1 is set if one or more argument bytes remain to be translated. Condition code 2 is set if no more argument bytes remain.

The contents of access register 1 always remain unchanged.

Access exceptions are recognized only for those bytes in the second operand which are actually required. Access exceptions are not recognized for those bytes in the first operand which are to the right of the first byte for which a nonzero function byte is obtained.

Resulting Condition Code:

0 All function bytes zero
1 Nonzero function byte; first-operand field not exhausted
2 Nonzero function byte; first-operand field exhausted

3 --

Program Exceptions:

- Access (fetch, operands 1 and 2)

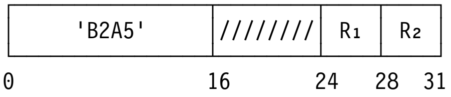
Programming Notes:

1. An example of the use of the TRANSLATE AND TEST instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. TRANSLATE AND TEST may be used to scan the first operand for characters with special meaning. The second operand, or list, is set up with all-zero function bytes for those characters to be skipped over and with nonzero function bytes for the characters to be detected.

TRANSLATE EXTENDED

```
TRE      R1,R2     [RRE]
```



The bytes of the first operand are compared to a test byte in general register 0 and, unless an equal comparison occurs, are used as eight-bit arguments to reference a 256-byte translation table designated by the second-operand address. Each function byte selected from the second operand replaces the corresponding argument in the first operand. The operation proceeds until a first-operand byte equal to the test byte is encountered, the end of the first operand is reached, or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R_1 field designates 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. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand location is specified by the contents of bit positions 32-63 of general register $R_1 + 1$, and those contents are treated as a 32-bit unsigned binary integer. In the 64-bit addressing mode, the number of bytes in the first-operand location is specified by the entire contents of general register $R_1 + 1$, and those contents are treated as a 64-bit unsigned binary integer.

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 test byte is in bit positions 56-63 of general register 0, and the contents of bit positions 0-55 of this register are ignored.

The contents of the registers just described are shown in Figure 7-75 on page 7-182.

The bytes of the first operand are selected one by one for translation, proceeding left to right. Each argument byte is first compared to the test byte in general register 0. If the result is an equal comparison, the operation is completed. If the argument byte is not equal to the test byte, the argument byte is added to the initial second-operand address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned binary integer and extended with zeros on the left. The sum is used as the address of the function byte, which then replaces the original argument byte. The second operand is not altered unless an overlap occurs.

The operation proceeds until a first-operand byte equal to the test byte is encountered, the first-operand location is exhausted, or a CPU-determined number of first-operand bytes have been processed.

When the first-operand location is exhausted without finding a byte equal to the test byte, condition code 0 is set. When a first-operand byte equal to the test byte is encountered, condition code 1 is set. When a CPU-determined number of bytes have been processed, condition code 3 is set. Condition code 3 may be set even when the first-operand location is exhausted or when the next byte to be processed is equal to the test byte. In these cases, condition code 0 or 1, respectively, will be set when the instruction is executed again.

If the operation is completed with condition code 0, the contents of general register R_1 are incremented by the contents of general register $R_1 + 1$, and then the contents of general register $R_1 + 1$ are set to zero. If the operation is completed with condition code 1, the contents of general register $R_1 + 1$ are decremented by the number of bytes processed before the first-operand byte equal to the test byte was encountered, and the contents of general register R_1 are incremented by the same number, so that general register R_1 contains the address of the equal byte. If the operation is completed with condition code 3, the contents of general register $R_1 + 1$ are decremented by the number of bytes processed, and the contents of general register R_1 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next byte to be processed. When general register R_1 is updated in the 24-bit or 31-bit addressing mode,

