1 Preface

This manual contains detailed individual descriptions of all Assembler instructions from the instruction set of the central processing units supported by the BS2000 operating system.

Assembler instructions are those instructions in the instruction set which users may use without restrictions to write Assembler programs. Moreover, these instructions are "safe" in the sense that complete protection of hardware components and the BS2000 operating system is ensured when they are executed.

1.1 Target group

The manual was written for users who write, use or update programs in the Assembler or macro language in BS2000. Users require basic knowledge of the operating system and Assembler.

1.2 Summary of contents

The Assembler instruction descriptions follow a uniform pattern. For each instruction the following is described:

- its function
- its Assembler format, i.e. how to write it in the Assembler language
- its machine format, i.e. how it is represented in the CPU
- its execution sequence in detail
- any condition code values which it sets
- possible program interrupts when it is executed.

In addition, most of the instruction descriptions also include

- programming notes and
- one or more examples.

The instructions themselves are divided into 4 groups:

- General Instructions (Chapter 3)
- Decimal Instructions (Chapter 4)
- Floating-Point Instructions (Chapter 5)
- ESA Instructions (Chapter 6)

Within these groups the instructions appear alphabetically according to their mnemonic name.

Chapter 2 contains basic considerations.

1.3 Changes since the last version of the manual

Description of ESA support in Chapter 2 (2.1.3, 2.1.4, 2.2.2) and in the new Chapter 6 (ESA instructions).
Instruction lists in Appendix 7.2 and 7.3 now include ESA instructions.

Access to shared data in multiprocessor systems is described in Appendix 7.6.

2 Basic considerations

2.1 Addressing main memory

The main memory can be considered as a sequence of individual bits. This sequence is divided into 8-bit units which are called bytes. Each byte is associated with a unique integer, called an address, which identifies it. Successive bytes have successive addresses. The value range of the addresses, the "address space", starts at 0 and ends at a system-specific upper limit.

2.1.1 Virtual addresses

All Assembler instructions described in this manual exclusively use so-called **virtual** addresses and only process operands whose addresses are virtual. The instructions themselves are likewise addressed virtually. In the central processing units themselves, however, there are two further types of addresses, namely absolute addresses and real addresses. These are only used beneath the Assembler instruction level, however, and can neither be seen nor influenced in an application program.

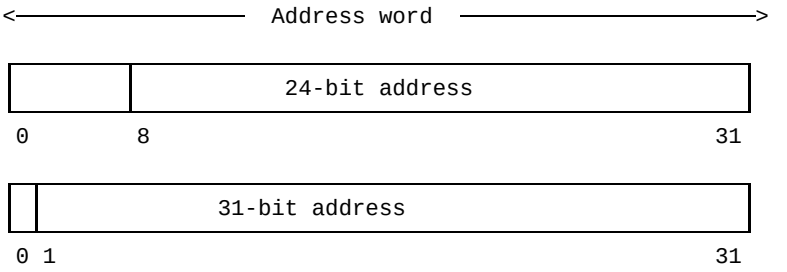
Virtual addresses are **allocated** exclusively for a single program by the operating system when a program is loaded or at its specific request. The allocation takes place in multiple of **pages**; pages are address spaces with a length of 4096 (2^{12}) bytes and a start address which is divisible by 4096 without a remainder. A program is only allowed to read-access or write-access virtual addresses which have been allocated to it. If it tries to access a non-allocated address, e.g. due to a programming error, the access is ignored and a program interrupt occurs due to an address translation error (see section 2.4).

2.1.2 24-bit and 31-bit addresses

Addresses occur in two different lengths: 24-bit addresses and 31-bit addresses.
A 24-bit address can identify 16 777 216 bytes (16 megabytes) of (virtual) address space; a 31-bit address can identify 2 147 483 648 bytes (2 gigabytes).

Both 24-bit and 31-bit addresses are allocated as so-called **address words**, right-justified, in 4-byte main memory areas or general-purpose registers. Unless otherwise stated, the 8 bits to the left of a 24-bit address and the single bit to the left of a 31-bit address are set to 0.

The bit positions in a 24-bit address are numbered from 8 to 31, those in a 31-bit address from 1 to 31:



2.1.3 Addressing modes

The length of addresses, and hence the size of the address space usable in an application program, is determined by the **addressing mode**. There are two addressing modes: the 24-bit addressing mode and the 31-bit addressing mode. At any given time the central processing unit is in one or other of these two modes, and thus generates either 24-bit addresses or 31-bit addresses. As a result, those Assembler instructions that allocate addresses explicitly (e.g. the LA instruction) react differently depending on which addressing mode they are executed in. This manual describes these differences in detail for each instruction involved.

V11.0 and higher of BS2000 support a **new addressing mode**, known as the **AR mode** (access register mode) to expand the virtual address space;
(see Chapter 6, AR Mode).
New hardware (ESA systems) creates the opportunity of using more virtual address spaces for data. These **ESA** (Enterprise System Architecture) systems support the address space as before - now known as **program space** - plus additional **data spaces**.
BS2000 can use the expanded memory space only if it receives notification to the effect that it can use the expanded register set (access registers, see 2.2.2 and Chapter 6), i.e., operate in AR mode (see SAC instruction, Chapter 6)

2.1.4 Instruction addresses, instruction continuation addresses

Normally, an application program consists of instructions and data. Both are stored in main memory, and both have (virtual) addresses. The address of an instruction, the **instruction address**, is the address of its first byte; in all instructions, this byte contains the so-called operation code. Each time an instruction is executed, the central processing unit computes the **instruction continuation address**; this is the instruction address incremented by the length (in bytes) of the current instruction. Unless the current instruction is a branch instruction whose branch condition is satisfied, the instruction continuation address is made the instruction address following execution of the current instruction; processing then continues with the next instruction, or, alternatively, at the address designated in the branch instruction as branch address. Even in AR mode, the instruction address is always located in the program space. If the instruction is a branch instruction, the corresponding access register is not evaluated, which means that it is not possible to branch to an address space.

2.1.5 Operand addresses, address computation

All instructions read (write) their operands either from (to) registers or from (to) main memory. In the case of register operands, the corresponding register number is defined in an R field of the operand address. In the case of main memory operands, the main memory address is computed from two components (three for RX instructions): the **base address**, the **displacement address** and, if applicable, the **index address**.

Base and index addressing enable indirect ("pointer") access to operands; displacement addressing enables addressing relative to a base or index address. Base addressing especially enhances the portability of a program section within the address space of an application program. Usually, a base address is the start address of a largish area of (logically contiguous) data or instructions; the displacement of an individual item from the start of this area (up to 4095 bytes) is then used as the displacement address. Index addressing, which is permitted in conjunction with RX instructions, enables doubly indirect access to operands, e.g. to items in a table within a table.

The effective address of a main memory operand is computed as the sum of

- the base address, i.e. the 32-bit binary number in the general-purpose register defined by the B field of an operand address (base register), plus
- the displacement address, i.e. the 12-bit binary number which is specified directly in the D field of an operand address, plus
- the index address, i.e. the 32-bit binary number in the general-purpose register defined by the X field in an operand address (index register). However, this sort of index address is only available in conjunction with the so-called RX instructions.

The addends are treated as unsigned binary numbers; any numbers carried beyond the highest-order binary position are ignored. The sum is truncated to the 24 or 31 lowest-order bits, depending on the addressing mode used, and the uppermost 8 bits or 1 bit is set to 0. The result is then the (virtual) address of the operand, which is in most cases the address of its first (highest-order) byte.

When the B field or the X field (or both) of an operand address is equal to 0, the corresponding components are not included in the addition operation. It is therefore impossible to use general-purpose register 0 for base addressing or index addressing.

In **AR mode** (see 2.1.3 and Chapter 6), the effective address is computed in the same way as before (base address + displacement address + index address), but the access register is not taken into account (see 2.2.2 and Chapter 6).

2.1.6 Alignment on halfword, word and doubleword boundaries

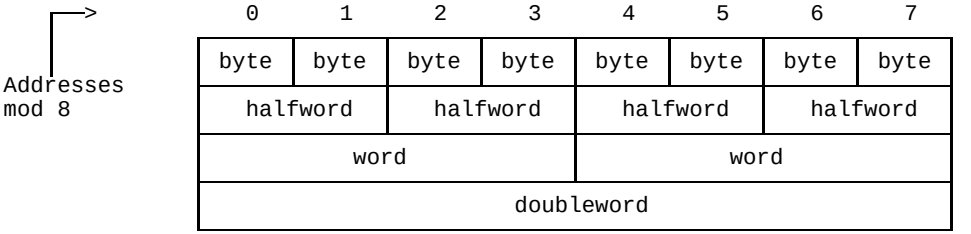
In addition to bits and bytes, the main memory elements **word**, **halfword** and **doubleword** are also used.

A halfword is a 2-byte main memory area with a start address which is divisible by 2 without remainder, i.e. an even number. Similarly, a word is a 4-byte main memory area whose start address is divisible by 4 without remainder, and a doubleword is an 8-byte main memory area with a start address divisible by 8 without remainder. Conversely, an address divisible by 2, 4 or 8, respectively, is commonly known as a "halfword boundary", "word boundary" or "doubleword boundary".

In all central processing units, all instructions must be aligned on a halfword boundary; this alignment is implemented automatically by Assembler [1]. The operands of many instructions, moreover, must satisfy an alignment condition. Note that although this second requirement applies only to certain central processing units (operand misalignment in the L instruction, say, would cause an older central processing unit to interrupt the program with weight ’5C’, whereas the newer central processing units would process the operand), it is represented in this manual as it applies to the most restrictive central processing unit. For the sake of performance, it is advisable to align these operands on the appropriate boundaries even in programs that run on the newer systems (see manual "ASSEMBH Reference Manual"). An Assembler program that satisfies the alignment conditions for the operands as described in this manual therefore results in maximum portability.

If an instruction presupposes that one of its operands is contained in a halfword, word or doubleword, we say that this operand has to be "aligned on" a halfword boundary, word boundary or doubleword boundary, depending on the case involved. For example, all instructions require of binary numbers that the main memory areas for these numbers be aligned. The alignment rule also applies to all floating-point numbers in main memory, but not to decimal numbers and character fields.

This yields the following relations:



2.2 Registers

Almost all instructions require that one or more of their operands be contained in a "register". Registers are storage areas with very high access speed which are independent of main memory. There are three kinds of register: general-purpose registers, access registers and floating-point registers.

2.2.1 General-purpose registers

Central processing units have 16 general-purpose registers at their disposal, numbered from 0 to 15. Each register is 32 bits long, and can therefore accommodate 4 bytes or one word. The general-purpose registers are used for base and index addressing or as accumulators in arithmetic operations. When used as accumulators, they are defined in instructions by explicitly specifying their register numbers; however, some instructions, such as TRT, define their general-purpose registers implicitly.

Many instructions use the contents of two adjoining general-purpose registers as operands. (These are then referred to as a **general-purpose register pair**.) In this case, the first of the two registers (with their higher-order part of the operand) must always be an even-numbered general-purpose register, and the second (with the lower-order operand) is the adjoining odd-numbered general-purpose register. For operands of this sort the even-numbered register is specified in the R field.

The general-purpose registers can also be used for the base addressing or index addressing of operands. In these cases, the B or X field of an operand address defines which register is used. However, the value 0 in a B or X field does not define general-purpose register 0 as the base register or index register; instead, it defines that base or index addressing is **not** to take place when computing the effective operand address. For this reason, general-purpose register 0 cannot be used as a base register or index register.

2.2.2 Access register

ESA systems have 16 access registers (ARs), numbered 0 to 15. Each register is 32 bits long, and can therefore accommodate 4 bytes or one word. The access registers are used to access data spaces.
The 16 access registers (ARs) are uniquely assigned to the general-purpose registers. If the B field (base register) of an operand address of contains the value 0, general-purpose register 0 is not used to compute the effective operand address, nor is access register 0 used to address a data space.

2.2.3 Floating-point registers

Central processing units have 4 floating-point registers at their disposal, which can be referenced only by floating-point instructions and used only for floating-point numbers. Floating-point registers are defined by the numbers 0, 2, 4 or 6 in an R field of the floating-point instructions. Each floating-point register is 64 bits long and can accommodate a short or a long floating-point number. Short floating-point numbers are stored in the leftmost 32 bits of a floating-point register; in all floating-point instructions with short floating-point operand the rightmost 32 bits are ignored or left unchanged. For extended (128-bit) floating-point numbers, two adjoining floating-point registers are used, i.e. a **floating-point register pair**. The floating-point register pair can be either floating-point registers 0 and 2 or floating-point registers 4 and 6. In these cases, the number 0 or 4 must be specified in the R field for the extended floating-point operands.

2.3 Condition code

Most instructions "set the condition code", i.e. during execution they create a value in an internal hardware register with the name "condition code". The condition code (abbreviated CC) is 2 bits long and can be set to the values 0_{10} or 1_{10} or 2_{10} or 3_{10}; it will retain a set value until it is set to a different value by a subsequent instruction.

The condition code is used most frequently in compare operations. All compare instructions set the condition code in accordance with the compare result they obtain, i.e. to 0 if the two compared operands are identical and to 1 or 2 if the first operand is less than or greater than the second. Following a compare instruction, the next instruction can query the set condition code and, depending on its value, trigger appropriate actions.

Each instruction description shows whether the instruction sets values in the condition code, and if so, which values these are and what they mean. However, all instructions (peculiarities arise with the instructions AL, ALR, SL, SPM and TM) that set the condition code use the following common pattern for setting one of the four possible values:

Value of CC	Meaning
0	The result of the instruction is =0. After a compare operation, this means that the first operand is identical to the second operand.
1	The result of the instruction is <0. After a compare operation, this means that the first operand is less than the second operand.
2	The result of the instruction is >0. After a compare operation, this means that the first operand is greater than the second operand.
3	An overflow occurred during command execution.

To help the reader remember this table, wherever a condition code explicitly appears in this manual we have included a mnemonic explanation which also suggests which query instruction, if any, should be entered. For example, we have written "the condition code is set to 2~High" to indicate that the instruction "Branch when High" may be used to query this condition code value 2.

2.4 Program interrupts

If an abnormal condition is detected when an instruction is executed (e.g. wrong operand or illegal data constellation), a program interrupt occurs. Unless special arrangements for this eventuality have been made in the application program, the BS2000 operating system will terminate the application program. If, however, a so-called STXIT task has been defined in the application program (prior to the first program interrupt), BS2000 will activate this task each time a program interrupt occurs, so that the application program can handle the interrupt appropriately.

Every potential program interrupt is identified by its **interrupt weight**. The interrupt weight is a two-digit hexadecimal number which is put in general-purpose register 3 for the STXIT task (if this task exists) or output to SYSOUT at program termination (if no STXIT task exists).

The following general pattern applies:

Type of program interrupt	Interrupt weight	General causes
Address translation error	48	An operand contains a virtual address which is not allocated for the application program. It is therefore not possible to read- or write-access this operand.
Privileged operation	54	The instruction invoked is not an Assembler instruction, but neither is it illegal.
Wrong operation code	58	An illegal instruction was invoked.
Addressing error	5C	A constraint on the instruction (e.g. an alignment condition) is not satisfied.
Data error	60	A decimal operand does not contain a correct, packed decimal number, or two decimal operands incorrectly overlap.
Exponent overflow	64	The resulting characteristic in a floating-point operation is > 127.
Division error	68	Division by zero or quotient too large.
Significance	6C	The resulting mantissa in a floating-point operation = 0.
Exponent underflow	70	The resultant characteristic of floating-point operation < 0.
Decimal overflow	74	The result of a decimal operation is too large.
Fixed-point overflow	78	The result of a fixed-point operation is too large.

The specific cause of each program interrupt is shown in each individual instruction involved; exceptions include the program interrupt types *privileged operation* and *wrong operation code*, which are not instruction-specific.

When a program interrupt occurs, execution of the instruction that caused the interrupt is usually not finished and the result of the instruction is incorrect.

Maskable program interrupts, program mask

Program interrupts due to fixed-point overflow, decimal overflow, exponent underflow and significance are **maskable**. This means that an application program can determine whether a program interrupt should take place in these cases. Masking is performed in the **program mask**, a 4-bit internal register in the central processing unit. Each of the 4 bits is assigned one of the above-mentioned program interrupts. A bit value of 1 means that a program interrupt will take place when the corresponding cause occurs; a bit value of 0 means that the interrupt will not take place. BS2000 presets all 4 bits to 1 at application startup time so that the 4 program interrupts will take place by default. However, the application program can change these presettings with the instruction SPM, and can suppress any of these program interrupt types by setting one or all four of the corresponding bits to zero.

The bits of the program mask have the following meaning:

Bit in program mask	Meaning
0	Fixed-point overflow
1	Decimal overflow
2	Exponent underflow
3	Significance (mantissa=0)

Note

The BS2000 macro STXIT, which can be used to define STXIT tasks for program interrupt handling, is described together with its parameters in "BS2000 Executive Macros" [3].

2.5 Data types

Assembler instructions make use of the following data types: character, character field, binary number, bit field, decimal number and floating-point number. Of these, the data types "decimal number" and "floating-point number" are described in Chapters 4 and 5, where the decimal or floating-point instructions that use them are explained. The others are described below.
(We have followed the common practice of using the data type itself as a name when in fact what is meant is an instance of the data type: e.g. instead of the clumsy phrase "data of data type X" we simply write "X".)

2.5.1 Characters and character fields

The **character** data type is intended for single characters which, for example, come from a keyboard or are output to printer. Examples of data items of this sort include the letters of our alphabet or punctuation marks in text.

Each character is represented by a single byte. The mapping of a character on the 8 bits of a byte is defined by the **EBCDI code**. This code defines, for example, that the character 'A' has the binary representation $(11000001)_2$, i.e. hexadecimal $(C1)_{16}$.
A complete EBCDIC table can be found in the appendix.

The **character field** data type is intended for a set of contiguous characters (= data of data type "character"), e.g. for a linguistic word or even an entire text. A character field is represented in main memory in consecutive bytes. It is identified to instructions that process character fields by two specifications: the address of its first (highest-order) byte, and its "length", i.e. the number of bytes included in the character field.

Comparing characters and character fields

Comparison of two operands of data type "character" or "character field" takes place bit by bit from left to right; characters and character fields are treated as bit sequences. Bit positions equidistant from the start of their respective operands are called "opposing". The comparison ends when either *all* opposing bit positions in both operands are the same, or two opposing bit positions are different. In the first case, the operands are "identical". In the second case, they are "nonidentical"; the operand whose most recently compared bit position is =0 is considered "less than" the other operand, which is in turn "greater than" the first.

2.5.2 Binary numbers

Along with the data types "decimal number" and "floating-point number" the **binary number** data type is intended for data which is to be handled arithmetically, e.g. added together.

Binary numbers are base 2 integers with an assumed binary point to the right of the lowest-order binary position. Each binary position in a binary number is represented by a bit, namely, from left to right in descending order of value.

Binary numbers occur in various lengths. The most frequent length is 32 bits long and is used above all for fixed-point numbers (see below). However, there are also instructions for 16-bit and 64-bit binary numbers.

Aligning binary numbers

Depending on their length, binary numbers require either 2, 4 or 8 consecutive bytes in order to be stored in main memory. The address of the first byte must be aligned, i.e. divisible by 2, 4 or 8, without remainder. We also say that binary numbers have to be aligned on halfword, word or doubleword boundaries.

Signs of binary numbers

Binary numbers can be either signed or unsigned. Signed binary numbers are referred to as fixed-point numbers; unsigned binary numbers do not have a name of their own. The special features of both types are explained below.

Unsigned binary numbers

With unsigned binary numbers, all binary positions are used to represent the amount; as a result, when arithmetic and compare operations are performed, all binary positions are involved in the operation. The terms "positive" and "negative" are irrelevant for binary numbers of this sort.

The value range of b-bit unsigned binary numbers ranges from 0 to 2^b-1. The (least) unsigned binary number with the value 0 is represented entirely by null bits; the (greatest) unsigned binary number with the value 2^b-1 is represented entirely by 1 bits.

Signed binary numbers (fixed-point numbers)

With signed binary numbers - called **fixed-point numbers** - the highest-order binary position is used for the sign; the binary positions to the right of the sign represent the numeric value of the fixed-point number:

– Positive fixed-point numbers are represented by their (absolute) value and have a null bit at the highest-order binary position.

– Negative fixed-point numbers are represented by the **twos complement** (absolute) value and have a 1 bit at the highest-order binary position.

The twos complement of a number is defined as the difference between 0 and the amount of this number. Any carry over beyond the highest-order binary position is ignored.

The table below shows some selected values from the value range of a 32 bit fixed-point number (in binary and hexadecimal format):

Fixed-point number	Binary representation	Hex. representation
	<———32 bits———>	
0	0000.....0000	00 00 00 00
+1	0000.....0001	00 00 00 01
+2	0000.....0010	00 00 00 02
.		
.		
$+2147483647 = +2^{31}-1$	0111.....1111	7F FF FF FF
-1	1111.....1111	FF FF FF FF
-2	1111.....1110	FF FF FF FE
.		
.		
$-2147483648 = -2^{31}$	1000.....0000	80 00 00 00

Fixed-point numbers have the following essential attributes:

– The value range of positive b-bit fixed-point numbers is from 0 to 2^{b-1}-1; the value range of negative, b-bit fixed-point numbers is from -1 to -2^{b-1}. Thus, for 32-bit fixed-point numbers the value range lies between -2^{31}=-2147483648 and $+2^{31}$-1=+2147483647. Therefore, the least negative fixed-point number does not have a positive pendant.

– The set of fixed-point numbers greater than 0 is one member larger than the set of fixed-point numbers less than 0.

– Negative zero does not exist.

– The most significant binary position of a (positive or negative) fixed-point number is the highest-order binary position other than the sign bit.

Note

Alternative methods for forming the twos complement of a binary number include:

1. Inverting all bit positions of the binary number, adding +1 to the result and ignoring any carry over beyond the highest- order binary position.
2. Inverting all bit positions to the left of the lowest-order 1 and leaving unchanged the lowest-order 1 and all binary positions to the right of it (they are all =0).

Signed and unsigned binary arithmetic

There are two different kinds of binary arithmetic: signed and unsigned (or logical). In signed binary arithmetic, the highest-order binary position of each operand and of the result is handled separately as a sign, whereas in unsigned binary arithmetic the highest-order binary position is handled in exactly the same way as the other binary positions. The following differences exist with regard to the individual arithmetic operations.

Addition and subtraction of binary numbers

Signed addition is performed by adding all the binary positions of both addends, including the sign positions. If one of the addends is shorter than the other, it is treated as if it were padded to the length of the longer addend, using binary digits which are identical to the value of the sign position.

Unsigned addition likewise consists in adding all the binary positions of both addends. If, however, one addend is shorter than the other, it is treated as though it were padded to the left to the length of the longer addend, using binary digits with the value 0. All address computations
are performed by means of unsigned addition.

Signed (and unsigned) subtraction is identical to the signed (or unsigned) addition of the ones complement of the second operand and the number 1 to the first operand. (The ones complement of a binary number is obtained by inverting all bit positions in the number.)

The difference between signed and unsigned addition or subtraction lies in the way the result is interpreted:

– With unsigned addition or subtraction, the result is interpreted as an unsigned binary number; the condition code shows whether the result is =0 or ≠0 and whether an overflow did or did not occur from the highestorder binary position, i.e. a carry over beyond binary position 0.

– With signed addition, or subtraction, the result is interpreted as a signed binary number (fixed-point number); the condition code shows whether the result = 0, <0 or >0, or whether a fixed-point overflow occurred.
  A fixed-point overflow occurs when any binary position overflow *to* the sign position of the result is not equal to the binary position overflow *from* the sign position. Arithmetically speaking, when 32-bit fixed-point numbers are used, this means that the result is greater than $+2^{31}-1$ or less than -2^{31}. With fixed-point overflow, the condition code is set to 3~Overflow; in addition, a program interrupt occurs, provided the bit for fixed-point overflow is set to 1 in the program mask (default value in BS2000).

Left or right shifting of binary numbers

Signed shifting of a binary number handles the sign position separately, whereas unsigned shifting does not.

In signed shifting, the sign position always remains the same and only those binary digits from bit position 1 are shifted. In shift right, binary positions freed to the left are filled with the bit value of the sign position; in shift left, binary positions freed to the right are filled with 0. If, during shift left, significant binary positions (i.e. those other than the sign position) are shifted to the left of bit position 1, fixed-point overflow occurs; the condition code is set to 3~Overflow, and in addition a program interrupt occurs if the fixed-point overflow bit in the program mask is set to 1 (default value in BS2000).

In an unsigned shift left or unsigned shift right, all binary positions, including the sign position, are shifted. In shift right, binary positions freed to the left are padded with 0; in shift left, binary positions freed to the right are padded with 0. The condition code is not changed by unsigned shifting.

Comparison of binary numbers

Signed comparison of two binary numbers is performed in the same way as a signed subtraction operation for which the result is not stored. The condition code is set to 0~Equal, or 1~Low, or 2~High, depending on whether the first operand is equal to or less than or greater than the second. Fixed-point overflow cannot occur.

Unsigned comparison of two binary numbers consists of a bit-by-bit comparison of both numbers from left to right. The comparison terminates either when the two operands have been processed or if two opposing bits are different. If the operands are identical, the condition code is set to 0~Equal; if they are not identical, the condition code is set to 1~Low or 2~High, depending on whether the last bit position compared in the first operand was =0 or =1.

2.5.3 Bit field

The **bit field** data type is a data type for sequences of single-bit values. Each bit value is represented in a bit position. In bit field instructions, the individual bit positions are handled independently of the remaining bit positions. Bit fields start or end at byte boundaries. The bit positions in a bit field are usually numbered from left to right starting at 0, but this convention is immaterial to the central processing unit.

One frequent application of the bit field data type is its use in **masks**. Masks are used for selecting individual bits or bytes in a register or in main memory or in the subsequent actions of an instruction. Their concrete meaning and effect are described under those individual instructions that make use of masks.

2.6 Instruction format

Every instruction consists of two parts:

1. the operation code, which determines the instruction's action and
2. the specification of its operands.

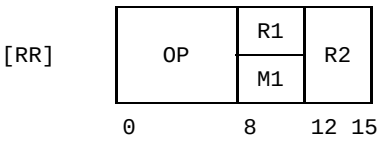
All instructions must be aligned on halfword boundaries in main memory. Depending on the type of instruction involved, instructions are either 2, 4 or 6 bytes long.

Instruction types

The following general instruction types exist:

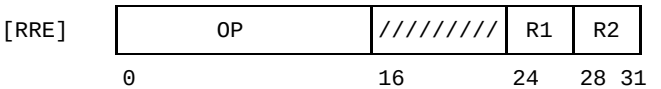
1. Instruction type RR

For instructions with two register operands or with one register operand and one mask.



2. Instruction type RRE

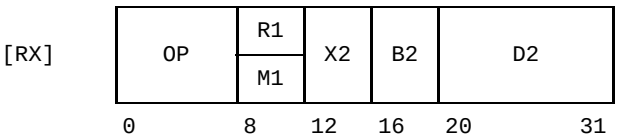
For instructions with extended operation code and two register operands.



In instructions of this type, bit positions 16 to 23 are ignored.

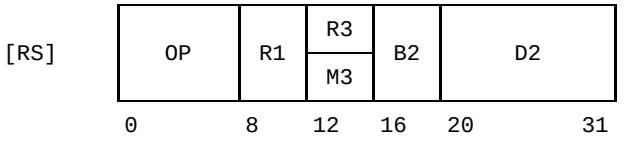
3. Instruction type RX

For instructions with one register operand or with one mask and one indexed main memory operand.



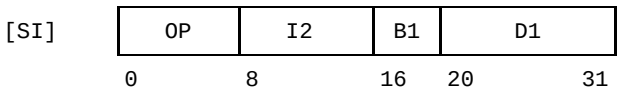
4. Instruction type RS

For instructions with two register operands and one main memory operand, or with one register operand and one main memory operand and one mask.



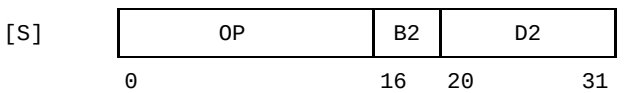
5. Instruction type SI

For instructions with one main memory operand and one direct operand.



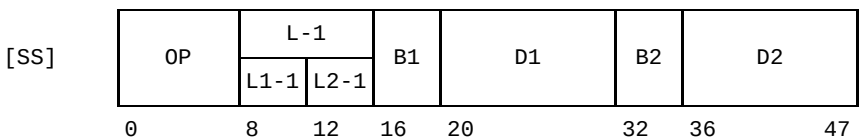
6. Instruction type S

For instructions with extended operation code and one main memory operand.



7. Instruction type SS

For instructions with two memory operands of identical or nonidentical operand length. The L, L1 and L2 fields contain the length minus 1.



Legend

The abbreviations on the formats above have the following meanings:

Name	Length	Meaning
OP	8/16 bits	Operation code
R1,R2,R3	4 bits	General-purpose,access or floating-point register
M1,M3	4 bits	Mask
B1,B2	4 bits	Base register
X1,X2	4 bits	Index register
D1,D2	12 bits	Displacement address
I2	8 bits	Direct operand
L-1	8 bits	Operand length minus 1
L1-1, L2-1	4 bits	Operand length minus 1

The operation code (OP) is an instruction-specific hexadecimal number of two or four characters. It is named in every instruction and is represented in Assembler notation, e.g. by X'D1' for the instruction MVN. The first two bit positions of the operation code determine the length of the instruction as follows:

Bit pos. 0 and 1 of the operation code	Instruction type(s)	Length of instruction
00	RR	2 bytes
01	RX	4 bytes
10	RRE/RS/S/SI	4 bytes
11	SS	6 bytes

Instruction operands

The operands involved in operation execution are defined to the right of the operation code in instruction-specific fields. Depending on the instruction type concerned, up to three operands are involved in an instruction. In this manual, they are called operand1, operand2 and operand3. The parameter values of these operands are distinguished in the instruction by the suffixes "1", "2" and "3".

Instruction operands may be stored either directly in the instruction, or separately from the instruction either in a (general-purpose or floating-point) register or in main memory. Depending on the method used, they are called direct operands, register operands or main memory operands.

A **direct operand** is represented as a bit field in the instruction using either an M field or an I field.

A **register operand** is defined by specifying the corresponding 4-bit register number in an R field. The instruction then defines whether the register involved is a general-purpose register, an access register or a floating-point register.

A **main memory operand** is defined by the address (of its highest-order byte) and by its length (in bytes). Its address is either obtained from an address computation or is taken as a ready-made address from a general-purpose register. Address computation consists of the unsigned addition of a displacement address (defined in a D field) and the contents of one or two general-purpose registers whose register numbers are defined in a B field and an X field (see above, "Address computation"). If so-called symbolic addresses of instructions and data are used, the Assembler [1] computes the address components B and D itself.

The length of a main memory operand is defined either implicitly *by* the instruction or explicitly *in* the instruction. In the case of implicit length, it is represented in the instruction description; in the case of explicit length (in SS-type instructions), the operand length is defined in the instruction by its value *minus 1*, using one of the fields indicated in the above diagram by "L-1" or "L1-1" or "L2-1". (The Assembler [1] generates the contents of these length fields automatically by reducing the operand length by 1.)

3 General instructions

Add

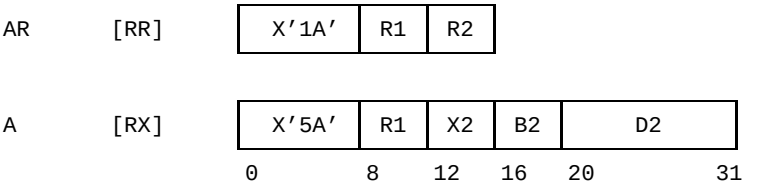
Function

The instructions AR and A perform signed addition of two 32-bit fixed-point numbers. The condition code is set in accordance with the value of the sum.

Assembler formats

Name	Operation	Operands	Remarks
	AR A	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats



Description

The AR instruction causes signed addition of the contents of general-purpose register R2 and that of general-purpose register R1. The A instruction causes addition of the word addressed by D2(X2,B2) and the contents of general-purpose register R1. Both operands are treated as 32-bit fixed-point numbers. The sum is also a 32-bit fixed-point number and replaces the original contents of general-purpose register R1.

Fixed-point overflow results when the sum is greater than 2^{31}-1 or less than -2^{31}. In this case, the result in R1 is too large or too small by 2^{32}; the condition code is then set to 3~Overflow and a program interrupt takes place, provided the fixed-point overflow bit is set to 1 in the program mask (default in BS2000).

Condition code

0~Zero	sum = 0
1~Minus	sum < 0
2~Plus	sum > 0
3~Overflow	fixed-point overflow

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	A: Read access of operand 2 illegal.
Address error	X'5C'	A: D2(X2,B2) not a word boundary.
Fixed-point overflow	X'78'	Sum > +2^{31}-1 or < -2^{31}.

Programming notes

Fixed-point overflow occurs when a binary position overflow to the sign position is not equal to the binary position overflow from the sign position. The result in register R1 then has an incorrect sign.

Examples

```
Name        Operation  Operands
            .
Example1    L          15,=F'-2147483647'   Reg 15: X'80000001' =-2**31+1
            A          15,=F'-1'            Reg 15: X'80000000' =-2**31
*                                           CC: 1~Minus
            .
Example2    LM         15,0,=F'2147483647'  Reg 15: X'7FFFFFFF' =+2**31-1
*           LA         0,1                  Reg 0 : 1
            AR         15,0                 Reg 15: X'80000000' =-2**31
*                                           CC: 3~Overflow and possibly
*                                           program interrupt due to
*                                           fixed-point overflow
            .
```

The condition code 3~Overflow in example 2 indicates that the result is arithmetically incorrect.

Add Halfword

Function

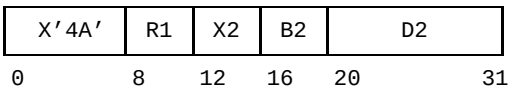
The AH instruction performs signed addition of a 16-bit fixed-point number and a 32-bit fixed-point number.
The condition code is set in accordance with the value of the sum.

Assembler format

Name	Operation	Operands	Remarks
	AH	R1,D2(X2,B2)	D2(X2,B2): halfword boundary

Machine format

AH [RX]



Description

This instruction performs signed addition of the halfword addressed by D2(X2,B2) in main memory and the contents of general-purpose register R1. The register operand is treated as a 32-bit fixed-point number, the halfword operand as a 16-bit fixed-point number, both of them signed. The sum is a 32-bit signed fixed-point number; it replaces the original contents of general-purpose register R1.

Fixed-point overflow results when the sum is greater than 2^{31}-1 or less than -2^{31}. In this case, the result in R1 is too large or too small by 2^{32}; the condition code is then set to 3~Overflow and a program interrupt takes place, provided the fixed-point overflow bit is set to 1 in the program mask (default in BS2000).

Condition code

0~Zero Sum = 0
1~Minus Sum < 0
2~Plus Sum > 0
3~Overflow fixed-point overflow

Program interrupts

Art	Weight	Causes
Address trans. error	X'48'	A: Read access of operand 2 illegal.
Address error	X'5C'	A: D2(X2,B2) not a word boundary.
Fixed-point overflow	X'78'	Sum > $+2^{31}-1$ or < -2^{31}

Programming notes

Fixed-point overflow occurs when a binary position overflow to the sign position is not equal to the binary position overflow from the sign position. The result in register R1 then has an incorrect sign.

Example

```
Name    Operation  Operands
        .
        L          13,=F'16999999'
        AH         13,=H'+1'            Register 13: F'17000000'
*                                       CC 2~Plus
        .
```

In many programs, simple register incrementation of this sort is performed with the instruction LA 13,1(13) rather than with the instruction AH 13,=H'+1', as in this example. However, an LA instruction of this sort would be meaningless in this example: if the program is running in 24-bit addressing mode, the LA instruction does not yield the (presumably desired) result 17000000, but rather a completely different result, namely 222784 (i.e. 17000000 mod 2^{24}). Only if the program is executed in 31-bit addressing mode does LA yield the value 17000000. Addresses of the sort created by the LA instruction are not the same thing as fixed-point numbers.

Add Logical

Function

The instructions ALR and AL perform unsigned (logical) addition of two 32-bit binary numbers.
The condition code is set in accordance with the value of the sum.

Assembler formats

Name	Operation	Operands	Remarks
	ALR AL	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats

Description

The ALR instruction performs unsigned addition of the contents of general-purpose register R2 and the contents of general-purpose register R1. The AL instruction performs unsigned addition of the word addressed by D2(X2,B2) in main memory and the contents of general-purpose register R1.

The sum is a 32-bit unsigned binary number. It replaces the original contents of general-purpose register R1.

All 32 bits of each operand are involved in the addition operation. Any carry over beyond bit position 0 is shown in the condition code.

Condition code

0~Zero Sum =0, no carry over
1~Minus Sum ≠0, no carry over
2~Plus Sum =0, carry over
3~Overflow Sum ≠0, carry over

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	AL: Read access of operand2 illegal.
Address error	X'5C'	AL: D2(X2,B2) not a word boundary.

Programming notes

– The condition code 0~Zero is set only when both operands =0.

– The AL instruction can be used for signed addition of fixed-point numbers which are more than 32 bits long. In this case, AL instructions are used to add the lowest-order word pairs, followed by instruction A to add the highest-order word pairs; if carry over occurs when adding a lower-order word pair, the number +1 must be added to the sum of the next highest word pair (see example 2).

Examples

```
Name        Operation   Operands
            .
Example1    L           15,=F'-1'
            AL          15,=F'1'          Register 15: 0, CC: 2~Plus
            .
Example2    LM          0,1,FPNO1         Addition of two 64-bit fixed-
LOWADD      AL          1,FPNO2+4         point numbers.
            BC          12,HIGHADD
            AH          0,=H'1'           Carry over in right portion.
HIGHADD     A           0,FPNO2
            .
```

Example2 illustrates signed addition of two 64-bit fixed-point numbers FPNO1 and FPNO2. the lower-order word pair is added using AL and the higher-order word pair is added using A. If a carry over occurs when adding the lower-order word pair, +1 must be added to the sum of the higher-order word pair. In the above example, the result is located in general-purpose registers 0 and 1.

Branch and Link

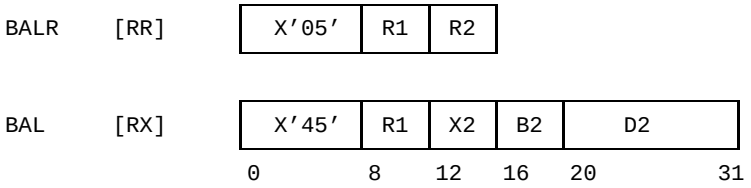
Function

The instructions BALR and BAL store the instruction continuation address in a specified general-purpose register and then branch to a specified address.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	BALR BAL	R1,R2 R1,D2(X2,B2)	

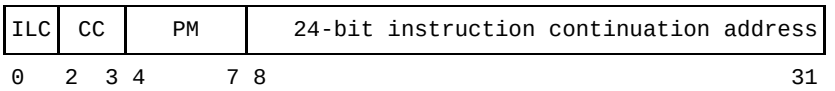
Machine formats



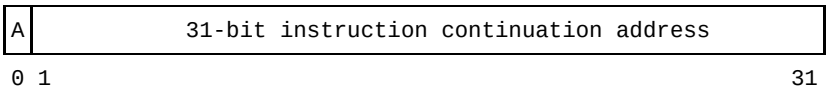
Description

The instructions BALR and BAL first store the instruction continuation address in general-purpose register R1.
The format of the stored address depends on the addressing mode used:

– 24-bit addressing mode:



– 31-bit addressing mode:



Where:

- ILC instruction length code; 01_2 for BALR and 10_2 for BAL.
- CC condition code; current value of CC: 0, 1, 2 or 3
- PM current value of program mask:
  BS2000 default is F_{16}, but this value can be changed by the application programing using SPM.
- A addressing mode; =1 for 31-bit addressing mode.