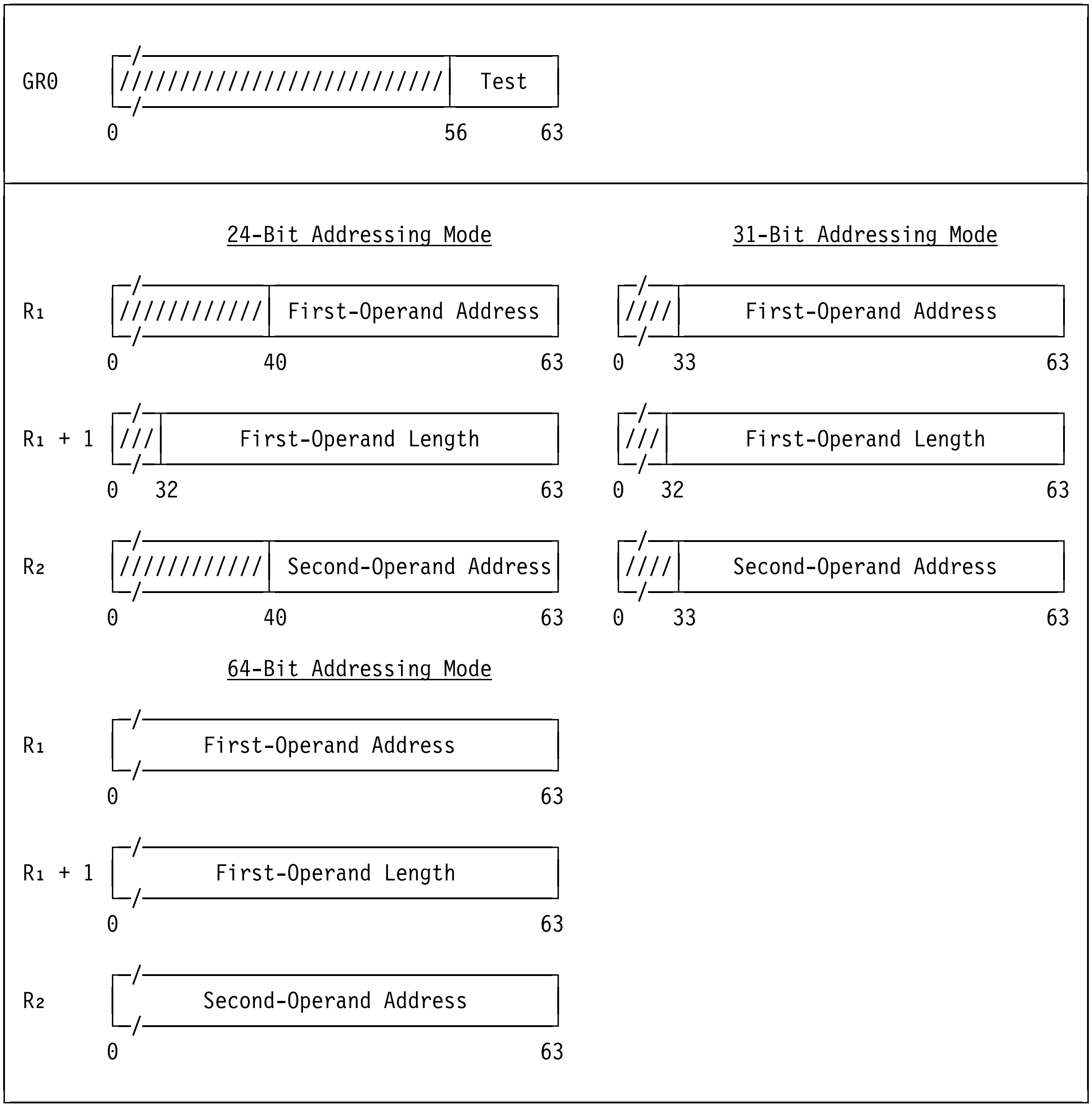


Figure 7-75. Register Contents for TRANSLATE EXTENDED

bits 32-39 of it, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged from their original values.

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

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.

When the R_2 register is the same register as the R_1 or $R_1 + 1$ register, the results are unpredictable.

When R_1 or R_2 is zero, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portion of the first operand to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

Access exceptions for all 256 bytes of the second operand may be recognized, even if not all bytes are used.

Access exceptions are not recognized if the R_1 field is odd. When the length of the first operand is zero, no access exceptions for the first operand are recognized.

Resulting Condition Code:

0 Entire first operand processed without finding a byte equal to the test byte
1 First-operand byte is equal to the test byte
2 --
3 CPU-determined number of bytes processed

Program Exceptions:

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

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the translation. The program need not determine the number of bytes that were translated.
2. The instruction can improve performance by being used in place of a TRANSLATE AND TEST instruction that locates an escape character, followed by a TRANSLATE instruction that translates the bytes preceding the escape character.
3. The storage-operand references of TRANSLATE EXTENDED may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

TRANSLATE ONE TO ONE

TROO R_1,R_2 [RRE]

'B993'	////////	R_1	R_2

0 16 24 28 31

TRANSLATE ONE TO TWO

TROT R_1,R_2 [RRE]

'B992'	////////	R_1	R_2

0 16 24 28 31

TRANSLATE TWO TO ONE

TRTO R_1,R_2 [RRE]

'B991'	////////	R_1	R_2

0 16 24 28 31

TRANSLATE TWO TO TWO

TRTT R_1,R_2 [RRE]

'B990'	////////	R_1	R_2

0 16 24 28 31

The characters of the second operand are used as arguments to select function characters from a translation table designated by the address in general register 1. Each function character selected from the translation table is compared to a test character in general register 0, and, unless an equal comparison occurs, is placed at the first-operand location. The operation proceeds until a selected function character equal to the test character is encountered, the end of the second operand is reached, or a CPU-determined number of characters have been processed, whichever occurs first. The result is indicated in the condition code.

The lengths of the operand and test characters are as follows:

- For TRANSLATE ONE TO ONE, the second-operand, first-operand, and test characters are single bytes.
- For TRANSLATE ONE TO TWO, the second-operand characters are single bytes, and the first-operand and test characters are double bytes.
- For TRANSLATE TWO TO ONE, the second-operand characters are double bytes, and the first-operand and test characters are single bytes.
- For TRANSLATE TWO TO TWO, the second-operand, first-operand, and test characters are double bytes.

For TRANSLATE ONE TO ONE and TRANSLATE TWO TO ONE, the test character is in bit positions 56-63 of general register 0. For TRANSLATE ONE TO TWO and TRANSLATE TWO TO TWO, the test character is in bit positions 48-63 of general register 0.

The R_1 field designates 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. In the 24-bit or 31-bit addressing mode, the number of bytes in the second-operand location is specified by the contents of bit positions 32-63 of general register $R_1 + 1$, and those contents are treated as a 32-bit unsigned binary integer. In the 64-bit addressing mode, the number of bytes in the second-operand location is specified by the contents of bit positions 0-63 of general register $R_1 + 1$, and those contents are treated as a 64-bit unsigned binary integer. The length of the first-operand location is considered to be the same as that of the second operand for TRANSLATE ONE TO ONE and TRANSLATE TWO TO TWO, twice that for TRANSLATE ONE TO TWO, and one half that for TRANSLATE TWO TO ONE.

For TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO, the length in general register $R_1 + 1$ must be an even number of bytes; otherwise, a specification exception is recognized.

The translation table is treated as being on a doubleword boundary for TRANSLATE ONE TO ONE and TRANSLATE ONE TO TWO and on a 4K-byte boundary for TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO. The rightmost bits of the register that are not used to form the address, which are bits 61-63 in the doubleword case and bits 52-63 in the 4K-byte case, are ignored.

The handling of the addresses in general registers R_1, R_2, and 1 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 and 40-60 or 40-51 of 1 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 registers R_1 and R_2 and 33-60 or 33-51 of 1 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 of registers R_1 and R_2 and 0-60 or 0-51 of 1 constitute the address.