Once the instruction continuation address has been stored, the BALR instruction branches to the address contained in general-purpose register R2, and the BAL instruction branches to the address D2(X2,B2). (This address is either 24 or 31 bits long, depending on which addressing mode is used.) BALR does not branch if the R2 field is =0. In any case, the current addressing mode remains the same.

The branch address is computed before general-purpose register R1 is changed.

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if the target address is not a halfword address, or if it cannot be accessed, a corresponding program interrupt (with the weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

- To return from a subprogram called with BALR or BAL, you should use the instruction BCR (or BC), it is not possible to use the instruction BSM for this purpose if the subprogram was called by means of BAL and the call took place in 24-bit addressing mode.
- When a BALR instruction is executed directly, the "instruction continuation address" is the instruction address plus 2; with BAL it is the instruction address plus 4. However, if a BALR or BAL instruction is executed using the EX instruction, the instruction continuation address is the instruction address of this EX plus 4.

- Note that in 24-bit addressing mode the instructions BALR and BAL do not create "genuine" 24-bit addresses in general-purpose register R1. Instead, they supply non-address information to the highest-order byte. This makes these instructions unsuitable for use in programs which are designed to run in both 24-bit and 31-bit addressing mode. In programs of this sort, the instructions BAS or BASR should be used instead of BAL or BALR. (The condition code values (CC) supplied by the BAL and BALR instructions, or the values of the program mask (P'Mask), may be obtained in portable form using the instruction IPM).
- Note the following difference between BALR and BAL: With BALR, the branch address is determined by the *contents* of the second operand, whereas with BAL it is determined by the *address* of the second operand.

Branch and Save

Function

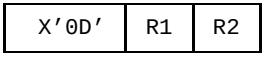
The instructions BASR and BAS store the current addressing mode and the instruction continuation address in a general-purpose register and branch to a specified address while staying in the current addressing mode.
The condition code is left unchanged.

Assembler formats

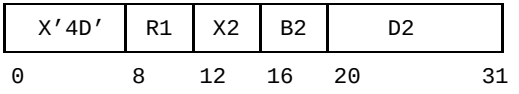
Name	Operation	Operands	Remarks
	BASR BAS	R1,R2 R1,D2(X2,B2)	

Machine formats

BASR [RR]



BAS [RX]



Description

The current addressing mode is stored in bit position 0 of general-purpose register R1, and the instruction continuation address in bit positions 1 to 31 of general-purpose register R1. The 24-bit addressing mode is shown by the value 0 in bit position 0; the 31-bit addressing mode is shown by the value 1 in bit position 0.

With BASR, a branch then takes place to the address contained in general-purpose register R2; with BAS, a branch takes place to the address D2(X2,B2). In both cases, the addressing mode remains the same. This address is either 24 or 31 bits long, depending on the addressing mode used. If, with BASR, the R2 field is =0, no branch takes place; instead, processing continues with the next instruction.

The branch address is determined before general-purpose register R1 is changed.

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if the target address is not a halfword address, or if it cannot be accessed, a corresponding program interrupt (with the weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

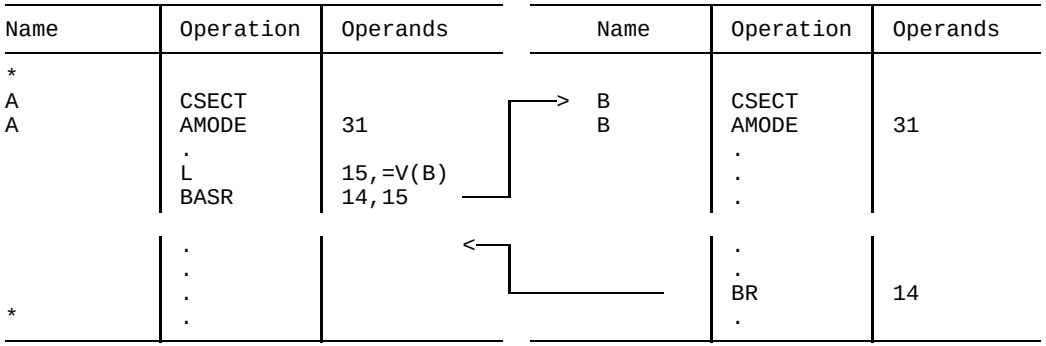
- The instructions BASR and BAS are used for calling a subprogram running in the same addressing mode as the calling program.
- Calling BASR with an R2 field =0 causes only the current addressing mode and the instruction continuation address to be stored in general-purpose register R1, i.e. no branch takes place. This can be used, for example, to obtain a "base address", e.g. in the instruction sequence

```
BASR   3,0
USING  *,3
```

- Normally, the instruction BCR (or BC) is used to return from a subprogram called with BASR or BAS. However, on central processing units which have 31-bit addressing mode at their disposal, this can also be accomplished using the instruction BSM.
- When a BASR instruction is executed directly, the "instruction continuation address" is the instruction address plus 2; with BAS it is the instruction address plus 4. However, if a BASR or BAS instruction is executed using the EX instruction, the instruction continuation address is the instruction address of this EX plus 4.
- The instructions BASR and BAS have the same effect as the instructions BALR and BAL. However, BASR and BAS create genuine instruction continuation addresses, even in 24-bit addressing mode, and they do not supply the highest-order byte in general-purpose register R1 with non-address information (e.g. ILC) which might cause problems in 31-bit addressing mode. Nevertheless, the instructions BASR and BAS are not available in the instruction set of older central processing units.
- Note the following difference between BASR and BAS: With BASR, the branch address is determined by the *contents* of the second operand, whereas with BAS it is determined by the *address* of the second operand.

Example

The BASR instruction can be employed as follows in order to perform a subprogram branch from program section A to program section B, both of which are running in the same addressing mode (24-bit or 31-bit) but are assembled in different assembly units:



```
Name   | Operation | Operands          |      Name | Operation | Operands
*      |           |                   |           |           |
A      | CSECT     |                   |  ------>  | B         | CSECT
A      | AMODE     | 31                |           | B         | AMODE    31
       | .         |                   |           |           | .
       | L         | 15,=V(B)          |           |           | .
       | BASR      | 14,15             |           |           | .
       |           |                   |           |           |
       | .         |          <--------|           |           | .
       | .         |                   |           |           | .
       | .         |                   |           |           | BR       14
*      | .         |                   |           |           | .
```

Branch and Save and Set Mode

Function

The BASSM instruction stores the current addressing mode and the instruction continuation address in a general-purpose register. It then sets a specified addressing mode and branches in this mode to a specified address.
The condition code is left unchanged.
The BASSM instruction is available only in the instruction set of central processing units which have 31-bit addressing mode at their disposal.

Assembler format

Name	Operation	Operands	Remarks
	BASSM	R1,R2	

Machine format

BASSM [RR]

Description

The current addressing mode is stored in bit position 0 of general-purpose register R1, and the instruction continuation address is stored in bit positions 1 to 31 of general-purpose register R1.

Then, if the R2 field ≠0, the addressing mode resulting from bit 0 of general-purpose register R2 is set, and a branch is performed in this mode to the address contained in bit positions 1 to 31 of R2. This address is 24 or 31 bits long, depending on the addressing mode used. If the R2 field =0, no branch takes place; instead, processing continues with the next instruction in the current addressing mode.

In general-purpose registers R1 and R2, a value 0 at bit position 0 indicates 24-bit addressing mode, and a value 1 indicates 31-bit addressing mode.

The branch address is determined before general-purpose register R1 is changed.

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if the target address is not a halfword address, or if it cannot be accessed, a corresponding program interrupt (with the weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

- The BASSM instruction is used to call a subprogram running in an addressing mode which is the same as or different from that of the calling program.
- When a BASSM instruction is executed directly, the "instruction continuation address" is the instruction address plus 2; when, however the BASSM instruction is executed by means of the instruction EX, the instruction continuation address is the address of this EX, plus 4.
- The BASSM instruction is available only on central processing units which have 31-bit addressing mode at their disposal. In order to make an Assembler program independent of the central processing unit on which it is running, the macro ##BASSM is provided in the BS2000 macro instruction set (V9.0 and successors). This macro "switches on" the BASSM instruction. The ##BASSM macro generates the BASSM instruction on systems with 31-bit addressing mode, and the BALR instruction on systems which only have 24-bit addressing mode at their disposal. A complete discussion of the addressing modes and the various forms of program linkage can be found in the manual "Introduction to XS Programming (for ASSEMBLER Programmers)" [2].
- The BSM instruction should be used for returning from a subprogram called with BASSM in order to ensure that the addressing mode of the calling program will again be in effect.

Example

The BASSM instruction can be employed as follows in order to perform a subprogram branch from program section A (running in 24-bit addressing mode) to program section B (running in 31-bit addressing mode):

```
Name   | Operation | Operands          |      Name | Operation | Operands
       |     .     |                   |           |     .     |
A      | CSECT     |                   |---->  B   | CSECT     |
A      | AMODE     | 24                |       B   | AMODE     | 31
       |     .     |                   |           |     .     |
       | L         | 15,=V(B)          |           |     .     |
       | O         | 15,AM31           |           |     .     |
       | BASSM     | 14,15             |           |     .     |
       |     .     |                   |           |     .     |
       |     .     |             <-----|           |     .     |
       |     .     |                   |           | BSM       | 0,14
       | DS        | 0F                |           |     .     |
AM31   | DC        | X'80000000'       |           |     .     |
       |     .     |                   |           |     .     |
```

Branch on Condition

Function

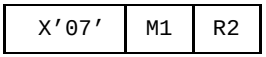
The instructions BCR and BC cause a branch to a specified address depending on the current value of the condition code.
The condition code is left unchanged.

Assembler formats

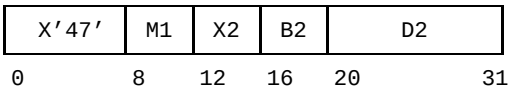
Name	Operation	Operands	Remarks
	BCR BC	M1,R2 M1,D2(X2,B2)	B'0000' ≤ M1 ≤ B'1111' B'0000' ≤ M1 ≤ B'1111'

Machine formats

BCR [RR]



BC [RX]



Description

The "mask" M1 is a 4-bit field. Its 4 bit positions, from left to right, correspond as follows to the 4 possible values of the condition code:

Value of CC	Bit position of mask M1	Value of mask M1
0	0	8
1	1	4
2	2	2
3	3	1

If the condition code has the value *i* at the time the instruction is executed, and bit position *i* in mask M1 =1, the BCR instruction branches to the address contained in general-purpose register R2, and the BC instruction branches to the address D2(X2,B2). This address is 24 or 31 bits long, depending on the addressing mode used. If bit position *i* in mask M1 =0, no branch takes place; instead, processing continues with the next instruction. Similarly, no branch takes place for the BCR instruction when the R2 field =0.

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if the target address is not a halfword address, or if it cannot be accessed, a corresponding program interrupt (with the weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

– It is possible that more than one or no bit position in mask M1 =1; a branch takes place for every condition code value whose corresponding mask bit =1. Accordingly, a branch will take place in every case if mask M1 consists entirely of ones, and never if it consists entirely of zeros.

– The assembler considerably simplifies the writing of BCR and BC instructions: for both instructions, and for all meaningful bit combinations in the mask, it has at its disposal so-called "extended mnemonic operation codes" that create BC and BCR instructions *including* mask. For example, from the mnemonic operation code BE the assembler [1] creates a BC instruction with the mask B'1000' (=8), which can be written following arithmetic compare operations if a branch is to take place with CC =0 (i.e. equivalence). This means that the author of the program does not have to memorize the above table, and the reader of the program will be able to understand the control flow. For further information, see the complete table of "extended mnemonic operation codes" in the appendix.

– Note the following difference between BCR and BC: With BCR the branch address is determined by the *contents* of the second operand, whereas with BC it is determined by the *address* of the second operand.

Example

```
Name        Operation   Operands
            .
Example1    CL          4,5
            BC          7,AGAIN          Branch when CC  0
*                                        using mask = 7(10) = (0111)(2)
            .
Example2    TM          SEMAPHOR,X'80'
            BO          ON               Extended mnemonic operation
*                                        code BO;
*                                        causes branch when CC =3 using
*                                        mask =1(10) =(0001)(2)
            .
Example3    BR          14               Extended mnemonic operation
*                                        code BR;
*                                        causes unconditional branch
*                                        using mask =15(10) =(1111)(2).
*                                        Branch destination: address
*                                        in register 14.
            .
```

Branch on Count

Function

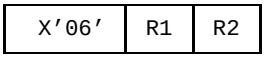
The instructions BCTR and BCT decrement the contents of a specified register by one and branch to a specified address when the resulting register has a value ≠0.
The condition code is left unchanged.

Assembler formats

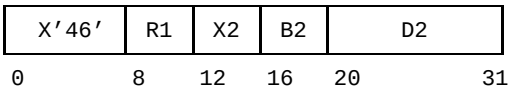
Name	Operation	Operands	Remarks
	BCTR BCT	R1,R2 R1,D2(X2,B2)	

Machine formats

BCTR [RR]



BCT [RX]



Description

The number 1 is subtracted from the binary number in general-purpose register R1: any carry over beyond the higher-order bit position will be ignored. If the result ≠0, the BCTR instruction branches to the address contained in general-purpose register R2, and the BCR instruction to the address D2(X2,B2). This address is 24 or 31 bits long, depending on the addressing mode used. If the result =0, no branch takes place; instead, processing continues with the next instruction. Similarly, no branch takes place for the BCTR instruction when the R2 field =0.

The branch address is determined before general-purpose register R1 is changed.

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if a branch actually does take place, and if the target address is not a halfword address or cannot be accessed, a corresponding program interrupt (with weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

– The contents of general-purpose register R1 can be regarded equally as a signed or an unsigned number, since in both cases subtracting 1 yields the same result.

– Note the following borderline cases: When register R1 has the contents -2^{31}, subtracting 1 yields the contents $+2^{31}-1$; when it has the contents 0, subtracting 1 yields the contents -1. Therefore, a branch will take place in both cases. The only time a branch will not take place is when general-purpose register R1 contains the value +1 prior to the BCTR or BCT instruction (or, with the BCTR instruction, when the R2 field is =0).

– Note the following difference between BCTR and BCT: With BCTR the branch address is determined by the *contents* of the second operand, whereas with BCT it is determined by the *address* of the second operand.

Example

```
Name      Operation   Operands
          .
          LH          5,=H'100'
AGAIN     EQU         *
            .
            .
            .
          BCT         5,AGAIN
          .
```

In the above example, AGAIN is run through exactly 100 times.

Branch and Set Mode

Function

The BSM instruction stores the current addressing mode; it then sets a specified addressing mode and branches in this mode to a specified address.
The condition code is left unchanged.
The BSM instruction is only available in the instruction set of central processing units which have 31-bit addressing mode at their disposal.

Assembler format

Name	Operation	Operands	Remarks
	BSM	R1,R2	

Machine format

BSM [RR]

Description

If the R1 field ≠0, the current addressing mode is stored in bit position 0 of general-purpose register R1; bit positions 1 to 31 remain unchanged. If the R1 field =0, the current addressing mode is not stored.

Then, if the R2 field ≠0, the addressing mode resulting from bit 0 of general-purpose register R2 is set, and a branch is performed in this mode to the address contained in bit positions 1 to 31 of R2. This address is 24 or 31 bits long, depending on the addressing mode used. If the R2 field =0, no branch takes place; instead, processing continues with the next instruction in the current addressing mode.

In both general-purpose registers R1 and R2, a value 0 at bit position 0 indicates 24-bit addressing mode, and a value 1 indicates 31-bit addressing mode.

The branch address is determined before general-purpose register R1 is changed

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if the target address is not a halfword address or cannot be accessed, a corresponding program interrupt (with weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

- The BSM instruction is used primarily for returning from a subprogram that was called with one of the instructions BASR, BAS or BASSM. However, the instruction can also be used in other cases as an unconditional branch.
- Take care not to use the BSM instruction to return from a subprogram that was called with the BAL instruction. Namely, if the calling program is running in 24-bit addressing mode, the wrong addressing mode may govern the calling program following the return (due to the ILC value $(10)_2$ in bit positions 0 and 1 of the instruction continuation address). Subprograms called with BALR or BAL should be exited with the instruction BCR or BC.
- The BSM instruction is only available on central processing units which have 31-bit addressing mode at their disposal. In order to make an Assembler program independent of the central processing unit on which it is running, the macro ##BSM is provided in the BS2000 macro instruction set (V9.0 and successors). This macro "switches on" the BSM instruction. The ##BSM macro generates the BSM instruction on systems with 31-bit addressing mode, and the BR instruction on systems which only have 24-bit addressing mode at their disposal.
  A complete discussion of the addressing modes and the various forms of program linkage can be found in the manual "Introduction to XS Programming (for ASSEMBLER Programmers)" [2].

Example

See example under BASSM for use of BSM.

Branch on Index

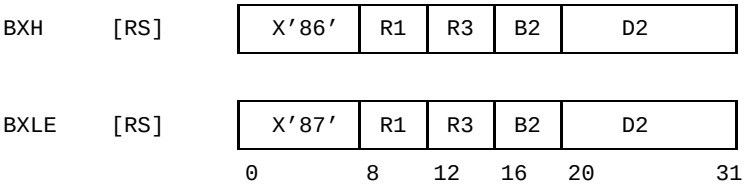
Function

The instructions BXH and BXLE perform signed addition of the contents of a general-purpose register and the contents of another general-purpose register. They then branch to a specified address if the resultant sum is greater than (BXH) or less than or equal to (BXLE) the contents of a third general-purpose register.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	BXH BXLE	R1,R3,D2(B2) R1,R3,D2(B2)	

Machine formats



Description

First, the 32-bit fixed-point number in general-purpose register R1 is added to the 32-bit fixed-point number in general-purpose register R3. The sum is then turned into a signed 32-bit fixed-point number, but any carry over beyond the highest-order binary position will be ignored.

The sum is then compared with the 32-bit fixed-point number in a compare register, with the signs being taken into account. Following the compare operation, the sum is stored in general-purpose register R1. The compare register is general-purpose register R3+1 if R3 is an even number; otherwise it is general-purpose register R3.

Next, a branch takes place to the address D2(B2), provided the sum is greater than (with BXH) or less than or equal to (with BXLE) the contents of the compare register; otherwise, no branch takes place and processing continues with the next instruction.

The compare operation is performed and the branch address D2(B2) is determined before general-purpose register R1 is changed.

Condition code

Stays the same.

Program interrupts

None with the instruction itself.
However, if a branch actually does take place, and if the target address is not a halfword address or cannot be accessed, a corresponding program interrupt (with weight $5C_{16}$ or 48_{16}) will take place at the target address.

Programming notes

- If R3 is not an even number, its contents are used both as an increment value for general-purpose register R1 and as a compare value.
- General-purpose register 0 may be used for R1 and R3.
- If R1=R3, the increment is doubled. If R1 is additionally an odd number, note that the compare operation takes place *before* general-purpose register R1 is changed, i.e. the doubled contents of R1 are compared with the nondoubled contents of R1 and only then used to replace the nondoubled contents of R1.
- The instructions BXH and BXLE are very helpful for programming "for" loops in which a runtime variable increases (or decreases) from an initial value to a final value by a constant increment (or decrement). In these cases, the increment (or decrement) is kept in an even-numbered register R3 and the final value in the neighboring odd-numbered register R3+1. An arbitrary different register R1 is used for the initial value and the runtime variable.
- If the loop problem can be arranged in such a way that the runtime variable runs from an initial value >0 to a final value =0 by a constant decrement, it is even possible to skip one of the general-purpose registers: simply use BXH and enter the decrement (as a negative increment) in an *odd-numbered* general-purpose register; the BXH instruction will then use the same register for decrementation and comparison purposes.

Examples

Example 1

Consider the first case mentioned above in the "Programming Notes", i.e. a loop with a runtime variable which increases from an initial value *a* via the values *a*+*i*, *a*+2*i*, ... , to the final value *z*. This case can be programmed using BXLE as follows:

```
Name    Operation   Operands
        .
        LA          3,a         Any register for the runtime variable
        LA          8,i         Even-numbered register for the increment
        LA          9,z         Neighboring register for the final value
BODY    EQU         *           ┐
         .                      │
         .                      ├ Loop body
         .                      │
        BXLE        3,8,BODY    ┘
        .
```

Note that BODY is also executed for the runtime variable value *z* itself.

Example 2

Consider the second case mentioned above under "Programming Notes", i.e. a runtime variable which decreases from an initial value *a* to a final value 0 via the values *a*-*i*, *a*-2*i*, etc. Using BXH, this might look as follows:

```
Name    Operation   Operands
        .
        LA          3,a         Any register for the runtime variable
        LH          9,=H'-i'    Odd-numbered register for the decrement
*                               and compare values
BODY    EQU         *           ┐
         .                      │
         .                      ├  Loop body
         .                      │
        BXH         3,9,BODY    ┘
        .
```

Compare the operand field of BXH in this example with that of BXLE in the preceding example.
With the final execution of BXH, -*i* is compared with -*i*. Therefore no branch takes place, whereas with the non-last executions of BXH the value 0 is compared with -*i* and hence a return occurs. With both BXH and BXLE, the comparison with the final value takes the sign into account, so that the desired result is obtained by comparing a positive runtime variable with a negative final value. (We do not strongly recommend this "trick", but it is legal.)

Compare

Function

The instructions CR and C perform a signed comparison of two 32-bit fixed-point numbers.
The condition code is set in accordance with the comparison result.

Assembler formats

Name	Operation	Operands	Remarks
	CR C	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats

Description

The CR instruction compares the contents of general-purpose register R2 with the contents of general-purpose register R1; the C instruction compares the word addressed in main memory with D2(X2,B2) with the contents of general-purpose register R1. Both operands are treated as 32-bit signed binary numbers (fixed-point numbers).

The contents of general-purpose register R1 are left unchanged.

Condition code

0~Equal	operand1 = operand2
1~Low	operand1 < operand2
2~High	operand1 > operand2
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	C: Read access of operand 2 illegal.
Addressing error	X'5C'	C: D2(X2,B2) not a word boundary.

Example

Name	Operation	Operands
	.	
	L	5,=F'-1'
	C	5,=F'+1' sets condition 1~Low
	.	

If used instead of instruction C, the CL instruction would set the condition code 2~High.

Compare Halfword

Function

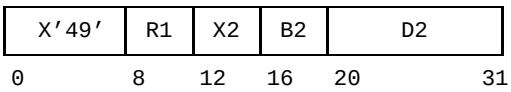
The CH instruction performs signed comparison of a 32-bit fixed-point number with a 16-bit fixed-point number.
The condition code is set in accordance with the comparison result.

Assembler formats

Name	Operation	Operands	Remarks
	CH	R1,D2(X2,B2)	D2(X2,B2): halfword boundary

Machine format

CH [RX]



Description

The contents of general-purpose register R1 are compared with the halfword addressed in main memory with D2(X2,B2). The register operand is treated as a 32-bit fixed-point number, the halfword operand as a 16-bit fixed-point number.

Condition code

0~Equal	operand1 = operand2
1~Low	operand1 < operand2
2~High	operand1 > operand2
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	Read access of operand2 illegal. D2(X2,B2) not a halfword boundary.

Example

```
Name      Operation   Operands
          .
          L           5,=F'-1'
          CH          5,=H'-1'          sets condition code 0~Equal
          .
```

In the example, a binary number consisting of 32 ones is compared with a binary number consisting of 16 ones. The comparison yields "equal" because the second operand is treated as though it were padded to the left 16 ones.

Compare Logical

Function

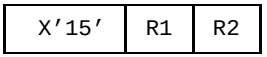
The instructions CLR and CL perform unsigned comparison of two 32-bit binary numbers.
The condition code is set in accordance with the comparison result.

Assembler formats

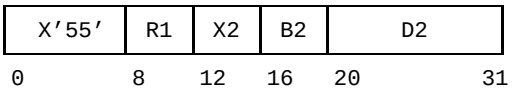
Name	Operation	Operands	Remarks
	CLR CL	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats

CLR [RR]



CL [RX]



Description

The CLR instruction causes an unsigned (logical) comparison between the contents of general-purpose register R2 and the contents of general-purpose register R1; the CL instruction causes an unsigned comparison between the word addressed in main memory with D2(X2,B2) and the contents of general-purpose register R1. Both operands are treated as 32-bit unsigned binary numbers.

The contents of general-purpose register R1 remain unchanged.

Condition code

0~Equal	operand1 = operand2
1~Low	operand1 < operand2
2~High	operand1 > operand2
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	CL: Read access of operand2 illegal.
Addressing error	X'5C'	CL: D2(X2,B2) not a halfword boundary.

Example

Name	Operation	Operands
	.	
	L	5,=F'-1'
	CL	5,=F'+1' sets condition code 2~High
	.	

If used instead of instruction CL, the C instruction would set the condition code 1~Low.

Compare Logical Characters

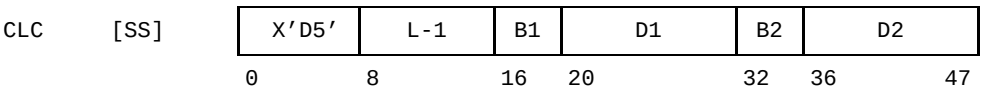
Function

The CLC instruction performs an unsigned (logical) comparison between two character fields.
The condition code is set in accordance with the comparison result.

Assembler formats

Name	Operation	Operands	Remarks
	CLC	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine format



Description

The character field addressed in main memory by D1(B1), with a length of L bytes, is compared logically from left to right with the character field of the same length addressed by D2(B2). The instruction is terminated if the operands are not equal or when they have been completely processed.

Condition code

0~Equal	operand1 = operand2
1~Low	operand1 < operand2
2~High	operand1 > operand2
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand1 or operand2 illegal.

Programming notes

– With the CLC instruction, it is possible to compare operands with a field length of up to 256 bytes. The CLCL instruction has been provided for fields which are more than 256 bytes long.

– The character fields to be compared may overlap in any way you wish. This feature can be used, for example, to check whether a character field contains recurring character subfields (see Example 3).

Examples

```
Name        Operation   Operands
            .
FIELD1      DC          C'AC'
FIELD2      DC          C'AB'
FIELD3      DC          C'*********'
            .
            .
            .
Example1    CLC         FIELD1(1),FIELD2      sets condition code 0~Equal
            .
            .
Example2    CLC         FIELD1,FIELD2         sets condition code 2~High
            .
            .
Example3    CLC         FIELD3+1(L'FIELD3-1),FIELD3
*
*                                             FIELD3 is tested to see if it
*                                             consists of identical chars.
*                                             Sets condition code 0~Equal
            .
```

Compare Logical Long

Function

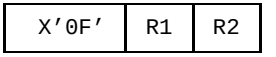
The CLCL instruction performs unsigned (logical) comparison of a main memory area with another main memory area, from left to right. Both areas may be up to 2^{24} bytes long, i.e. 16 MB.
The condition code is set in accordance with the comparison result.

Assembler format

Name	Operation	Operands	Remarks
	CLCL	R1,R2	R1 and R2 even-numbered

Machine format

CLCL [RR]

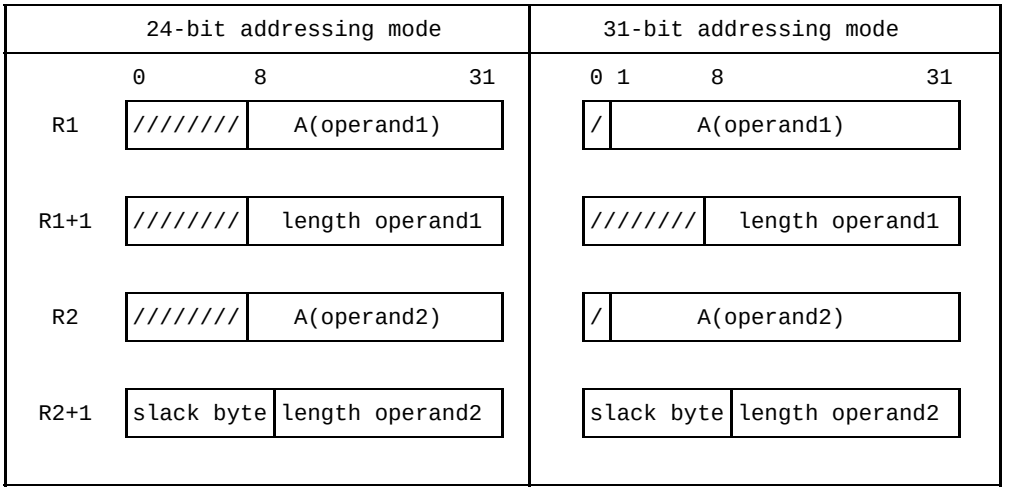


Description

R1 and R2 each determine a pair of registers, consisting of general-purpose registers R1 and R1+1, or R2 and R2+1. R1 and R2 must be even-numbered; otherwise, a program interrupt will occur due to an addressing error.

Operand1 and operand2 are determined respectively by the register pairs R1 and R1+1 or R2 and R2+1. Their start addresses are taken from the first register R1 or R2, their lengths (in bytes) from the second register R1+1 or R2+1. Register R2+1 also contains the coding of the slack byte.

The representation of the address in R1 or R2 depends on the addressing mode used. The following assignments apply:



"/" means: "is ignored".

The comparison takes place logically (unsigned) from left to right. It terminates when either nonidentical is determined or the end of the longer operand is reached. If the operands are of different lengths, the shorter operand is treated as though it were padded to the right with slack bytes to the length of the longer operand. The coding of this slack byte is taken from the highest-order byte of R2+1.

The CLCL instruction can be interrupted on the hardware side. If it is interrupted, the progress of the comparison up to that point is retained in the register pairs R1 and R2 (by storing the incremented addresses and the decremented lengths). Following the interrupt, the comparison is then resumed at the position in main memory where the interrupt occurred.

Once the instruction has been executed, the register pairs R1 and R1+1, or R2 and R2+1, have the following values:

– If the operands are identical, registers R1 and R2 contain the address of the first operand and the second operand respectively, increased by the length fields in R1+1 or R2+1. The length fields in R1+1 and R2+1 receive the value 0.

– If the operands are not identical, and their nonidentity did not occur in the area of the slack bytes, R1 and R2 contain the address of the first nonidentical byte in the first operand and the second operand, respectively. The length fields in R1+1 or R2+1 are reduced by the number of identical bytes.

– If the operands are not identical but their nonidentity was only determined in the area of the slack bytes, then the first register of the longer operand is increased by the number of "identical" bytes, and the first register of the shorter operand by the length of that operand; in each of the two second registers, the length field of the longer operand is reduced accordingly by the number of "identical" bytes, and that of the shorter operand is set to 0.

In all three cases, once the instruction has been executed the highest-order bytes of R1+1 and R2+1 remain unchanged and the uppermost 8 bits or the uppermost bit (depending on the address mode used) of R1 and R2 is set to 0.

Condition code

0~Equal	operand1 = operand2 or L'operand1=0 and L'operand2=0
1~Low	operand1 < operand2
2~High	operand1 > operand2
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand1 and operand2 illegal.
Addressing error	X'5C'	R1 or R2 not even-numbered.

Programming notes

– If both operands have the length 0, they are considered identical.

– The two operands may overlap in any way you wish.

– The application program should ensure on its own that all of the address spaces of both operands lie exclusively in its own address space. This is because the CLCL instruction does not check whether the address spaces are within the permissible limits at the start of execution but only during execution. If an illegal subspace is detected, the instruction does not necessarily abort; instead, it may resume execution following the subspace. In any case, however, the results both in main memory and in the condition code are unusable.

– The user should not activate the CLCL instruction with the EX instruction if this EX uses the same registers as the CLCL.

Example

The examples below compare a maximum of 20000 bytes of the main memory areas C1 and C2. Since C2 has a length of 15000 bytes, the last 5000 bytes of C1 are compared with the slack byte, provided the first 15000 bytes are identical.

```
Name    Operation   Operands
        .
        LM          4,5,=A(C1,20000)    Registers 4 and 5: operand1
        LM          10,11,=A(C2,15000)  Registers 10 and 11: operand2
        ICM         11,B'1000',=C' '    Slack byte to byte 0 from reg 11
        CLCL        4,10
        .
```

After execution of the CLCL instruction, the following values may occur:

- If identity occurs, registers 4 and 10 contain the values A(C1+20000) and A(C2+15000), and the length fields of registers 5 and 11 contain the value 0.
- If nonidentity occurs within the first 15000 bytes, registers 4 and 10 point to the first nonidentical byte in C1 or C2, and the length fields of registers 5 and 11 are reduced by the number of identical bytes.
- If nonidentity occurs in the area of the last 5000 bytes, register 4 contains the address of the first nonidentical byte, and the length field of register 5 is reduced by the number of identical bytes. Register 10, however, contains the value A(C2+15000) and the length field of register 11 contains the value 0.

In all cases, the uppermost byte of registers 5 and 11 remains unchanged, i.e. in the above example, =X'00' or =C'_'.

Compare Logical Immediate

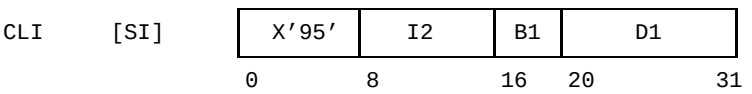
Function

The CLI instruction compares one byte of main memory with the direct operand I2. The condition code is set in accordance with the comparison result.

Assembler format

Name	Operation	Operands	Remarks
	CLI	D1(B1),I2	X'00' ≤ I2 ≤ X'FF'

Machine format



Description

The byte addressed in main memory by D1(B1) is compared logically (unsigned) with the direct operand I2.

Condition code

0~Equal operand1 = I2
1~Low operand1 < I2
2~High operand1 > I2
3 Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand1 illegal.

Example

```
Name        Operation   Operands
            .
FIELD1      DC          C'AC'
            .
            .
            .
            CLI         FIELD1,C'A'          sets condition code 0~Equal
            .
```

Compare Logical under Mask

Function

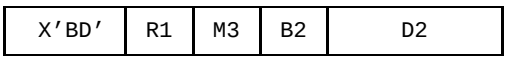
The CLM instruction performs unsigned (logical) comparison of selected bytes in a general-purpose register with a character field in main memory.
The condition code is set in accordance with the comparison result.

Assembler format

Name	Operation	Operands	Remarks
	CLM	R1,M3,D2(B2)	B'0000' ≤ M3 ≤ B'1111'

Machine format

CLM [RS]



Description

The 4 bits of the "mask" M3 correspond one-to-one to the 4 bytes of general-purpose register R1 (from left to right, both in the mask and in the register). Those bytes in R1 which correspond to ones in the mask are treated as a contiguous field; this field is compared logically (unsigned) with the character field in main memory by D2(B2).

Condition code

0~Zero	The selected bytes of R1 are identical to the character field, or the mask is $=0_{16}$.
1~Minus	The selected bytes of R1 are less than the character field.
2~Plus	The selected byes of R1 are greater than the character field.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access to operand2 illegal, even if M3 = 0_{16}.

Programming notes

– The length (in bytes) of the main memory field is equal to the number of ones in the mask.

– When a mask consisting entirely of ones is used (B'1111'), the CLM instruction has the same effect as the CL instruction, except that the main memory field need not be aligned on a word boundary.

Example

Name	Operation	Operands
	. CLM .	3,B'1001',=C'LR'

The instruction in the example performs a logical comparison of the highest-order and the lowest-order bytes of general-purpose register 3 with the character string LR.

Compare and Swap

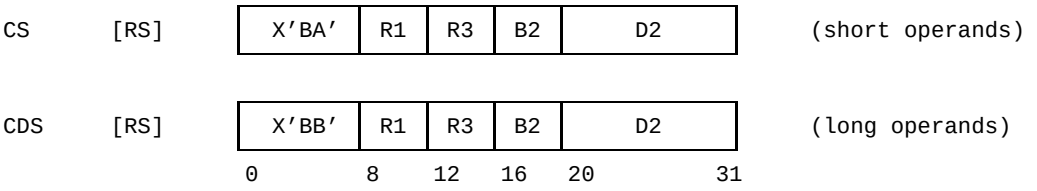
Function

The instructions CS and CDS store the contents of a general-purpose register in a main memory area, provided the contents of this main memory area are identical to the contents of another general-purpose register. The instructions preserve the integrity of their data during execution of the instruction.
The condition code is set.

Assembler formats

Name	Operation	Operands	Remarks
*	CS CDS	R1,R3,D2(B2) R1,R3,D2(B2)	D2(B2): word boundary R1, R3 even-numbered and D2(B2): doubleword boundary

Machine formats



Description

The first operand (R1) is compared logically (unsigned) with the second operand (D2(B2)). If they are identical, the third operand (R3) replaces the second operand and the condition code is set to 0~Equal; if they are not identical, the second operand replaces the first operand and the condition code is set to 1~Not Equal. The second operand is write-locked until the instruction execution has finished.

With CS, all three operands are 32 bits long; with CDS, they are 64 bits long.

Instr.	Operand1	Operand2	Operand3
CS CDS *	Register R1 Register pair R1,R1+1	Word at D2(B2) Double word at D2(B2)	Register R3 Register pair R3, R3+1

Condition code

0~Equal	Operand2 was identical to operand1; operand2 is replaced by operand3.
1~Not Equal	Operand2 was not identical to operand1; operand1 is replaced by Operand2.
2	Not used.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	Read/write access of operand2 illegal. CS: D2(B2) not a word boundary. CDS: D2(B2) not a double word boundary or R1 or R3 not even-numbered.

Programming notes

– The instructions CS and CDS (as well as TS) are the only instructions in the instruction set that read- and write-lock the main memory data which they address while they are being executed. During this period this data cannot be read or written by any other programs, either on the same central processing unit or on different ones. These instructions also prevent access to subsequent instructions and/or their operands while they are being executed. The instructions thereby ensure that the data status at the time of their (initial) read operation is identical to the data status at the time of their (final) write operation. For this purpose, a so-called "serialization" takes place in the hardware before and after the CS or CDS instruction, during which all outstanding memory access operations are processed. This mechanism predestines CS and CDS instructions for synchronization problems in multiprocessor applications.

  In applications of this sort, each of the concurrently running programs must take into account that while they are processing and modifying a common memory area another program may be doing the same thing, thereby causing the processing results to cancel each other out.

Consider the simple case where two concurrently running programs A and B increment a common word in main memory, called *counter*, by one. If both programs accidentally perform this incrementation at the same time, but program B is running one instruction behind program A, they may produce the following effect:

Time	Counter	Program A	Program B
t_0	assume 100	Reads counter (100)	
t_1	100	Increments counter (101)	Reads counter (100)
t_2	101	Stores counter (101)	Increments counter (101)
t_3	101		Stores counter (101)

Although both programs have performed incrementation, *counter* only contains the value 101 at the end since program A had not finished storing it when program B began.