The contents of the registers just described are shown in Figure 7-76 on page 7-185.

In the access-register mode, the contents of access registers R_1, R_2, and 1 are used for accessing the first operand, second operand, and translation table, respectively.

The length of the translation table designated by the address contained in general register 1 is as follows:

- For TRANSLATE ONE TO ONE, the translation-table length is 256 bytes; each of the 256 function characters is a single byte.
- For TRANSLATE ONE TO TWO, the translation-table length is 512 bytes; each of the 256 function characters is a double byte.
- For TRANSLATE TWO TO ONE, the translation-table length is 65,536 (64K) bytes; each of the 64K function characters is a single byte.
- For TRANSLATE TWO TO TWO, the translation-table length is 131,072 (128K) bytes; each of the 64K function characters is a double byte.

The characters of the second operand are selected one by one for translation, proceeding left to right. Each argument character is added to the initial translation-table address. The addition is performed following the rules for address arithmetic, with the argument character treated as follows:

- For TRANSLATE ONE TO ONE, the argument character is treated as an eight-bit

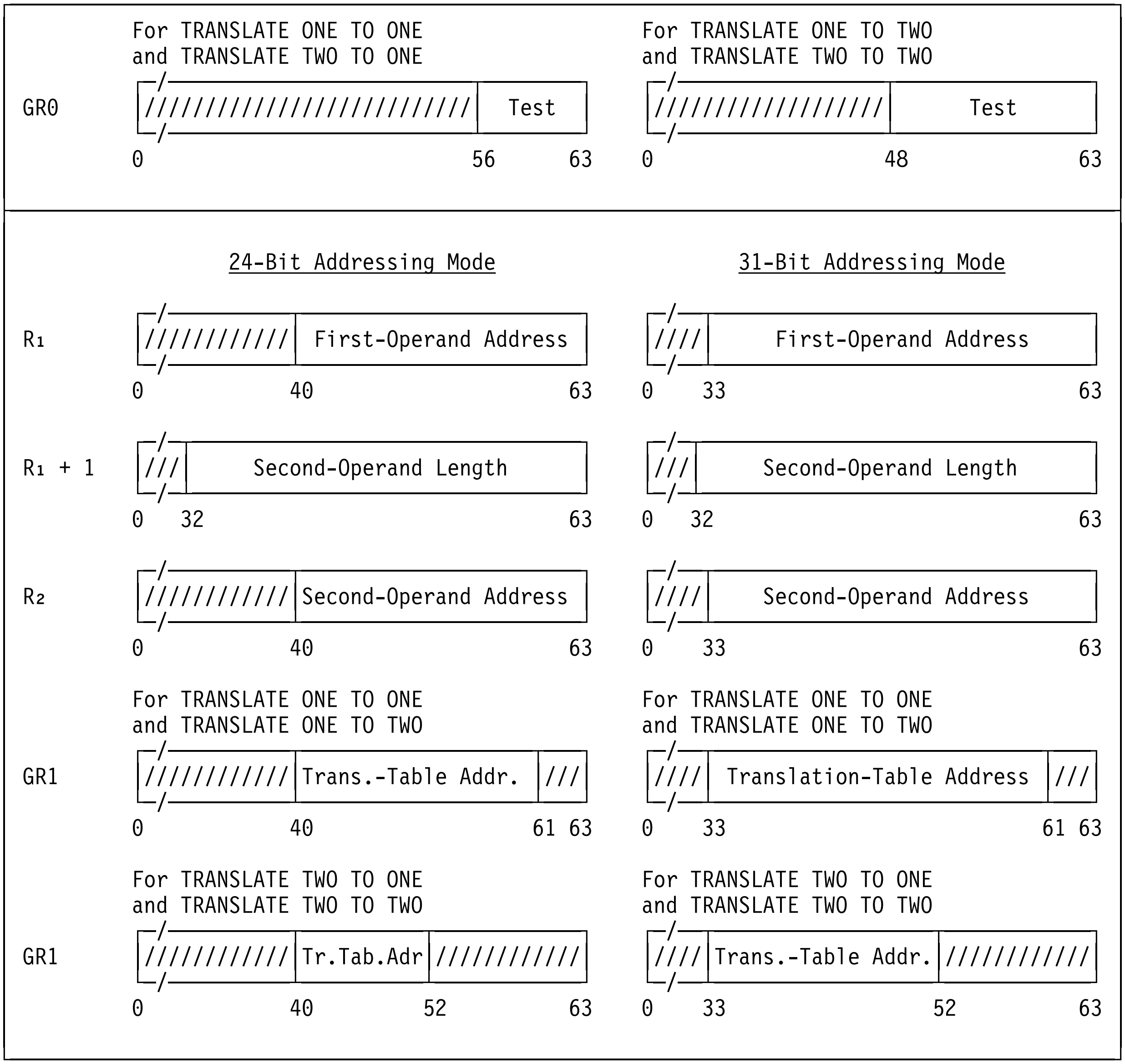


Figure 7-76 (Part 1 of 2). Register Contents for TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO

unsigned binary integer extended on the left with 56 zeros.

- For TRANSLATE ONE TO TWO, the argument character is treated as an eight-bit unsigned binary integer extended on the right with a zero and on the left with 55 zeros.
- For TRANSLATE TWO TO ONE, the argument character is treated as a 16-bit unsigned binary integer extended on the left with 48 zeros.
- For TRANSLATE TWO TO TWO, the argument character is treated as a 16-bit unsigned binary integer extended on the right with a zero and on the left with 47 zeros.

The rightmost bits of the translation-table address that are ignored (61-63 or 52-63) are treated as zeros during this addition.

The sum is used as the address of the function character.

Each function character selected as described above is first compared to the test character in general register 0. If the result is an equal comparison, the operation is completed. If the function character is not equal to the test character, the function character is placed in the next available character position in the first operand, that is, the first function character is placed at the beginning of the first-operand location, and each successive function character is placed immediately to the right of the preceding character. The second operand and the translation table are not altered unless an overlap occurs.

The operation proceeds until a selected function character equal to the test character is encountered, the second-operand location is exhausted, or a CPU-determined number of second-operand characters have been processed.

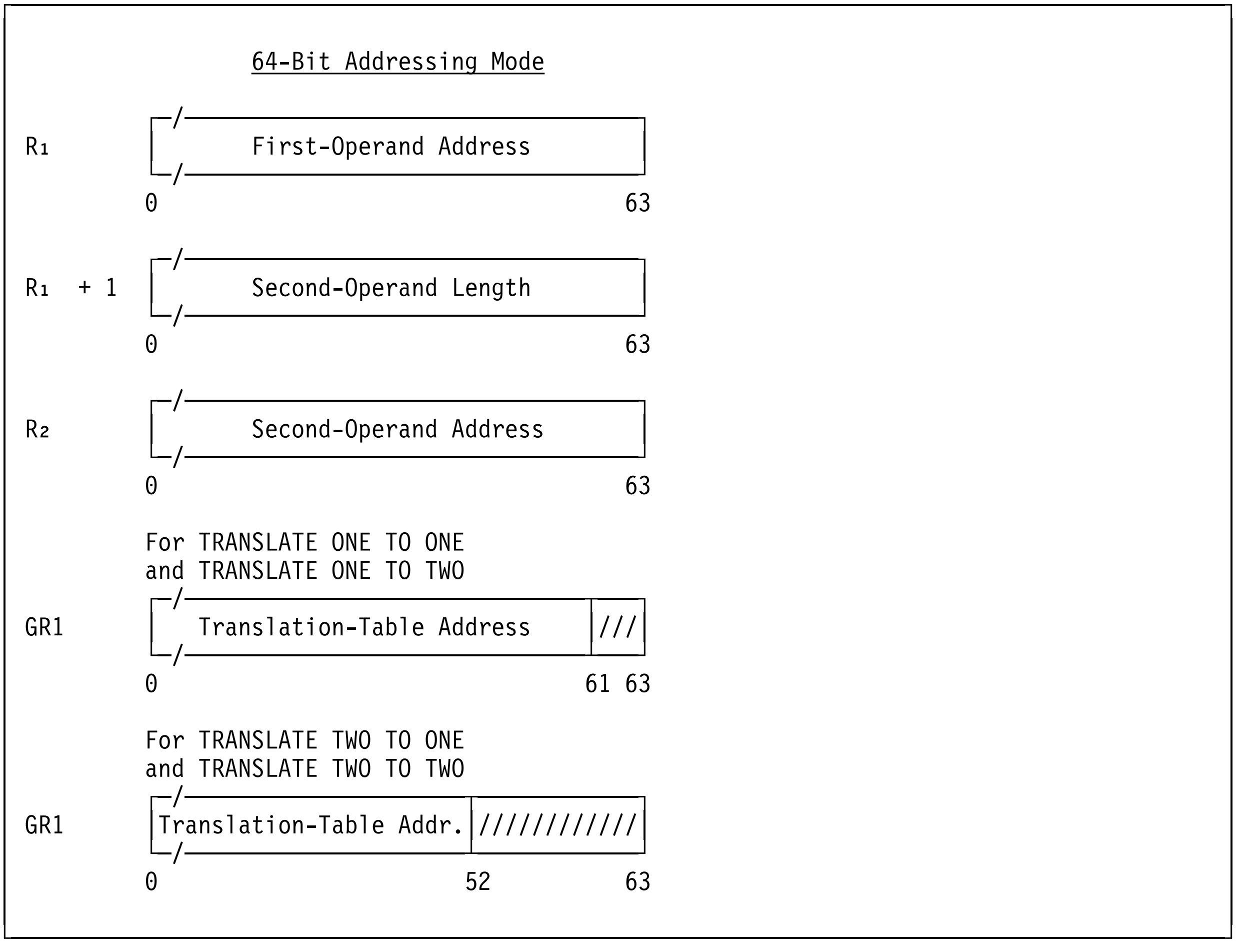


Figure 7-76 (Part 2 of 2). Register Contents for TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO

When a selected function character equal to the test character is encountered, condition code 1 is set. When the second-operand location is exhausted without finding a selected function character equal to the test character, condition code 0 is set. When a CPU-determined number of characters have been processed, condition code 3 is set. Condition code 3 may be set even when the next character to be processed results in a function character equal to the test character or when the second-operand location is exhausted. In these cases, condition code 1 or 0, respectively, will be set when the instruction is executed again.

If the operation is completed with condition code 0, the contents of general register R_2 are incremented by the contents of general register $R_1 + 1$, and the contents of general register R_1 are incremented as follows:

- For TRANSLATE ONE TO ONE and TRANSLATE TWO TO TWO, the same as for general register R_2.
- For TRANSLATE ONE TO TWO, by twice the amount for general register R_2.
- For TRANSLATE TWO TO ONE, by one half the amount for general register R_2.

The contents of general register $R_1 + 1$ are then set to zero.

If the operation is completed with condition code 1, the contents of general register $R_1 + 1$ are decremented by the number of second-operand bytes processed before the character that selected a function character equal to the test character was encountered, and the contents of general register R_2 are incremented by the same number, so that general register R_2 contains the address of the character that selected a function character equal to the test character. The contents of general register R_1 are incremented by the same, twice, or one half the number, as described above for condition code 0.

If the operation is completed with condition code 3, the contents of general register $R_1 + 1$ are decremented by the number of second-operand bytes processed, and the contents of general register R_2 are incremented by the same number, so that the instruction, when reexecuted, contains the address of the next character to be processed. The contents of general register R_1 are incremented by the same, twice, or one half the number, as described above for condition code 0.