On the pages that follow we will show you an instruction sequence that forms a safe method against this effect. The idea is to use the CS (or CDS) instruction for storing the modified value instead of the ST instruction. Namely, the CS instruction determines, prior to time t_3 of program B, that the current contents of *counter* are no longer identical to its contents at time t_1, since program A has modified these contents in the meantime, namely at time t_2. In this case, program B does not perform storage, but repeats incrementation, correctly proceeding from the value 101.

– Even if a shareable main memory area is longer than 4 (or 8) bytes, the instructions CS and CDS may still be used. This is usually done by setting up a full word or doubleword which stands for this storage area and in which the possible "statuses" of the storage are contained. CS or CDS then only manages this status word rather than the actual storage area.

– The instructions CS and CDS should *only* be used for coordinating programs (in the same or in different central processing units), and not to replace an instruction sequence of CL and ST instructions: namely CS and CDS are time-consuming and block instruction execution in other central processing units.

Example

One safe method of updating a shared main memory word (SHAREWD) by means of CS is the instruction sequence shown below:

```
Name        Operation   Operands
            .
UPDATE      L           R1,SHAREWD
AGAIN       LR          R3,R1
            .
    < compute update value in R3, R1 must remain unchanged >
            .
            CS          R1,R3,SHAREWD
            BNE         AGAIN
            .
```

The instruction sequence begins by loading the initial value of SHAREWD into the general-purpose register R1; here it must remain unchanged until the CS instruction is issued. The update value of SHAREWD is produced in another register (R3). The final storage operation takes place via the CS instruction, which first checks whether the current value of SHAREWD is (still) identical to its initial value (in R1). Only if this is the case does CS actually perform the storage; it also sets the condition code to 0~Equal, thereby exiting the instruction sequence. If, however, at the time CS is executed the value of SHAREWD is not (no longer) identical to the contents of R1, no storage takes place; instead, the contents of the now modified main memory word are loaded into register R1, the condition code is set to 1~Not Equal and the program section is repeated.

Convert to Binary

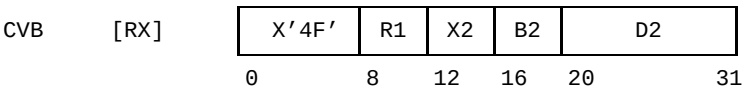
Function

The CVB instruction converts a packed decimal number into a 32-bit fixed-point number.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	CVB	R1,D2(X2,B2)	D2(X2,B2): doubleword boundary

Machine format



Description

D2(X2,B2) must address a packed decimal number which is exactly 8 bytes in length and is located in a doubleword in main memory. This number is converted into a 32-bit signed fixed-point number and stored in general-purpose register R1.

The decimal number to be converted must lie in the range -2^{31} ... $+2^{31}-1$, i.e. it must be at least -2147483648 and may be as large as +2147483647. If this condition is not satisfied, conversion is performed anyway, but general-purpose register R1 will only contain the lowest-order 32 bits of the fixed-point number following instruction execution, and a program interrupt will also occur due to a division error.

The decimal number to be converted is checked for a correct, packed format. In case of error, a program interrupt occurs due to a data error.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address. trans. error	X'48'	Read access of operand2 illegal.
Addressing error	X'5C'	D2(X2,B2) not a doubleword boundary.
Division error	X'68'	The decimal number to be converted is >+2147483647 or <-2147483648.
Data error	X'60'	The number to be converted is not a correctly packed, 8-byte decimal number.

Programming notes

– If the decimal number is negative, the fixed-point number is represented by its twos complement.

Examples

The examples below produce the following results in general-purpose register 3:

FIELD (on double-word boundary)	Sample instruction	Register 3 after
PL8'255'	CVB 3,FIELD	F'255' = X'000000FF'
PL8'-255'	CVB 3,FIELD	F'-255' = X'FFFFFF01'
PL8'+2147483647'	CVB 3,FIELD	F'2147483647' = X'7FFFFFFF'
PL8'-2147483649'	CVB 3,FIELD	F'2147483647' = X'7FFFFFFF'

In all examples it is assumed that the main memory operand FIELD is aligned on a doubleword boundary.

In the last example, a program interrupt occurs due to a division error, since the decimal number to be converted is too small (by one). Register 1 contains the lowest-order 32 bits of the correct fixed-point number. In this case these bits are identical to the fixed-point number from the largest possible decimal number.

Convert to Decimal

Function

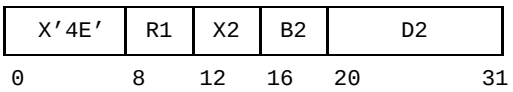
The CVD instruction converts a 32-bit fixed-point number to a packed 8-byte decimal number.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	CVD	R1,D2(X2,B2)	D2(X2,B2): doubleword boundary

Machine format

CVD [RX]



Description

The fixed-point number (with sign) in general-purpose register R1 is converted to a 15-digit packed decimal number which is exactly 8 bytes long and is located in main memory (at a doubleword addressed by D2(X2,B2). General-purpose register R1 is left unchanged.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	Write access to operand2 illegal. D2(X2,B2) not a doubleword boundary.

Programming notes

– Any 32-bit signed fixed-point number can be converted.

– If the decimal number is positive, its sign is set to = C_{16}; otherwise it is = D_{16}.

Examples

The examples below produce the following results in FIELD (which must be aligned on a doubleword boundary):

Register 3	Sample instruction	FIELD (on doubleword boundary)
F’255’	CVD 3,FIELD	PL8’255’
F’-255’	CVD 3,FIELD	PL8’-255’
X’FFFFFFFF’	CVD 3,FIELD	PL8’-1’
X’80000000’	CVD 3,FIELD	PL8’-2147483648’

Divide

Function

The instructions DR and D perform signed division of a 64-bit fixed-point number by a 32-bit fixed-point number. The remainder and the quotient replace the dividend. The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
*	DR D	R1,R2 R1,D2(X2,B2)	R1 even-numbered R1 even-numbered and D2(X2,B2): word boundary

Machine format

Description

The R1 field of instruction DR and D defines a pair of general-purpose registers consisting of the registers R1 and R1+1. R1 must be even-numbered, otherwise a program interrupt will occur due to an addressing error.

The dividend is taken from general-purpose registers R1 and R1+1. With the DR instruction, the divisor is taken from general-purpose register R2; with D it is taken from the main memory word addressed by D2(X2,B2). The remainder is stored in the (even-numbered) register R1, the quotient in the (odd-numbered) register R1+1; they overwrite the dividend.

The dividend is treated as a 64-bit fixed-point number with a sign at bit position 0 of R1; the divisor, remainder and quotient are treated as 32-bit signed fixed-point numbers.

The sign of the quotient is computed according to the usual algebraic rules; the remainder always has the same sign as the dividend.

If the quotient is too large to be included in register R1 or the divisor is =0, a program interrupt occurs due to a division error (even if the dividend is =0).

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	D : Read access of operand2 illegal.
Addressing error	X'5C'	DR, D : R1 not even-numbered.
		D : D2(X2,B2) not a word boundary.
Division error	X'68'	Divisor =0 or quotient too large.

Programming notes

– If R1=R2, this always causes a program interrupt due to a division error.

– The maximum values for the dividend are $+^{26}2+^{23}1-1$ and $-^{26}2+1$, not $+2^{63}-1$ and and -26^3 (see examples).

– Note that once the instruction has been executed, the remainder is stored *before* the quotient (R1: remainder, R1+1: quotient).

Examples

The following values of dividend and divisor produce the values shown below for quotient and remainder:

Dividend	Divisor	Remainder	Quotient
+500	+17	+7	+29
+500	-17	+7	-29
-500	+17	-7	-29
-500	-17	-7	+29
Limit values:			
$+2^{62}+2^{31}-1$	$2^{31}-1$	$+2^{31}-2$	$+2^{31}-1$
$+2^{62}+2^{31}-1$	-2^{31}	-2^{31}	$+2^{31}-1$
$-2^{62}+2$	$+2^{31}-1$	$-2^{31}+2$	-2^{31}
$-2^{62}+1$	-2^{31}	$-2^{31}+1$	$+2^{31}-1$

Execute

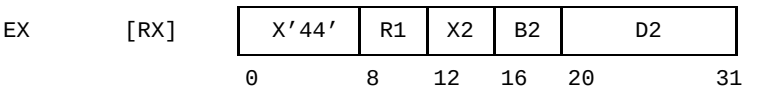
Function

The EX instruction executes another instruction. This instruction may be modified beforehand.
The condition code is changed only if it is changed by the executed instruction.

Assembler format

Name	Operation	Operands	Remarks
	EX	R1,D2(X2,B2)	D2(X2,B2): halfword boundary

Machine format



Description

The EX instruction executes the instruction addressed by D2(X2,B2) (the "target instruction"). Before this takes place, bit positions 8 to 15 are linked with logical OR to bit positions 24 to 31 of general-purpose register R1. The OR link does not change the instruction itself, nor does it change register R1; instead, it only influences the interpretation of the target instruction.

If R1=0, the target instruction is executed without a preceding OR link.

The target instruction can be 2, 4 or 6 bytes long. It is executed as though it were located at the memory position of instruction EX, and as though it too were 4 bytes long. If, for example, the target instruction is BALR, the continuation address of the EX instruction (and not that of the BALR instruction) is stored as the "instruction continuation address" and the value $(10)_2$ (rather than $(01)_2$) is stored as ILC.

The target instruction of the EX instruction must not be another EX; otherwise, a program interrupt will occur due to an addressing error. The address defined by D2(X2,B2) must be even-numbered; otherwise, a program interrupt due to an addressing error will likewise occur. If the target instruction is not a correct instruction of the instruction set, the results of EX will be unpredictable.

Condition code

The condition code is changed as modified by the target instruction.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	Read access of target instruction illegal. Target instruction itself an EX, or D2(X2,B2) not a halfword boundary.

Programming notes

- By ORing the second byte of a target instruction, the EC instruction makes it possible to indirectly determine the length field, index field, mask field, register field or second operation code byte of this target instruction.
- The EX instruction is especially important for programming so-called "read only" (or "reentrant") programs since it does not change the target instruction (see example).
- Caution is advised when the target instruction is interruptable (e.g. the instruction CLCL or MVCL). In this case, not one of the registers X2 and B2 in the EX instruction should be used, not even in the target instruction, since their integrity can no longer be ensured following an EX. Nor should the EX instruction itself be contained in the receive field of the target instruction in the case of MVCL.
- The EX instruction is very time-consuming.

Examples

Example 1

The following instructions turn a variable-length number into a fixed-length number, at the same time converting it into packed format:

```
Name        Operation     Operands
            LH            5,SLENGTH        SLENGTH: Length of SFIELD number
            BCTR          5,0              minus 1
            EX            5,PACKINST
              .
              .
              .
PACKINST    PACK          DFIELD,SFIELD(0)  L1=L'DFIELD, L2=0
              .
```

The length of the SFIELD number is taken from the halfword SLENGTH; it is then ORed by the EX instruction to the L2 field of the PACK instruction. For this reason the L2 field must be $=0_{16}$. Note that the length used in the PACK reason the L2 field must be $=0_{16}$.

Note that the length used in the PACK instruction must be reduced by 1 from the true length, as is done here by the BCTR instruction. The assembler itself [1] computes the length of the DFIELD number from the data declaration of DFIELD and reduces it by 1.

Example 2

The following two instruction sequences AAAA and BBBB have the same effect, namely, they move a variable number of bytes from SFIELD to DFIELD:

```
Name        Operation   Operands
            .
AAAA        LH          5,SLENGTH         SLENGTH: number bytes to be moved
            BCTR        5,0               minus 1
            EX          5,MOVEINST
              .
              .
MOVEINST    MVC         DFIELD(0),SFIELD
```

```
            .
BBBB        LH          5,SLENGTH         SLENGTH: number bytes to be moved
            BCTR        5,0
            STC         5,MOVEINST+1      Enters length reduced by 1
MOVEINST    MVC         DFIELD(0),SFIELD  in L field of an MVC
              .
```

The difference between the two is that the instruction sequence AAAA remains "read only" while the instruction sequence BBBB does not. The EX instruction executes the MVC instruction with ORed byte 1, but does not change the instruction itself.
EX is (virtually) indispensable for problems in which the program text must remain constant, but in which dynamic modifications in the parameters of individual instructions are required.

Insert Character

Function

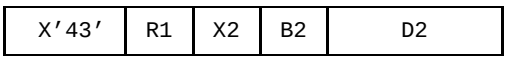
The IC instruction moves a byte from memory to the lowest-order byte of a general-purpose register.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	IC	R1,D2(X2,B2)	

Machine format

IC [RX]



Description

The byte addressed in main memory by D2(X2,B2) is moved to byte 3 (i.e. bit positions 24 to 31) of general-purpose register R1. Bit positions 0 to 23 of R1 remain unchanged.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand2 illegal.

Example

```
Name    Operation   Operands
        .
        L           5,=XL4'ANNA'
        IC          5,='E'              'ANNE' in general-purpose reg. 5
*                                       CC remains unchanged
        .
```

Insert Characters under Mask

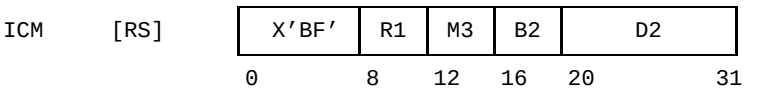
Function

The ICM instruction moves a character field in main memory to selected bytes of a general-purpose register.
The condition code is changed in accordance with the value of the moved field.

Assembler format

Name	Operation	Operands	Remarks
	ICM	R1,M3,D2(B2)	B'0000' ≤ M3 ≤ B'1111'

Machine format



Description

The 4 bits of the "mask" M3 (direct operand) correspond one-to-one with the 4 bytes of general-purpose register R1 (from left to right in both the mask and the register). Those bytes in R1 with corresponding ones in the mask are replaced by consecutive bytes in the main memory field addressed by D2(B2). The bytes of the general-purpose register with corresponding zeros in the mask remain unchanged.

If the mask is $=0_{16}$ or if all the bytes used are $=00_{16}$, the condition code is set to 0~Zero. In all other cases, the highest-order bit of the first byte used determines the condition code. If this bit is 1, the condition code is set to 1~Minus; otherwise, it is set to 2~Plus.

Condition code

0~Zero	All bytes used are $=00_{16}$ or the mask is $=0_{16}$.
1~Minus	The highest-order bit of the first byte used is =1.
2~Plus	The highest-order bit of the first byte used is =0, but at least one further bit is =1.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X’48’	Read access of operand2 illegal, even if M3 = 0_{16}.

Programming notes

– The length in bytes of the main memory field is identical to the number of ones in the mask.

– When a mask consisting entirely of ones is used (B’1111’) the following differences to the L instruction apply:

  – The main memory field does not have to be aligned on a word boundary.
  – The condition code is set.
  – The L instruction is in RX format, the ICM instruction in RS format.
  – The L instruction is quicker.

Examples

The examples below produce the following results:

```
Name        Operation   Operands
                .
Example1    ICM         5,B’1111’,FBLENGTH
                .
                .
FBLENGTH    DC          C’ ’
            DC          FL3’20000’
                .
                .
Example2    ICM         5,B’0001’,=X’00’
                .
```

In example 1, 4 bytes are entered in general-purpose register 5: namely, a blank in byte 0 and the number 20000, binary, in bytes 1 to 3. Since the blank is coded as X’40’, i.e. the first bit entered is 0, the condition is set to 2~Plus.

In example 2, the lowest-order byte in general-purpose register 5 is replaced by a byte from main memory. Unlike the IC instruction, however, in this case the condition code is set, namely, 0~Zero.

Insert Program Mask

Function

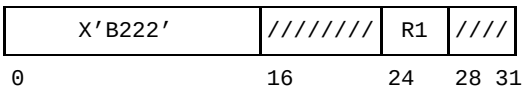
The IPM instruction moves the current values of the condition code and program mask to a general-purpose register.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	IPM	R1	

Machine format

IPM [RRE]



Description

The current value of the condition code (0_{10}, 1_{10}, 2_{10} or 3_{10}) is moved in binary form to bit positions 2 and 3, and the current value of the (4-bit) program mask is moved to bit positions 4 to 7 of general-purpose register R1. Bit positions 0 and 1 of register R1 are set to 0; bit positions 8 to 31 of register R1 remain unchanged.

Bit positions 16 to 23 and 28 to 31 of the instruction are ignored.

Condition code

Stays the same.

Program interrupts

None.

Programming notes

– The IPM instruction "compensates" for the fact that, in 31-bit addressing mode, it is impossible to read the condition code and the program mask with the BALR or BAL instruction. This continues to be possible in 24-bit addressing mode, but here too the use of the IPM instruction is the better solution.

– The IPM instruction does *not* supply the Instruction Length Code (ILC) which is provided by the BALR and BAL instructions (though only in 24-bit addressing mode). When using 31-bit addressing mode one must make do without the ILC (assuming it is ever needed at all).

– The bits in the program mask have the following meaning:

Bit in program mask	Bit position in R1	Meaning
0	4	Fixed-point overflow
1	5	Decimal overflow
2	6	Exponent underflow
3	7	Significance (mantissa = 0)

BS2000 presets all 4 bits of the program mask to 1, so that a program interrupt will take place when the corresponding event occurs. The SPM instruction, however, makes it possible for an application program to change this presetting.

Example

```
Name    Operation   Operands
        .
        ICM         15,B'1000',=X'3C'
        SPM         15                  CC: 3~Overflow
        SLR         11,11               CC: 2~Plus
        IPM         11
        .
```

The instruction sets the condition code to 3 and the program mask to C_{16}. (This suppresses subsequent program interrupts due to exponent underflow and significance, but permits those due to fixed-point and decimal overflow). The instruction SLR 11,11 leaves the program mask unchanged, but sets the condition code to 2. This value is read by the IPM instruction, so that in the end the highest-order byte of register 11 contains the value X'2C' (not X'3C').

Load

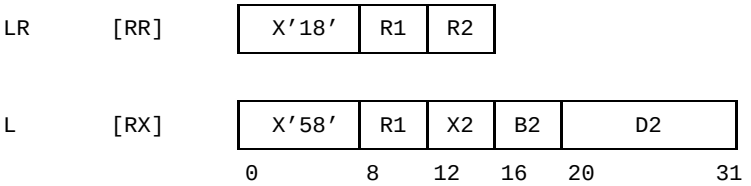
Function

The instructions LR and L move a 32-bit binary number from a general-purpose register or from a word in main memory to a general-purpose register.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	LR L	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats



Description

The word in main memory (L) addressed by D2(X2,B2), or the contents of general-purpose register R2 (LR), are moved to general-purpose register R1.

Instr.	Operand1	Operand2
LR L	Register R1 Register R1	Register R2 Word addressed by D2(X2,B2)

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	L: Read access of operand2 illegal.
Addressing error	X'5C'	L: D2(X2,B2) not a word boundary.

Load Address

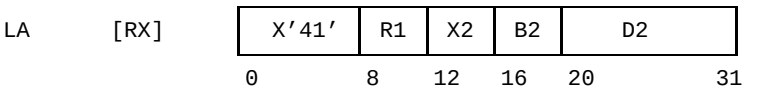
Function

The LA instruction loads an address into a general-purpose register.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	LA	R1,D2(X2,B2)	

Machine format



Description

The address D2(X2,B2) is loaded into general-purpose register R1. The address is computed logically as the sum of the addresses in general-purpose registers X2 and B2 and the binary value of the 12-bit D2 field; any signs and any carry over beyond the highest-order binary position are ignored. If X2=0, the contents of register X2 are *not* added; if B2=0, the contents of register B2 are *not* added.