When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, the bits in bit positions 32-39 of them that are not part of the address may be set to zeros or may remain unchanged from their original values. In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, and R_2 always remain unchanged.

The contents of general registers 0 and 1 remain unchanged.

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.

During instruction execution, CPU retry may result in condition code 3 being set with possibly incorrect data having been stored in the first operand location at or to the right of the location designated by the final address in general register R_1. The amount of data stored depends on the operation and the point in time at which CPU retry occurred. In all cases, the storing will occur again, with correct data stored, when the instruction is executed again to continue processing the same operands.

When the R_1 register is the same register as the R_2 register, the R_1 or R_2 register is register 0, or the R_2 register is register 1, the results are unpredictable.

When any of the first and second operands and the translation table overlaps another of them, the results are unpredictable.

Access exceptions for the portion of the first or second operand to the right of the last character processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last character processed.

Access exceptions for all characters of the translation table may be recognized even if not all characters are used.

Access exceptions are not recognized if the R_1 field is odd. When the length of the second operand is zero, no access exceptions for the first or second operand are recognized, and access exceptions for the translation table may or may not be recognized.

Resulting Condition Code:

0 Entire second operand processed without finding a resulting function character equal to the test character
1 Second-operand character found resulting in a function character equal to the test character
2 --
3 CPU-determined number of characters processed

Program Exceptions:

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

Programming Notes:

1. These instructions differ from the TRANSLATE EXTENDED instruction by having the following attributes:
   - Depending on the instruction used, the sets of argument characters and function characters each can contain single-byte or double-byte characters.
   - The test character is compared to a resulting function character instead of to an argument character.
   - The argument (source) and function (destination) operands are different operands.
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the translation. The program need not determine the number of characters that were translated.
3. The storage operand references of these instructions may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

UNPACK

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

'F3'	L_1	L_2	B_1	D_1	B_2	D_2	
0	8	12	16	20	32	36	47

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

The second operand is treated as having the packed format. Its digits and sign are placed unchanged in the first-operand location, using the zoned format. Zone bits with coding of 1111 are supplied for all bytes except the rightmost byte, the zone of which receives the sign of the second operand. The sign and digits are not checked for valid codes.

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 field 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 the operands were processed one byte at a time and as if the first result byte were stored immediately after fetching the first operand byte. The entire rightmost second-operand byte is used in forming the first result byte. For the remainder of the field, information for two result bytes is obtained from a single second-operand byte, and execution proceeds as if the leftmost four bits of the byte were to remain available for the next result byte and need not be refetched. Thus, the result is as if two result bytes were to be stored immediately after fetching a single operand byte.

Condition Code: The code remains unchanged.

Program Exceptions:

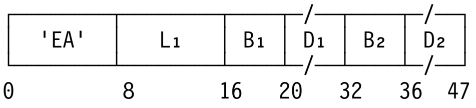
- Access (fetch, operand 2; store, operand 1)

Programming Notes:

1. An example of the use of the UNPACK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. A field that is to be unpacked can be destroyed by improper overlapping. To save storage space for unpacking by overlapping the operands, the rightmost byte of the first operand must be to the right of the rightmost byte of the second operand by the number of bytes in the second operand minus 2. If only one or two bytes are to be unpacked, the rightmost bytes of the two operands may coincide.
3. The storage-operand references of UNPACK may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

UNPACK ASCII

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



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

The second operand is treated as having the packed format. Its digits are converted to ASCII characters by extending them on the left with 0011 binary, and the ASCII characters are then placed at the first operand location. The digits are not checked for valid codes.

The sign of the second operand is not transferred to the first operand but is checked for validity and determines the condition code. If the sign is 1010, 1100, 1110 or 1111 binary (plus), condition code 0 is set. If the sign is 1011 or 1101 binary (minus), condition code 1 is set. If the sign is not one of the codes for plus or minus, condition code 3 is set.

The converted last digit is placed in the rightmost byte position of the result field, and the other converted digits are placed adjacent to the last and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left.

The length of the second operand is 16 bytes. The second operand consists of 31 digits and a sign.

The length of the first operand is designated by the contents of the L_1 field. The first-operand length must not exceed 32 bytes (L_1 must be less than or equal to 31); otherwise, a specification exception is recognized.

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 length of the first operand is 32 bytes, the leftmost byte is set to ASCII zero, 30 hex.

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 is not necessarily stored into in any particular order.

Resulting Condition Code:

0	Sign is plus
1	Sign is minus
2	--
3	Sign is invalid

Program Exceptions:

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

Programming Note:

1. The following example illustrates the use of the instruction to unpack to ASCII digits:

```
ASDIGITS  DS     CL31
PKDIGITS  DS     0PL16
          DC     X'1234567890'
          DC     X'1234567890'
          DC     X'1234567890'
          DC     X'1C'
          ...
          UNPKA  ASDIGITS(31),PKDIGITS
```

2. The storage-operand references of UNPACK ASCII may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

UNPACK UNICODE

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

┌────────┬────────┬────┬────┬────┬────┐
│  'E2'  │   L1   │ B1 │ D1 │ B2 │ D2 │
└────────┴────────┴────┴────┴────┴────┘
0        8        16   20   32   36  47
```

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

The second operand is treated as having the packed format. Its digits are converted to two-byte Unicode characters by extending them on the left with 000000000011 binary (003 hex), and the Unicode characters are then placed at the first operand location. The digits are not checked for valid codes. The sign of the second operand is not transferred to the first operand but is checked for validity and determines the condition code. If the sign is 1010, 1100, 1110 or 1111 binary (plus), condition code 0 is set. If the sign is 1011 or 1101 binary (minus), condition code 1 is set. If the sign is not one of the codes for plus or minus, condition code 3 is set.

The converted last digit is placed in the rightmost character position of the result field, and the other converted digits are placed adjacent to the last and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left.

The length of the second operand is 16 bytes; the second operand consists of 31 digits and a sign.

The length of the first operand is designated by the contents of the L_1 field. The first-operand length must not exceed 32 characters or 64 bytes (L_1 must be less than or equal to 63 and must be odd); otherwise a specification exception is recognized.

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 length of the first operand is 32 characters, the leftmost character is set to Unicode Basic Latin zero, 0030 hex.

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 is not necessarily stored into in any particular order.

Resulting Condition Code:

0 Sign is plus
1 Sign is minus
2 --
3 Sign is invalid

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 the instruction to unpack to European numbers:

```
UNDIGITS  DS     CL62
PKDIGITS  DS     0PL16
          DC     X'1234567890'
          DC     X'1234567890'
          DC     X'1234567890'
          DC     X'1C'
          ...
          UNPKU  UNDIGITS(62),PKDIGITS
```

2. The storage-operand references of UNPACK UNICODE may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

UPDATE TREE

```
UPT      [E]

 ┌────────────────┐
 │     '0102'     │
 └────────────────┘
 0               15
```

The nodes of a tree in storage are examined successively on a path toward the base of the tree, and contents of general register 0, conceptually followed on the right by contents of general register 1, are conditionally interchanged with the contents of the nodes so as to give a unique maximum logical value in general register 0. The first half of a node and general register 0 contain a codeword, which is for use in sort/merge algorithms.

If the codeword in general register 0 equals the codeword in a node, the contents of the node are placed in general registers 2 and 3.

General register 4 contains the base address of the tree, and general register 5 contains the index of a node whose parent node will be examined first.

In the access-register mode, access register 4 specifies the address space containing the tree.

This instruction may be interrupted between units of operation. The condition code is unpredictable if the instruction is interrupted.

The size of a node, the size of a codeword, and the participation of bits 0-31 of general registers 1-5 in the operation depend on the addressing mode. In the 24-bit or 31-bit addressing mode, a node is eight bytes, a codeword is four bytes, and bits 0-31 are ignored and remain unchanged. In the 64-bit addressing mode, a node is 16 bytes, a 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

In the 24-bit or 31-bit addressing mode, the doubleword nodes of a tree in storage are examined successively on a path toward the base of the tree, and the contents of bit positions 32-63 of general register 0, conceptually followed on the right by the contents of bit positions 32-63 of general register 1, are conditionally interchanged with the contents of the nodes so as to give a unique maximum logical value in bit positions 32-63 of general register 0.

Bit positions 32-63 of general register 4 contain the base address of the tree, and bit positions 32-63 of general register 5 contain the index of a node whose parent node will be examined first. The base address is eight less than the address of the root node of the tree. The initial contents of bit positions 32-63 of general registers 4 and 5 must be a multiple of 8; otherwise, a specification exception is recognized.

A unit of operation begins by shifting the contents of bit positions 32-63 of general register 5 right logically one position and then setting bit 61 to zero. However, bits 32-63 of general register 5 remain unchanged if the execution of a unit of operation is nullified or suppressed. If after shifting and setting bit 61 to zero, bits 32-63 of general register 5 are all zeros, the instruction is completed, and condition code 1 is set; otherwise, the unit of operation continues.

Bit 32 of general register 0 is tested. If bit 32 of general register 0 is one, the instruction is completed, and condition code 3 is set.

If bit 32 of general register 0 is zero, the sum of bits 32-63 of general registers 4 and 5 is used as the intermediate value for normal operand address generation. The generated address is the address of a node in storage.