In 24-bit addressing mode, only the lowest-order 24 bits of the contents of general-purpose registers B2 and X2 are used to form the sum; the sum is entered in bit positions 8 to 31 of general-purpose register R1, and bit positions 0 to 7 of R1 are set to 0.
In 31-bit addressing mode, only the lowest-order 31 bits of B2 and X2 are used to form the sum; the sum is entered in bit positions 1 to 31 of general-purpose register R1, and bit 0 is set to 0.

No memory access takes place to the resulting address.

Condition code

Stays the same.

Program interrupts

None.

Programming notes

– The LA instruction is often critical for importing programs from 24-bit addressing mode to 31-bit addressing mode. In older programs, namely, the highest-order 8 bits to the left of a 24-bit address are not infrequently used up with additional information (e.g. condition codes), and the LA instruction is then employed for the purpose of setting the highest-order 8 bits to 0. Before moving programs from a 24-bit environment to a 31-bit environment, we particularly recommend tracing all the addresses proceeding from LA instructions.

– The LA instruction can be used to increment a general-purpose register by a constant value. This is done by entering this constant in the instruction as a D2 value and setting R1=B2 and X2=0, i.e. by writing, for example, LA 5,6(5) in order to increment register 5 by 6. However, note that the result of the LA instruction is not a fixed-point number but rather an address that has a different length in 24-bit addressing mode than in 31-bit addressing mode. This difference is immaterial only as the result is less than 16 MB.

Example

The instructions below illustrate the "dangers" of the LA instruction:

```
Name    Operation    Operands
                .
A       CSECT
A       AMODE        31
                .
                .
        LA           5,A
        L            15,=V(B)
        BASSM        14,15
*               .
B       CSECT
B       AMODE        24
                .
        LA           5,1(5)
        BSM          0,14
                .
```

In program section A a 31-bit address is created in register 5, and in program section B this address is incremented by 1 using an LA instruction. Since B is running in 24-bit addressing mode, the address created and returned to A is only 24 bits long. The problem here is that A will run correctly in the address space when less than 16 MB, but incorrectly when greater than 16 MB.

Load Complement

Function

The LCR instruction moves the twos complement of a 32-bit fixed-point number from a general-purpose register to a general-purpose register.
The condition code is set in accordance with the resulting fixed-point number.

Assembler format

Name	Operation	Operands	Remarks
	LCR	R1,R2	

Machine format

LCR [RR]

Description

The twos complement of the fixed-point number in general-purpose register R2 is moved to general-purpose register R1.

Fixed-point overflow occurs when the least negative number (-2^{31}) is to be complemented; the result in R1 is then once again the least negative number, and the condition code is set to 3~Overflow; moreover, a program interrupt takes place if the bit for fixed-point overflow in the program mask is 1 (default value in BS2000).

Condition code

0~Zero result = 0
1~Minus result < 0
2~Plus result > 0
3~Overflow fixed-point overflow

Program interrupts

Type	Weight	Causes
Fixed-point overflow	X'78'	R2 contents $=-2^{31}$

Programming notes

– R1 may be equal to R2.

– When the contents of R2 =0, the contents of R1 (and the condition code) are set to =0.

Examples

```
Name        Operation   Operands
            .
Example1    L           0,=F'-1'              Register 0: -1
            LCR         0,0                        now: +1, CC: 2
            LCR         0,0                        now: -1, CC: 1
            .
            .
Example2    L           5,=F'-2147483648'     Register 5 : -2^31
            LCR         6,5                   Register 6 : -2^31
            .
```

Load Halfword

Function

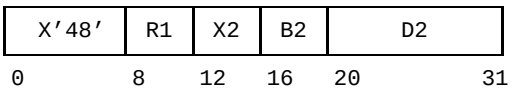
The LH instruction moves a halfword from main memory to bytes 2 and 3 of a general-purpose register and fills bytes 0 and 1 with the sign bit of the halfword.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	LH	R1,D2(X2,B2)	D2(X2,B2): halfword boundary

Machine format

LH [RX]



Description

The halfword addressed by D2(X2,B2) is moved to bit positions 16 to 31 of general-purpose register R1. Bit positions 0 to 15 are set to the value of the highest-order bit of the halfword.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	Read access of operand2 illegal. D2(X2,B2) not a halfword boundary.

Example

```
Name        Operation   Operands
            .
            LH          0,=H'-1'
            CLM         0,B'1100',=X'FFFF'    yields CC 0~Equal
            .
```

Load Multiple

Function

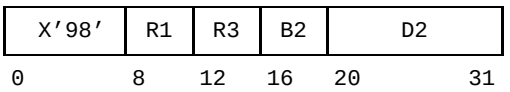
The LM instruction loads up to 16 consecutive words from main memory into consecutive general-purpose registers.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	LM	R1,R3,D2(B2)	D2(B2): word boundary

Machine format

LM [RS]



Description

The consecutive general-purpose registers, beginning with R1 and ending with R3, are loaded with consecutive words, of which the first is addressed with D2(B2).

If R3 is less than R1, loading takes place in ascending order from R1 to general-purpose register 15, and from general-purpose register 0 up to and including R3. If R1=R3, only one register (R1) is loaded.

Instr.	Operand1	Operand2
LM	Contents of registers R1 to R3	Word sequence addressed by D2(B2) No of words =R3-R1+1 if R3≥R1 =R3-R1+17 if R3<R1

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X’48’ X’5C’	Read access of operand2 illegal. D2(B2) not a word boundary.

Example

Name	Operation	Operands
	.	
	LM	14,1,=A(ONE,TWO,THREE,FOUR)
*		General-purpose registers 14, 15, 0 and 1
*		are loaded with 4 consecutive words
*		(in this case addresses).
	.	

Load Negative

Function

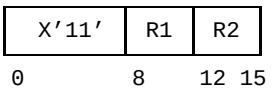
The LNR instruction moves the negative value of a 32-bit fixed-point number from a general-purpose register to a general-purpose register.
The condition code is set in accordance with the moved fixed-point number.

Assembler format

Name	Operation	Operands	Remarks
	LNR	R1,R2	

Machine format

LNR [RR]



Description

If the fixed-point number in general-purpose register R2 is positive, i.e. its bit position 0 =0, then its twos complement is moved to general-purpose register R1; otherwise, it is moved in its original form.

If the fixed-point number to be moved is =0, the moved number is also set to =0.

Condition code

0~Zero	result = 0 (R2 is likewise = 0)
1~Minus	result < 0
2	Not used.
3	Not used.

Program interrupts

None.

Programming notes

R1 may be equal to R2.

Examples

```
Name        Operation   Operands
            .
Example1    L           0,=F'1'          Register 0 before:  +1
            LNR         0,0              Register 0 after: -1
*                                        CC: 1~Minus
            .
            .
Example2    SLR         5,5              Register 5 : 0
            LNR         6,5              Register 6 : 0
*                                        CC: 0~Zero
            .
```

Load Positive

Function

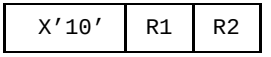
The LPR instruction moves the amount of a 32-bit fixed-point number from a general-purpose register to a general-purpose register.
The condition code is set in accordance with the value of the moved fixed-point number.

Assembler format

Name	Operation	Operands	Remarks
	LPR	R1,R2	

Machine format

LPR [RR]



Description

If the fixed-point number in general-purpose register R2 is negative, i.e. its bit position 0 has the value =1, then its twos complement is moved to general-purpose register R1; otherwise, it is moved in its original form.

Fixed-point overflow occurs when the least negative number (-2^{31}) is to be complemented; the result in R1 is then once again the least negative number, and the condition code is set to 3~Overflow; moreover, a program interrupt takes place if the bit for fixed-point overflow in the program mask is 1 (default value in BS2000).

Condition code

0~Zero	result = 0
1	Not used.
2~Plus	result > 0
3~Overflow	fixed-point overflow

Program interrupts

Type	Weight	Causes
Fixed-point overflow	X'78'	R2 contents $=-2^{31}$

Programming notes

R1 may be equal to R2

Examples

```
Name        Operation   Operands            Remarks

            .
Example1    L           0,=F'-1'            Register 0 before:  -1
            LPR         0,0                 Register 0 after: +1
*                                           CC: 2~Plus
            .
            .
Example2    L           5,=F'-2147483648'   Register 5 : -2**31
            LPR         6,5                 Register 6 : -2**31
*                                           CC: 3~Overflow and
*                                           possibly program interrupt
            .
```

Load and Test

Function

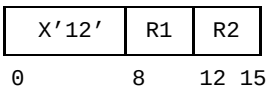
The LTR instruction moves a 32-bit fixed-point number from a general-purpose register to a general-purpose register.
The condition code is set in accordance with the fixed-point number.

Assembler format

Name	Operation	Operands	Remarks
	LTR	R1,R2	

Machine format

LTR [RR]



Description

The fixed-point number in general-purpose register R2 is moved in its original form to general-purpose register R1, and its value is tested.
Fixed-point overflow cannot occur.

Condition code

0~Zero result = 0
1~Minus result < 0
2~Plus result > 0
3~Overflow Not used.

Program interrupts

None.

Programming notes

- R1 may be equal to R2.
- The LTR instruction does the same thing as the LR instruction except that it also sets the condition code.

Multiply

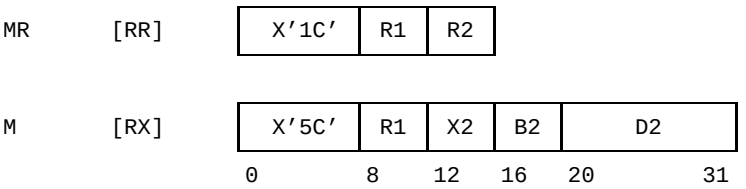
Function

The instruction MR and R perform signed multiplication of two 32-bit fixed-point numbers and create a 64-bit product.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	MR M	R1,R2 R1,D2(X2,B2)	R1 even-numbered R1 even-numbered and D2(X2,B2): word boundary

Machine formats



Description

The R1 field of the instructions MR and M determines a pair of general-purpose registers R1 and R1+1. R1 must be even-numbered, otherwise a program interrupt will occur due to an addressing error.

Der multiplicand is taken from the odd-numbered general-purpose register R1+1; the contents of the even-numbered register R1 are ignored. With the MR instruction, the multiplier is in general-purpose register R2; with M, it is in the main memory word addressed by D2(X2,B2). The product is stored in registers R1 and R1+1.

The multiplicand and multiplier are treated as 32-bit signed fixed-point numbers. The resultant product is a 64-bit fixed-point number with the sign at bit position 0 of general-purpose register R1.

The sign of the product is computed according to the usual algebraic rules.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	M : Read access of operand2 illegal. MR, M : R1 not even-numbered. M : D2(X2,B2) not a word boundary.

Programming notes

– With the MR instruction, R2=R1 or R2=R1+1 is permitted. If R2=R1+1, the square is determined from R2.

– The least and greatest possible value for the product are, respectively, $+2^{62}$ and $-2^{62}+2^{31}$.

Examples

The following values of multiplicand and multiplier yields the values shown below for the product:

Multiplicand	Multiplier	Product
+29	+17	+493
+29	-17	-493
Minimum and maximum values for the product :		
$+2^{31}-1$	-2^{31}	$-2^{62}+2^{31}$
-2^{31}	-2^{31}	$+2^{62}$

Monitor Call

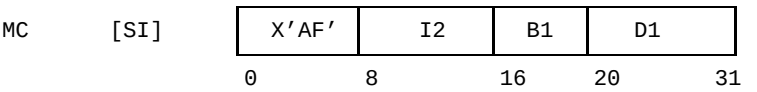
Function

The MC instruction creates a program interrupt due to a monitor call.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	MC	D1(B1),I2	X'00' ≤ I2 ≤ X'0F'

Machine format



Description

A program interrupt occurs when the mask bit for the monitor class determined by the 12 field of the instruction is set to =1.

The address value D1(B1) (either 24 bits or 31 bits long, depending on the address mode used) serves as the argument for the interrupt routine.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Addressing error Monitor interrupt	X'5C'	I2 > 15 see Programming Notes.

Programming notes

The MC instruction is not supported by BS2000. If it is called anyway, it functions in the same way as a NOP operation.

Multiply Halfword

Function

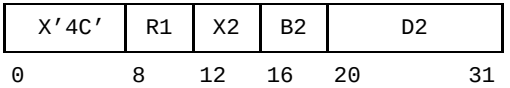
The MH instruction performs signed multiplication of a 32-bit fixed-point number and a 16-bit fixed-point number, and stores the lowest-order 32 binary positions of the product.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	MH	R1,D2(X2,B2)	D2(X2,B2): halfword boundary

Machine format

MH [RX]



Description

The multiplicand is taken from general-purpose register R1, the multiplier from the halfword addressed in main memory by D2(X2,B2). The lowest-order 32 binary positions of the product are stored in general-purpose register R1 and replace the multiplicand.

The multiplicand is treated as a 32-bit fixed-point number and the multiplier as a 16-bit fixed-point number with the sign at the highest-order bit position. The product is a 48-bit fixed-point number of which only the rightmost 32 binary positions are stored. The leftmost 16 binary positions, including the sign bit, are lost. There is no test to see whether the lost binary positions are identical to the value of the highest-order bit of the stored result.

The sign of the product is computed according to the usual algebraic rules.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand2 illegal.
Addressing error	X'5C'	D2(X2,B2) not a halfword boundary.

Programming notes

– Since the highest-order 16 bits of the real product are discarded, it may happen that the value and/or the sign position of the result differ from the sign or value of the real product. Even if this does happen, it is not indicated in the condition code. The MH instruction should therefore only be used when it is known that the product of the multiplicand and multiplier will lie within the range of -2^{31} and $+2^{31}-1$.

Examples

The following values for multiplicand and multiplier yield the values shown below for the result. Note that the result is only identical to the product when the product lies in the value range of 32-bit fixed-point numbers.

Multiplicand	Multiplier	Result	Remark
+29	+17	+493	arithmetically correct
+29	-17	-493	arithmetically correct
+131072	-32768	0	arithmetically incorrect correct is -4 295 464 296
+65538	+32767	+2 147 483 646	arithmetically correct

The last example illustrates an arithmetic limit for the MH instruction: values as small as +65539 and +32767 already yield an arithmetically unusable result.

Move Characters

Function

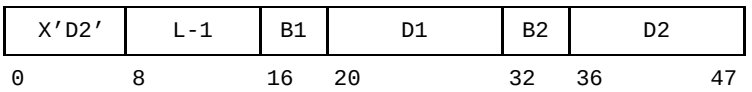
The MVC instruction moves 1 to 256 bytes from a main memory area to another main memory area.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	MVC	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine format

MVC [SS]



Description

The character field which is addressed by D2(B2) and has a length of L bytes is moved byte-to-byte from left to right to the main memory area addressed by D1(B1).

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand1 or read access of operand2 illegal.

Programming notes

- The operand fields may overlap.
- The overlap option of the MVC instruction can be put to use in order to "erase" a field, i.e. to pad it with a constant byte value. This is done by storing this byte value in byte 0 (D1(B1)) of the first operand (e.g. with MVI) and then executing an MVC with an operand1 address of D1(B1)+1 and an operand2 address of =D1(B1). In this way, byte 0 is "spread" over the first operand (see example).
- For field lengths >256 bytes the MVCL instruction has been provided.

Example

```
Name    Operation   Operands
        .
        MVI         FIELD,C' '                  "Erase" FIELD with
        MVC         FIELD+1(L'FIELD-1),FIELD    blanks
        .
```

Move Long

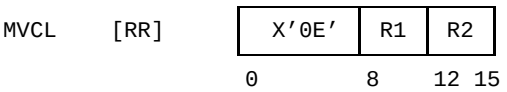
Function

The MVCL instruction moves the contents of a main memory area from left to right into another main memory area and, if necessary, pads this area to the right with slack bytes. Both areas may be up to 2^{24} bytes long i.e. 16 MB
The condition code is set in accordance with the difference in length of the two areas.

Assembler formats

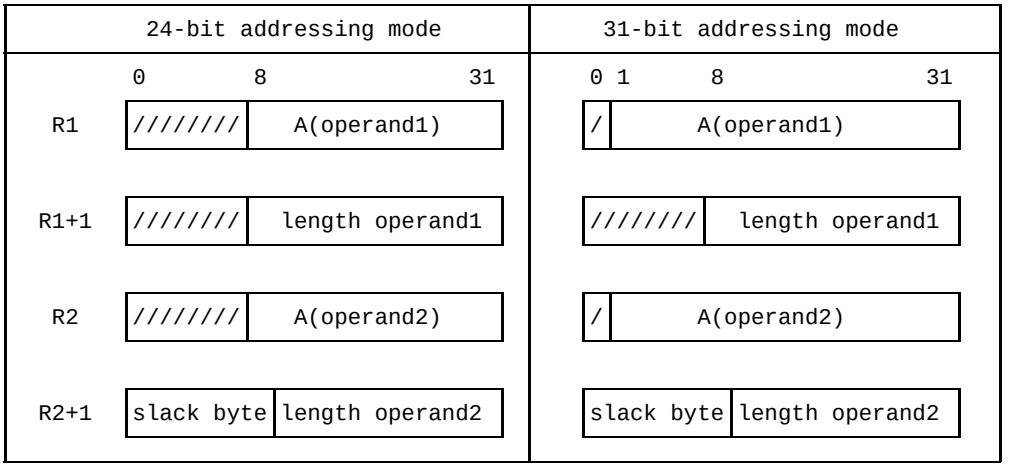
Name	Operation	Operands	Remarks
	MVCL	R1,R2	R1 and R2 even-numbered

Machine format



Description

The R1 field of the instruction determines the receive field, and the R2 field determines the source field. R1 and R2 each determine a pair of general-purpose registers, consisting of registers R1 and R1+1 or R2 and R2+1, respectively. R1 and R2 must be even-numbered, otherwise no move will take place and a program interrupt will occur due to an addressing error.
The start addresses of the receive field and the source field are taken from the first even-numbered register R1 (or R2). Their lengths (in bytes) are determined in the second, odd-numbered register R1+1 (or R2+1). Register R2+1 also contains the coding of the slack byte.
The address representation in R1 or R2 depends on which addressing mode is used. The following assignment applies:



"/" means: "is ignored"

The move operation takes place byte-by-byte from left to right. It ends when the number of bytes in the source field (as determined by R2+1) has been moved to the receive field. If this is not enough to reach the length of the receive field (as determined by R1+1), the receive field is padded with slack bytes whose coding is taken from the highest-order byte of R2+1.

The move operation will only be performed if the receive field does not overlap with the source field, or if the overlap occurs in such a way that the receive field does not begin to the right of the source field. The following rules apply for correct overlapping:

A(receive fld) ≤ A(source fld)
or A(receive fld) ≥ A(source fld) + Min(L'receive fld, L'source fld)

If incorrect (or "destructive") overlapping occurs, the instruction does not start and the condition code is set to 3~Overflow.

The MVCL instruction can be interrupted on the hardware side. When an interrupt occurs, the moves made up to that point are retained in the register pairs R1 and R2 (by storing the incremented addresses and the decremented lengths). Following the interrupt the move operation resumes at the position where the interrupt occurred.

When the instruction is finished, i.e. the move is complete and any necessary slack bytes have been added, the following values are stored in register pairs R1 and R2: the addresses in R1 and R2 are incremented by the length value in registers R1+1 and R2+1 respectively; registers R1+1 and R2+1 contain 00_{16} in their lowest-order 3 bytes; the leftmost 1 or 8 bits in front of the addresses in R1 and R2 are set to 0, but the leftmost 8 bits of R1+1 and R2+1 are left unchanged (slack byte).