Bits 32-63 of general register 0 are logically compared with the contents of the first word of the currently addressed node. If the register operand is low, the contents of bit positions 32-63 of general registers 0 and 1 are interchanged with those of the node, and a unit of operation is completed. If the register operand is high, no additional action is taken, and the unit of operation is completed. If the compare values are equal, bit positions 32-63 of general register 2, conceptually followed on the right by bit positions 32-63 of general register 3, are loaded from the currently addressed node, the instruction is completed, and condition code 0 is set.

In those cases when the value in the first word of the node is less than or equal to the value in bit positions 32-63 of the register, the contents of the node remain unchanged. However, in some models, these contents may be fetched and subsequently stored back.

Operation in the 64-Bit Addressing Mode

In the 64-bit addressing mode, the quadword nodes of a tree in storage are examined successively on a path toward the base of the tree, and the contents of general register 0, conceptually followed on the right by the contents of general register 1, are conditionally interchanged with the contents of the nodes so as to give a unique maximum logical value in general register 0.

General register 4 contains the base address of the tree, and general register 5 contain the index of a node whose parent node will be examined first. The base address is 16 less than the address of the root node of the tree. The initial contents of general registers 4 and 5 must be a multiple of 16; otherwise, a specification exception is recognized.

A unit of operation begins by shifting the contents of general register 5 right logically one position and then setting bit 60 to zero. However, general register 5 remains unchanged if the execution of a unit of operation is nullified or suppressed. If after shifting and setting bit 60 to zero, the contents of general register 5 are zero, the instruction is completed, and condition code 1 is set; otherwise, the unit of operation continues.

Bit 0 of general register 0 is tested. If bit 0 of general register 0 is one, the instruction is completed, and condition code 3 is set.

If bit 0 of general register 0 is zero, the sum of the contents of general registers 4 and 5 is used as the intermediate value for normal operand address generation. The generated address is the address of a node in storage.

The contents of general register 0 are logically compared with the contents of the first doubleword of the currently addressed node. If the register operand is low, the contents of general registers 0 and 1 are interchanged with those of the node, and a unit of operation is completed. If the register operand is high, no additional action is taken, and the unit of operation is completed. If the compare values are equal, general registers 2 and 3 are loaded from the currently addressed node, the instruction is completed, and condition code 0 is set.

In those cases when the value in the first doubleword of the node is less than or equal to the value in the register, the contents of the node remain unchanged. However, in some models, these contents may be fetched and subsequently stored back.

Specifications Independent of Addressing Mode

Access exceptions are recognized only for one node at a time. Access exceptions, change-bit action, and PER storage alteration do not occur for subsequent nodes until the previous node has been successfully compared and updated, and they also do not occur if a specification-exception condition exists.

Resulting Condition Code:

0 Equal compare values at currently addressed node
1 No equal compare values found on path, or no comparison made
2 --
3 In 24-bit or 31-bit mode, bits 32-63 of general register 5 nonzero and bits 32-63 of general register 0 negative; in 64-bit mode, general register 5 nonzero and general register 0 negative

Program Exceptions:

- Access (fetch and store, nodes of tree)
- Specification

Programming Notes:

1. An example of the use of UPDATE TREE is given in “Sorting Instructions” in Appendix A, “Number Representation and Instruction-Use Examples.”

2. For use in sorting in the 24-bit or 31-bit addressing mode, when equal compare values have been found, the contents of bit positions 32-63 of general registers 1 and 3 can be appropriate (depending on the contents of the tree) for the subsequent execution of COMPARE AND FORM CODEWORD. The contents of bit positions 32-63 of general register 2, shifted right 16 bit positions, can be similarly appropriate, and they can provide for

minimal recomparison of partially equal keys. The same applies in the 64-bit addressing mode except to the contents of bit positions 0-63 of the registers and with the contents of bit positions 0-63 of general register 2 shifted right 48 bit positions. Refer to “Sorting Instructions” on page A-51 for a discussion of trees and their use in sorting.

3. The program should avoid placing a nonzero value in bit positions 32-38 of general register 5 when in the 24-bit addressing mode. If any bit in bit positions 32-38 is a one, the nodes of the tree will not be examined successively.

4. When bits 32-63 of general register 0 are negative in the 24-bit or 31-bit addressing mode, or when bits 0-63 are negative in the 64-bit mode, and provided that the tree has been updated properly previously, the node represented by general registers 0 and 1 either is the node or is equal to the node (equal keys) that would be selected if the unit of operation continued. In this case, ending the unit of operation and setting condition code 3 is a faster means of selecting an appropriate node because it does not require further examination and updating of the tree.

5. Setting condition code 3 provides improved performance when the replacement record is equal to the old winner and, more importantly (since the first case can be detected by means of the condition code of CFC), when the update path contains a negative codeword, indicating equality with the old winner.

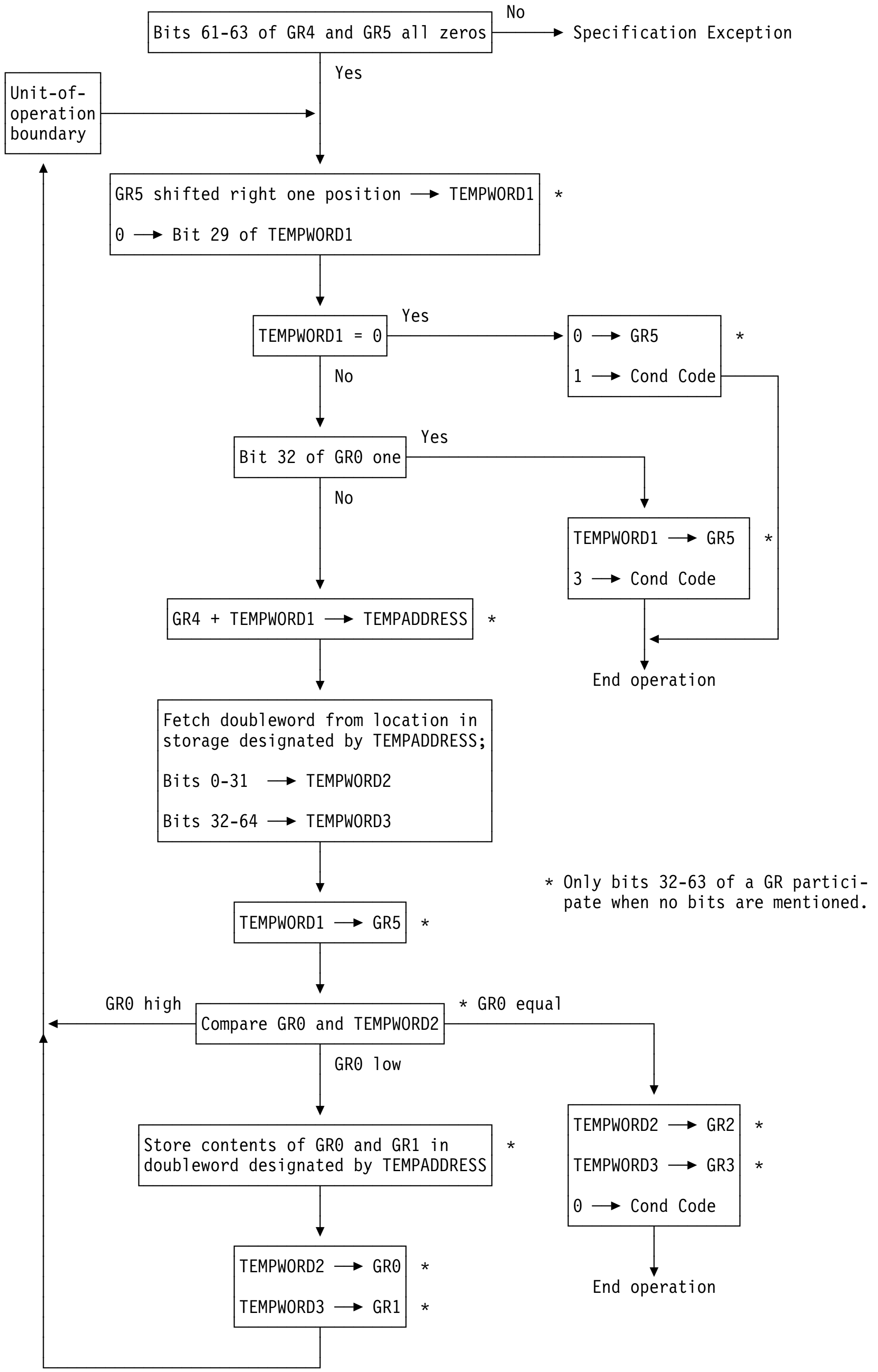
6. In those cases when the codeword in the node is less than or equal to the codeword in general register 0, depending on the model, the contents of the node may be fetched and subsequently stored back. As a result, any of the following may occur for the storage location containing the node: a PER storage-alteration event may be recognized; a protection exception for storing may be recognized; and, provided no access exceptions exist, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.

7. Special precautions should be taken when UPDATE TREE is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

8. Further programming notes concerning interruptible instructions are included in “Interruptible Instructions” on page 5-21.

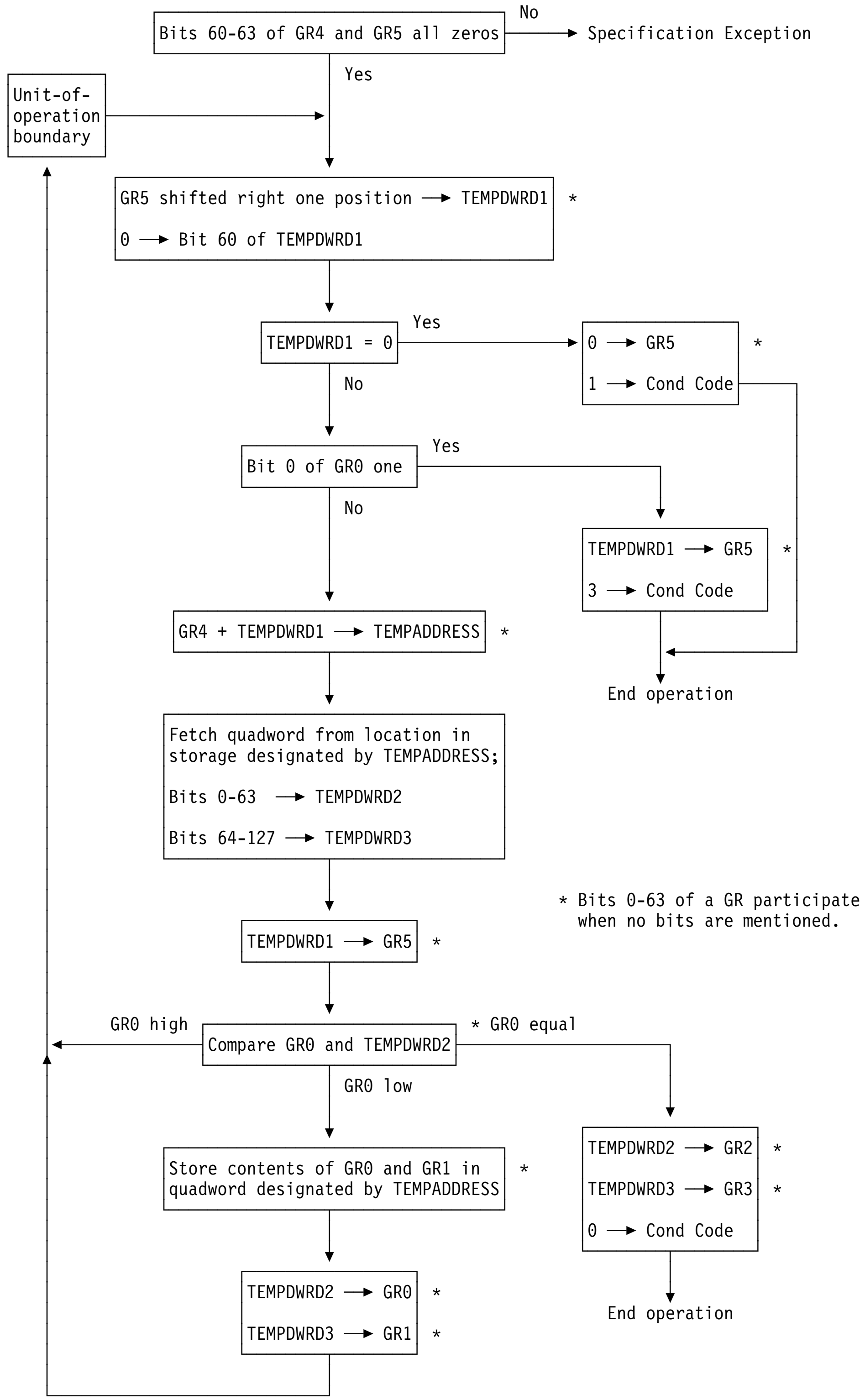
9. The storage-operand references for UPDATE TREE may be multiple-access references. (See “Storage-Operand Consistency” on page 5-87.)

10. Figure 7-77 on page 7-194 is a summary of the operation of UPDATE TREE in the 24-bit or 31-bit addressing mode, and Figure 7-78 on page 7-195 is a summary of the operation in the 64-bit addressing mode.



* Only bits 32-63 of a GR participate when no bits are mentioned.

Figure 7-77. Execution of UPDATE TREE in the 24-Bit or 31-Bit Addressing Mode



* Bits 0-63 of a GR participate when no bits are mentioned.

Figure 7-78. Execution of UPDATE TREE in the 64-Bit Addressing Mode