Address updating in the source field and receive field take place with mod 2^{24} in 24-bit addressing mode and with mod 2^{31} in 31-bit addressing mode. Accordingly, once a move has taken place from or to the byte with the (virtual) address 2^{24}-1 or 2^{31}-1, the next move (or padding) operation will take place from or to the byte with the address 0, provided the operands have not finished being processed.

Condition code

0~Equal	length of receive field = length of source field
1~Low	length of receive field < length of source field
2~High	length of receive field > length of source field
3~Overflow	Receive field overlaps incorrectly with source field.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand1 or read access of operand2 illegal.
Addressing error	X'5C'	R1 or R2 not even-numbered

Programming notes

- If the length of the receive field is =0, no move or padding takes place, and only the condition code is set.
- if the length of the source field is =0, the source field is only padded with slack bytes. In this way, for example, a receive field can be "erased", i.e. padded with a constant byte value.
- The MVCL instruction cannot be used to erase a receive field in the way that is possible and customary with the MVC instruction, namely by "spreading" its byte 0. The reason for this is that when the address of the receive field, incremented by 1, is used as a source field, MVCL causes the instruction to abort due to incorrect overlapping.
- The check for incorrect overlapping, which takes place at the start of execution is made on the basis of the data in R1 and R2. If incorrect overlapping is detected, the instruction is aborted, leaving the receive field unchanged. No further check takes place, so that, for example, it is not known whether all source and receive field addresses have also been allocated by the operating system.

- Another way of interpreting the conditions for correct overlapping is as follows: the receive field must lie in such a relation to the source field that no byte has to be moved twice.
- If the length of the source field is =0 or =1, incorrect overlapping is impossible.
- In multiprocessor applications you may have to note the following: Since the instruction can be interrupted on the hardware side, the receive field may not have been completely filled (or erased) when it is accessed by another central processing unit.
- The application program must determine on its own whether all addresses of both operands for the program lie entirely within its own address space. If the instruction terminates due to an address translation error, the move operation may already have begun.
- Since the MVCL instruction can be interrupted by central processing units working in parallel, you should not move the MVCL instruction that activates the move operation. Similarly, you should not move an EX instruction that executes an MVCL instruction.

Example

The following instructions move 15000 bytes from area SF to area DF and pad the next 5000 bytes in area DF with the character '*'.

```
Name    Operation   Operands
        LM          4,5,=A(DF,20000)      R4,R5 : operand1
        LM          10,11,=A(SF,15000)    R10,R11 : operand2
        ICM         11,B'1000',='*'       Set slack byte in byte 0
        MVCL        4,10
```

Following MVCL the condition is set to 2~High (20000 > 15000). Register 4 or 10 contains the address A(DF+20000) or A(SF+15000); registers 5 and 11 contain 00 00 00 in their rightmost 3 bytes and the value 00_{16} or the character '*' in their leftmost byte.
The prerequisite for this result is that area DF starts either before SF or after SF+14999, or that A(DF)=A(SF) (otherwise the condition code is set to 3~Overflow and no move took place).

Move Immediate

Function

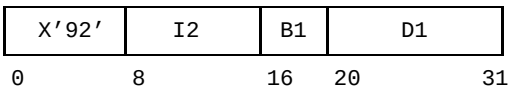
The MVI instruction moves one byte (direct operand) to main memory.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	MVI	D1(B1),I2	X'00' ≤ I2 ≤ X'FF'

Machine format

MVI [SI]



Description

The direct operand I2 replaces the main memory byte addressed by D1(B1).

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand1 illegal.

Example

See example under MVC.

Move Numerics

Function

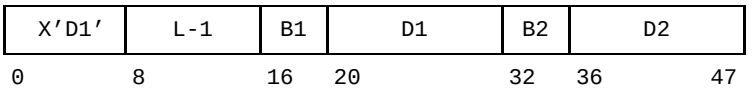
The MVN instruction moves the rightmost halfbytes of a main memory area to the rightmost halfbytes of another main memory area.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	MVN	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine format

MVN [SS]



Description

The rightmost halfbytes of the character field which is addressed by D2(B2) and has the length L bytes (i.e. the numeric parts) are moved to the rightmost halfbytes of the character field addressed by D1(B1); the leftmost halfbytes of the first operand are left unchanged.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.

Programming notes

– The operands may overlap.

– The overlap option of MVN can be used in order to "erase" the numeric parts of the field, i.e. to fill them with a constant value. To do this, you store this value in byte 0 of the first operand (e.g. with OI) and then perform an MVN whose first operand address is D1(B1)+1 and whose second operand address is D1(B1). This "spreads" the right portion of byte 0 over the first operand.

Example

```
Name    Operation   Operands
        .
        XC          DFIELD(3),DFIELD      DFIELD : X'000000'
        MVN         DFIELD(3),=C'123'     DFIELD after: X'010203'
        .
```

Move with Offset

Function

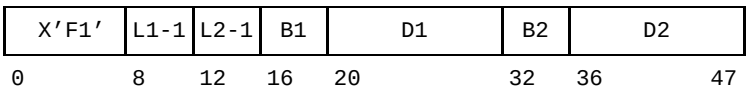
The MVO instruction moves a character field in main memory one halfbyte to the left into another character field.
The condition code is set in accordance with the comparison result.

Assembler format

Name	Operation	Operands	Remarks
	MVO	D1(L1,B1),D2(L2,B2)	1 ≤ L1,L2 ≤ 16

Machine format

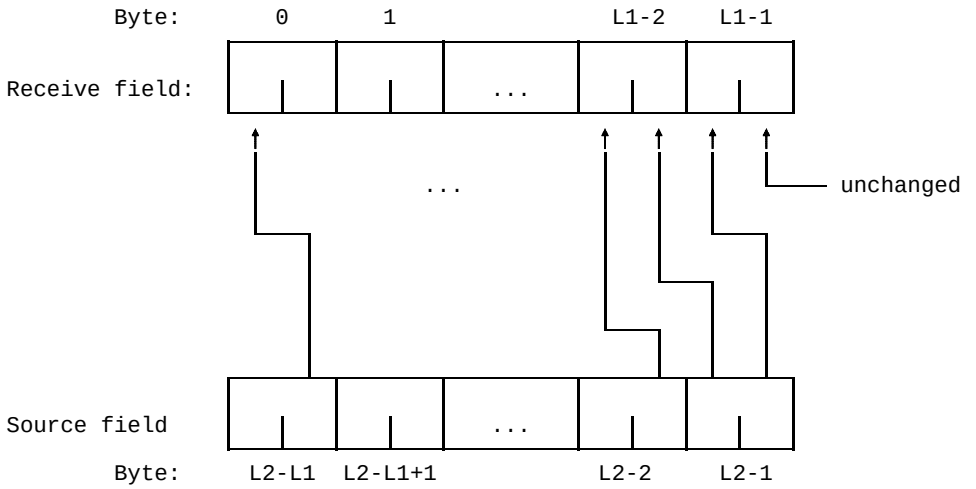
MVO [SS]



Description

The character field addressed in main memory by D1(B1) (L1 bytes long) is the receive field; the character field addressed in main memory by D2(B2) (L2 bytes long) is the source field.

The move operation takes place from right to left. The rightmost 4 bits of each byte in the source field are moved into the leftmost 4 bits of the opposing byte in the receive field, and the leftmost 4 bits are moved into the rightmost 4 bits of the preceding byte in the receive field. The rightmost 4 bits of the lowest-order byte in the receive field are left unchanged.



(In this diagram, L1 is assumed to be less or equal to L2; otherwise the highest-order byte of the receive field is the byte L1-L2-1, and the highest-order byte in the source field is byte 0.)

The receive field is padded to the left with 0_{16} if it is longer than the source field; if the receive field is too short to accommodate all halfbytes of the source field, the highest-order halfbytes of the source field are lost.

The receive field may overlap with the source field. The move operation is performed as though each byte of the receive field is stored the moment both of its halfbytes have been determined.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.

Programming notes

The MVO instruction can be used to move a packed decimal number to the right by an odd number of decimal positions (see example). However, the decimal number is not checked to see whether it is correctly packed.

Example

Name	Operation	Operands
	. MVO .	FIELD,FIELD(L'FIELD-2)

The above instruction moves the contents of FIELD 3 halfbytes to the right, but leaves the rightmost byte of FIELD unchanged. For example, FIELD-before =X'ABCDEF' is changed to FIELD-after =X'000ABF'.
If the contents of FIELD are a packed decimal number, the result is equivalent to integral division by 1000.

Move Zones

Function

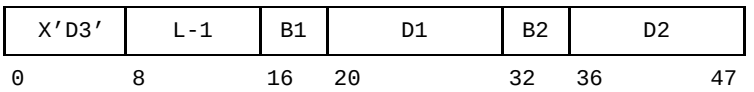
The MVZ instruction moves the leftmost halfbytes of a main memory area to the leftmost halfbytes of another main memory area.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	MVZ	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine format

MVZ [SS]



Description

The leftmost halfbytes of the character field which is addressed by D2(B2) and is L bytes long (i.e. the zone parts) are moved from left to right into the leftmost halfbytes of the character field addressed by D1(B1); the rightmost halfbytes of the first operand are left unchanged.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.

Programming notes

– The operands may overlap.

– The overlap option of MVZ can be used in order to "erase" the numeric parts of a field, i.e. to fill them with a constant value. To do this, you store this value in byte 0 of the first operand (e.g. with NI and OI) and then perform an MVZ whose first operand address is D1(B1)+1 and whose second operand address is D1(B1). This "spreads" the left portion of byte 0 over the first operand (see example).

Example

```
Name      Operation   Operands
          .
          NI          DFIELD,X'0F'          Sets filler zone
          OI          DFIELD,X'C0'          in byte 0
          MVZ         DFIELD+1(L'DFIELD-1),DFIELD
*                                           All leftmost halfbytes
*                                           are set to =C16.
*                                           The rightmost halfbytes
*                                           are left unchanged.
          .
```

AND

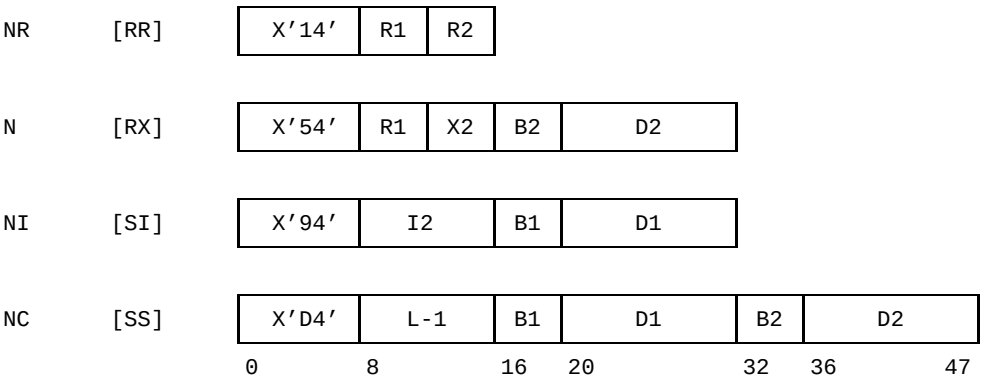
Function

The instructions NR, N and NC cause two operands to be ANDed bit by bit.
The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	NR N NI NC	R1,R2 R1,D2(X2,B2) D1(B1),I2 D1(L,B1),D2(B2)	 D2(X2,B2): word boundary X'00' ≤ I2 ≤ X'FF' 1 ≤ L ≤ 256

Machine formats



Description

The bits of the first operand are changed by the opposing bits of the second operand in accordance with the following table. The result replaces the first operand.

Table of AND conjunctions

Bit value in first operand	Bit value in second operand	Bit value in result
0	0	0
0	1	0
1	0	0
1	1	1

Operands

Instr.	Operand1	Operand2
NR	Contents of register R1	Contents of register R2
N	Contents of register R1	Word addressed by D2(X2,B2)
NI	Byte addressed by D1(B1)	Direct operand I2
NC	Field addressed by D1(B1) with length of L bytes	Field addressed by D2(B2) with length of L bytes

Condition code

0~Zero	result = 0
1~Not Zero	result ≠ 0
2	Not used.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	N: Read access of operand2 illegal. NI: Read/write access of operand1 illegal. NC: Read/write access of operand1 or read address of operand2 illegal.
Addressing error	X'5C'	N: D2(X2,B2) not a word boundary.

Programming notes

- AND instructions set all bit positions in the first operand to 0 whose opposing bit positions in the second operand are 0. The other bit positions in the first operand are left unchanged.
- The operands are processed byte-by-byte from left to right.
- With NC, the operands may overlap. However, among other things, this means that earlier byte operands are changed by later ones.
- If R1=R2 in the NE instruction, the contents of R1 are not changed, but the condition code is set.
- When using the NI and NC instructions in multiprocessor systems, note the following:
  Memory access operations of the first operand of the NI and NC instructions consist of reading a byte from memory and then writing the changed value into memory. These read and write operations on a single byte are not necessarily consecutive, if another processor or another application (or an input/output channel program) attempts to modify the memory location in question. A safe way of updating a shared word in memory is described in Appendix 7.6 and in the programming notes for the CS and CDS instructions.

Example

```
Name    Operation  Operands
                .
        NI      SEMAPHOR,X'F0'      Set rightmost 4 bits of
*                                   byte SEMAPHOR to 0(16);
*
*                                   left unchanged.
                .
```

OR

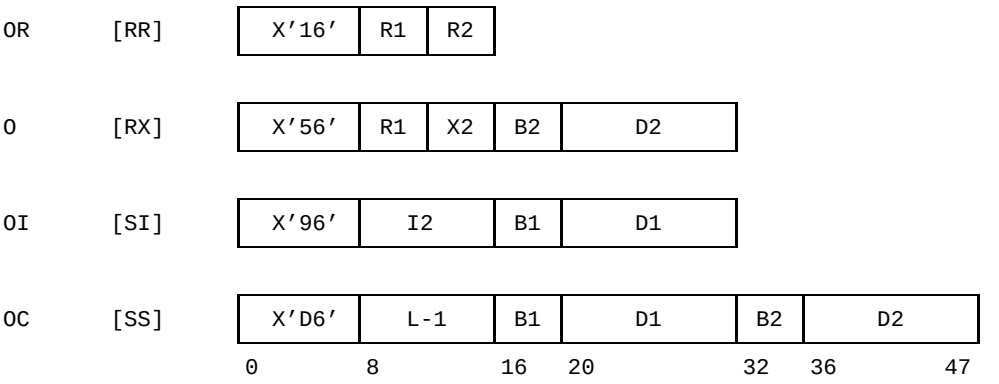
Function

The instructions OR, O, OI and OC cause two operands to be logically ORed bit by bit. The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	OR O OI OC	R1,R2 R1,D2(X2,B2) D1(B1),I2 D1(L,B1),D2(B2)	 D2(X2,B2): word boundary X'00' ≤ I2 ≤ X'FF' 1 ≤ L ≤ 256

Machine formats



Description

The bits of the first operand are changed by the opposing bits of the second operand according to the following table. The result replaces the first operand.

Table of OR conjunctions

Bit value in first operand	Bit value in second operand	Bit value in result
0	0	0
0	1	1
1	0	1
1	1	1

Operands

Instr.	Operand1	Operand2
OR	Contents of register R1	Contents of register R2
O	Contents of register R1	Word addressed by D2(X2,B2)
OI	Byte addressed by D1(B1)	Direct operand I2
OC	Field addressed by D1(B1) with length of L bytes	Field addressed by D2(B2) with length of L bytes

Condition code

0~Zero result = 0
1~Not Zero result ≠ 0
2 Not used.
3 Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	O: Read access of operand2 illegal. OI: Read/write access of operand1 illegal. OC: Read/write access of operand1 or read access of operand2 illegal.
Addressing error	X'5C'	O: D2(X2,B2) not a word boundary.

Programming notes

- OR instructions set all bit positions in the first operand to 1 for which, in the second operand, there is an opposing bit position with the value 1. The other bit positions in the first operand are left unchanged.
- The operands are processed byte-by-byte from left to right.
- With OC, the operands may overlap. However, among other things, this means that earlier byte operations are changed by later ones.
- If R1=R2 in the OR instruction, i.e. general-purpose register R1 is ORed with itself, the contents of R1 are not changed, but the condition code is set.
- When using the OI and OC instructions in multiprocessor systems, note the following:
  Memory access operations of the first operand of the OI and OC instructions consist of reading a byte from memory and then writing the changed value into memory. These read and write operations on a single byte are not necessarily consecutive, if another processor or another application (or an input/output channel program) attempts to modify the memory location in question. A safe way of updating a shared word in memory is described in Appendix 7.6 and in the programming notes for the CS and CDS instructions.

Pack

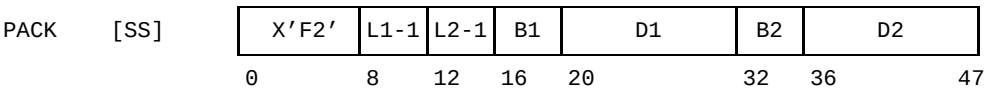
Function

The PACK instruction turns an (unpacked) decimal number in the source field into a packed decimal number in the receive field.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	PACK	D1(L1,B1),D2(L2,B2)	

Machine format



Description

D1(L1,B1) addresses the receive field and D2(L2,B2) addresses the source field (where $1 \leq L1,L2 \leq 16$). The (unpacked) decimal number contained in the source field is moved to the receive field and converted to packed format.

The source field is *not* checked to see whether it contains a correct, unpacked decimal number. Instead, it is treated as though it contains one.

Both operands are processed from right to left. Only the right halfbyte (the numeric part) of each byte in the source field is used; each left halfbyte is ignored, except for the left halfbyte in the lowest-order byte of the source field, which is used for the sign.

The sign and the right halfbyte of the lowest-order byte of the source field are moved - in opposite order - to the lowest-order byte of the receive field. All other right halfbytes in the source field are moved consecutively to the other bytes of the receive field, with two halfbytes of the source field always being moved to one byte in the receive field.

If the source field is exhausted before the receive field is filled, i.e. if L2 < 2L1-1, the highest-order 2L1-L2-1 halfbytes of the receive field are filled with 0_{16}. If the receive field is too short to accommodate all right halfbytes of the source field, i.e. if 2L1 < L2+1, the highestorder L2-2L1+1 bytes of the source field are ignored.

The two operands may overlap. In this case, a subsequent byte operation will generally change an earlier operation of the same instruction. The instruction is executed as though each byte in the receive field is stored the moment the halfbytes which it needs have been read in the source field.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand1 or read access of operand2 illegal.

Examples

The sample PACK instruction given below yield the following results:

DFIELD before	Sample instruction	DFIELD after
any	PACK DFIELD(1),=Z'1'	P'1'
any	PACK DFIELD(3),=Z'123'	P'00123'
any	PACK DFIELD(1),=Z'123'	P'3' plus decimal overflow
X'89'	PACK DFIELD(1),DFIELD(1)	X'98'
X'23456789'	PACK DFIELD(2),DFIELD(4)	X'87986789'

The last example illustrates a (hopefully warning) instance of overlapping in which the source field overwrites itself (!) when the instruction is executed.

Subtract

Function

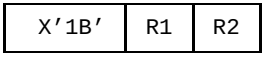
The instructions SR and S perform signed subtraction of two 32-bit fixed-point numbers. The condition code is set in accordance with the value of the difference.

Assembler formats

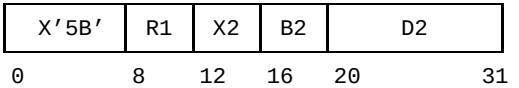
Name	Operation	Operands	Remarks
	SR S	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats

SR [RR]



S [RX]



Description

The SR instruction subtracts the contents of general-purpose register R2 from the contents of general-purpose register R1, taking the signs into account; the S instruction subtracts the word addressed in main memory by D2(B2) from the contents of general-purpose register R1, likewise taking the signs into account. Both operands are treated as 32-bit signed binary numbers (fixed-point numbers). The difference is likewise a 32-bit signed binary number, and replaces the original contents of general-purpose register R1.

Fixed-point overflow occurs when the difference is greater than -2^{31} or less than -2^{31}. In this case, the result in R1 is 2^{32} too small or too large; the condition code is then set to 3~Overflow and a program interrupt occurs, provided the bit for fixed-point overflow in the program mask has been set to 1 (default value in BS2000).

Condition code

0~Zero	difference = 0
1~Minus	difference < 0
2~Plus	difference > 0
3~Overflow	fixed-point overflow

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	S: Read access of operand2 illegal.
Addressing error	X'5C'	S: D2(X2,B2) not a word boundary.
Fixed-point overflow	X'78'	Difference > $+2^{31}-1$ or < -2^{31}

Programming notes

– Fixed-point overflow occurs whenever a binary position overflow to the sign position is not equal to the binary position overflow from the sign position. The result, in register R1, then has the wrong sign at bit position 0.

– SR with R1=R2 "zeros" general-purpose register R1 and sets the condition code to 0~Zero. (SLR with R1=R2 likewise zeros general-purpose register R1 but sets the condition code to 2~Plus).

Subtract Halfword

Function

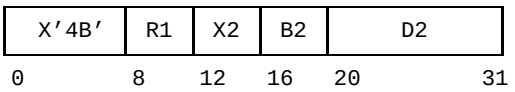
The SH instruction performs signed subtraction of a 16-bit fixed-point number from a 32-bit fixed-point number.
The condition code is set in accordance with the value of the difference.

Assembler format

Name	Operation	Operands	Remarks
	SH	R1,D2(X2,B2)	D2(X2,B2): halfword boundary

Machine format

SH [RX]



Description

The halfword addressed in main memory by D2(X2,B2) is subtracted from the contents of general-purpose register R1, with the signs being taken into account. The register operand is treated as a 32-bit fixed-point number, the halfword operand as a 16-bit fixed-point number, both of them signed. The difference is a 32-bit signed fixed-point number, and replaces the original contents of general-purpose register R1.
Fixed-point overflow when the difference is greater than 2^{31}-1 or less than -2^{31}. In this case, the result in R1 is 2^{32} too small or too large; the condition code is then set to 3~Overflow and a program interrupt occurs, provided the bit for fixed-point overflow in the program mask has been set to =1 (default value in BS2000).

Condition code

0~Zero	difference = 0
1~Minus	difference < 0
2~Plus	difference > 0
3~Overflow	overflow

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand2 illegal.
Addressing error	X'5C'	D2(X2,B2) not a halfword boundary.
Fixed-point overflow	X'78'	Difference > $+2^{31}-1$ or < -2^{31}

Programming notes

Fixed-point overflow occurs whenever a binary position overflow to the sign position is not equal to the binary position overflow from the sign position. The result in register R1, then has the wrong sign at bit position 0.

Subtract Logical

Function

The instructions SLR and SL perform logical (unsigned) subtraction of two 32-bit binary numbers.
The condition code is set in accordance with the value of the difference.

Assembler formats

Name	Operation	Operands	Remarks
	SLR SL	R1,R2 R1,D2(X2,B2)	 D2(X2,B2): word boundary

Machine formats

Description

The SLR instruction logically subtracts the contents of general-purpose register R2 from the contents of general-purpose register R1; the SL instruction subtracts the contents of the word addressed in main memory by D2(X2,B2) from the contents of general-purpose register R1.

Both operands are treated as 32-bit unsigned binary numbers.

The difference is likewise a 32-bit unsigned binary number, and replaces the original contents of general-purpose register R1.

All 32 bits of both operands are involved in the subtraction operation. Any carry over beyond bit position 0 is shown in the condition code.

Condition code

0	Not used (see Programming Notes).
1~Minus	difference ≠0, no overflow
2~Plus	difference =0, overflow
3~Overflow	difference ≠0, overflow

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	SL: Read access of operand2 illegal.
Addressing error	X'5C'	SL: D2(X2,B2) not a word boundary.

Programming notes

– Logical subtraction consists in adding the ones complements of the two operands and also adding 1 to the contents of general-purpose register R1 (i.e. not in adding their twos complement). For this reason, whenever the second operand is =0 an overflow always occurs, as is shown by the condition code value 2 or 3.

– A resulting difference of =0 always creates a condition code of 2~Plus, not 0~Zero.

– Logical subtraction always creates the same result as arithmetic subtraction (by means of SR, S or SH), except that the condition code is set differently and no program interrupt occurs in case of overflow.

– Another way of interpreting the condition code values is as follows:

0	Not used.
1~Minus	operand1 < operand2
2~Plus	operand1 = operand2
3~Overflow	operand1 > operand2

– The SL instruction can find use with signed subtraction of fixed-point numbers which are more than 32 bits long. This is done by using SL instructions to subtract the lower-order word pairs and using the S instruction to subtract the highest-order word pair; if, after subtracting a lowest-order word pair, the condition code is set to 1~Minus (i.e. the operand1 word was smaller than the operand2 word), the number +1 must be subtracted from the difference of the next higher-order word pair (see example2).

Example

```
Name        Operation   Operands
            .
Example1    L           10,=F'1'
            SL          10,=F'1'            Register 10: 0
*                                           but CC =2, not =0
*                                           see Programming Notes
            .
            .
Example2    LM          0,1,FPNO1           Subtraction of two 64-bit
LOWSUB      SL          1,FPNO2+4           fixed-point numbers
            BNM         HIGHSUB
            SH          0,=H'1'             FPNO1+4 was < FPNO2+4
HIGHSUB     S           0,FPNO2
            .
```

Example 2 illustrates signed subtraction of two 64-bit fixed-point numbers FPNO1 and FPNO2: the lower-order word pair is subtracted using SL and the higher-order word pair using S. If the lower-order word pair produces an overflow when subtracted, +1 must be subtracted from the difference of the higher-order word pair. In the example, the result is located in general-purpose registers 0 and 1.

Shift Left Single

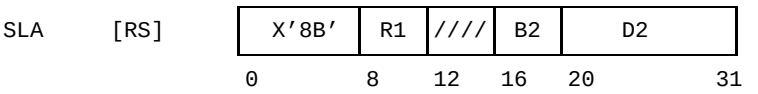
Function

The SLA instruction shifts a 32-bit fixed-point number in a general-purpose register a specified number of binary positions to the left, taking the sign into account.
The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	SLA	R1,D2(B2)	
* or also:			
	SLA	R1,<number>	

Machine format



Description

The contents of general-purpose register R1 are treated as a 32-bit fixed-point number with the sign at bit position 0.

The address determined by D2(B2) is not used as the data address, instead, the rightmost 6 bits of this address form the number of binary positions by which the fixed-point number is to be shifted to the left. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

With shift left, the sign is left unchanged; only the remaining 31-bit positions are shifted. Bit positions freed to the right are filled with 0; bit positions shifted to the left beyond bit position 1 or R1 are lost.

If one or more bits other than the sign bit are shifted beyond bit position 1 in register R1, a fixed-point overflow occurs and the condition code is set to 3~Overflow. Furthermore, if the bit for fixed-point overflow is set to 1 in the program mask (default value in BS2000), a program interrupt occurs.

Bit positions 12 through 15 in the instruction are ignored.

Condition code

0~Zero	shifted fixed-point number = 0
1~Minus	shifted fixed-point number < 0 (bit 0 of R1 =1)
2~Plus	shifted fixed-point number > 0 (bit 0 of R1 =0)
3~Overflow	One or more bits other than the sign bit were shifted beyond bit position 1 of R1.

Program interrupts

Type	Weight	Causes
Fixed-point overflow	X'78'	see condition code 3~Overflow

Programming notes

- If B2=0, D2 alone determine the number of shifts; in this case the B2 entry may be omitted from the Assembler format.
- If the number of shifts is =0 mod 64, register R1 is not changed, but the condition code is set.
- Shifting by a variable number of bit positions is achieved by loading the variable in general-purpose register B2.

Examples

The examples below yield the following results.

Register 0 before	Sample instruction	Register 0 after	CC
0...001 (+1)	SLA 0,1	0...010 (+2)	2
1...111 (-1)	SLA 0,1	1....10 (-2)	1
0...001 (+1)	SLA 0,30	010...0 ($+2^{30}$)	2
0...001 (+1)	SLA 0,31	0.....0 (0)	3
10...00 (-2^{31})	SLA 0,1	10....0 (-2^{31})	3
10...00 (-2^{31})	SLA 0,128	10....0 (-2^{31})	1

Note the two cases of fixed-point overflow (CC =3): this fixed-point overflow comes about because a non-sign bit was shifted beyond bit position 1. In these cases, the condition code 3~Overflow indicates that the result is arithmetically incorrect.
The last example illustrates a case without shift: only the lowest-order 6 bits of the shift number are used, and with 128 these yield the value 0. Register 0 is left unchanged, but the condition code is set.

Shift Left Double

Function

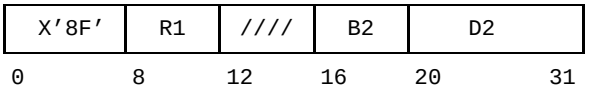
The SLDA instruction shifts a 64-bit fixed-point number in a general-purpose register pair by a specified number of binary positions to the left, taking the sign into account. The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	SLDA	R1,D2(B2)	R1 even-numbered
* or also:	SLDA	R1,<number>	R1 even-numbered

Machine format

SLDA [RS]



Description

The R1 field of the instruction defines a pair of general-purpose registers, consisting of registers R1 and R1+1; R1 must be even-numbered, otherwise a program interrupt will occur due to an addressing error.

Bit positions 12 to 15 of the instruction are ignored.

The address defined D2(B2) is not used as the data address; instead, the rightmost 6 bits of this address form the number of binary positions by which the fixed-point number is to be shifted to the left. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

The contents of the general-purpose register pair R1 and R1+1 are treated as a 64-bit signed fixed-point number. The sign at bit position 0 of the (even-numbered) register R1 is left unchanged, but all other 63 bit positions are shifted. Bit positions freed from the right are padded with 0, binary positions shifted to the left beyond bit position 1 of R1 are lost.

If one or more bits other than the sign bit are shifted beyond bit position 1 in register R1, a fixed-point overflow occurs and the condition code is set to 3~Overflow. Furthermore, if the bit for fixed-point overflow is set to 1 in the program mask (default value in BS2000), a program interrupt occurs.

Condition code

0~Zero	shifted fixed-point number = 0
1~Minus	shifted fixed-point number < 0 (bit 0 of R1 =1)
2~Plus	shifted fixed-point number > 0 (bit 0 of R1 =0)
3~Overflow	One or more bits other than the sign bit were shifted beyond bit position 1 of R1.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	R1 not even-numbered.
Fixed-point overflow	X'78'	see CC 3~Overflow

Programming notes

- If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.
- If the number of shifts is =0 mod 64, R1 and R1+1 are left unchanged, but the condition code is set.
- Shifting by a variable number of bit positions is achieved by loading the variable in general-purpose register B2.

Examples

The sample instructions below yield the following results:

Register 0,1 before	Sample instr.	Register 0,1 after	CC
0...0 0..01 (+1)	SLDA 0,1	00...0 0..010 (+2)	2
1...1 1...1 (-1)	SLDA 0,1	11...1 1...10 (-2)	1
01..1 1...1 ($+2^{63}-1$)	SLDA 0,1	01...1 1...10 ($+2^{63}-2$)	3
0...0 1...1 ($+2^{32}-1$)	SLDA 0,31	01...1 10...0 ($+2^{63}-2^{31}$)	2
10..0 0...0 (-2^{63})	SLDA 0,1	10...0 0....0 (-2^{63})	3
10..0 0...0 (-2^{63})	SLDA 0,64	10...0 0....0 (-2^{63})	1

Note the two cases of fixed-point overflow (CC=3): this fixed-point overflow comes about because a non-sign bit was shifted beyond bit position 1. In these cases, the condition code 3~Overflow indicates that the result is arithmetically incorrect.
The last example illustrates a case without shift: only the lowest-order 6 bits of the shift number are used, and with 64 these yield the value 0. Register 0 is left unchanged, but the condition code is set.

Shift Left Double Logical

Function

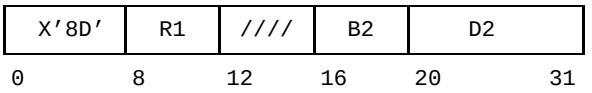
The SLDL instruction shifts a 64-bit binary number in a general-purpose register pair a specified number of binary positions to the left. The sign is not taken into account (logical shift).
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	SLDL	R1,D2(B2)	R1 even-numbered
* or also:			
	SLDL	R1,<number>	R1 even-numbered

Machine format

SLDL [RS]



Description

The R1 field of the instruction defines a pair of general-purpose registers, consisting of registers R1 and R1+1; R1 must be even-numbered, otherwise a program interrupt will occur due to an addressing error.

Bit positions 12 to 15 of the instruction are ignored.

The address defined by D2(B2) is not used as the data address; the rightmost 6 bits of this address form the number of binary positions by which the fixed-point number is to be shifted to the left. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

The contents of the general-purpose register pair R1 and R1+1 are treated as a 64-bit unsigned fixed-point number. All 64 binary positions in this number are shifted. Bit positions freed from the right are padded with 0; binary positions shifted beyond bit positions 0 are lost.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	R1 not even-numbered.

Programming notes

– If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.

– If the number of shifts is =0 mod 64, R1 and R1+1 (and the condition code as well) are not changed.

– Shifting by a variable number of bit positions is achieved by loading the variable into general-purpose register B2.

Examples

The sample instructions below yield the following results:

Register 0,1 before	Sample instr.	Register 0,1 after	CC
0...0 0..01	SLDL 0,1	00...0 0..010	unchanged
1...1 1...1	SLDL 0,1	11...1 1...10	unchanged
01..1 1...1	SLDL 0,1	1....1 1...10	unchanged
0...0 1...1	SLDL 0,31	01...1 10...0	unchanged
10..0 0...0	SLDL 0,1	0....0 0....0	unchanged
10..0 0...0	SLDL 0,64	10...0 0....0	unchanged

The reader should compare these examples with those given in the description of the SLDA instruction. Here, there is not a single case of fixed-point overflow. The last example illustrates (as with SLDA) a case without shift: only the lower-order 6 bits of the shift number are used, and with 64 these yield the value 0. The register contents and the condition code are left unchanged.

Shift Left Single Logical

Function

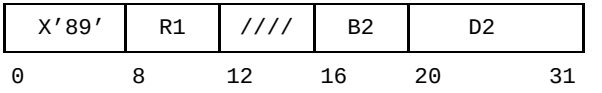
The SLL instruction shifts a 32-bit binary number in a general-purpose register a specified number of binary positions to the left. The sign is not taken into account. The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	SLL	R1,D2(B2)	
* or also:			
	SLL	R1,<number>	

Machine format

SLL [RS]



Description

The contents of general-purpose register R1 are treated as a 32-bit unsigned binary number.

The address defined by D2(B2) is not used as the data address; instead, the rightmost 6 bits of this address form the number of binary positions by which the binary number is to be shifted to the left. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

With shift left, all 32 bit positions are shifted. Bit positions freed from the right are padded with 0; binary positions shifted to the left beyond bit position 0 are lost.

Bit positions 12 to 15 of the instruction are ignored.

Condition code

Stays the same.

Program interrupts

None.

Programming notes

- If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.
- If the number of shifts is =0 mod 64, register R1 (and the condition code) are left unchanged.
- Shifting by a variable number of bit positions is achieved by loading the variable in general-purpose register B2.

Examples

The sample instructions below yield the following results:

Register 0 before	Sample instr.	Register 0 after	CC
00....01	SLL 0,1	00....010	unchanged
11....11	SLL 0,1	11....110	unchanged
10....00	SLL 0,1	00....000	unchanged
10....00	SLL 0,64	10....000	unchanged

The last example illustrates a case without shift: only the lowest-order 6 bits of the shift number are used, and with 64 these yield the value 0. Neither the registers nor the condition code is changed.

Set Program Mask

Function

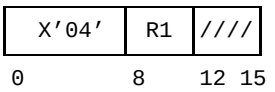
The SPM instruction sets the program mask and the condition code to specified values. The condition code is set in accordance with the new condition code value.

Assembler format

Name	Operation	Operands	Remarks
	SPM	R1	

Machine format

SPM [RR]



Description

Bit positions 2 and 3 of general-purpose register R1 replace the (previous) value of the condition code, and bit positions 4 to 7 replace the (previous) value of the program mask.

Bit positions 0 and 1 as well as 8 to 31 of general-purpose register R1 are ignored. Similarly, bit positions 12 to 15 of the SPM instruction are ignored.

Condition code

0~Equal	Bit positions 2 and 3 of R1 are $=00_2$.
1~Low	Bit positions 2 and 3 of R1 are $=01_2$.
2~High	Bit positions 2 and 3 of R1 are $=10_2$.
3~Overflow	Bit positions 2 and 3 of R1 are $=11_2$.

Program interrupts

None.

Programming notes

– The SPM instruction makes it possible to change the program mask, which is preset by BS2000 to $(1111)_2$. This enables the application program to suppress the customary program interrupt when any of the four types of events listed below occur. There can be good reasons for doing this. For example, in this way you can regularly switch off the unavoidable program interrupts due to significance which occur in programs that carry out intensive floating-point operations.
However, it is good programming style to reset the program mask afterwards to its original state.

– The bits in the program mask have the following meaning (from left to right):

Bit in program mask	Bit pos. in R1	Meaning
0	4	Fixed-point overflow
1	5	Decimal overflow
2	6	Exponent underflow
3	7	Significance

Example

After

```
Name        Operation   Operands
            .
*                                                  exponent underflow
*                                                  |
*                                                  |
            ICM         15,B'1000',=B'00111100'
            SPM         15
*                                                  |
*                                                  significance
            .
```

the condition code is set to 3~Overflow and program interrupts due to exponent underflow and significance are suppressed.

Shift Right Single

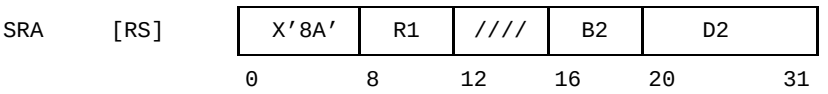
Function

The SRA instruction shifts a 32-bit fixed-point number in a general-purpose register a specified number of binary positions to the right. The sign is taken into account. The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	SRA	R1,D2(B2)	
* or also:			
	SRA	R1,<number>	

Machine format



Description

The contents of general-purpose register R1 are treated as a 32-bit fixed-point number with the sign at bit position 0.

The address defined by D2(B2) is used as the data address; instead, the rightmost 6 bits of this address form the number of binary positions by which the fixed-point number is to be shifted to the right. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

With shift right, the sign is left unchanged; only the remaining 31 bits are shifted. Bit positions freed from the left are padded with the value of this sign; binary positions shifted to the right beyond bit position 31 of R1 are lost.

Bit positions 12 to 15 of the instruction are ignored.

Condition code

0~Zero	shifted fixed-point number = 0
1~Minus	shifted fixed-point number < 0 (bit 0 of R1 =1)
2~Plus	shifted fixed-point number > 0 (bit 0 of R1 =0)
3	Not used.

Program interrupts

None.

Programming notes

- If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.
- If the number of shifts is 0 mod 64, R1 is left unchanged, but the condition code is set.
- Shifting negative fixed-point numbers to the right causes "downward rounding" to the next lowest negative integer. Thus, for example, the number -1 again yields the number -1 no matter how it is rightshifted; and the number -5, when shifted 2 binary positions to the right (i.e. when divided by 4), yields the number -2 (and not -1). For further information on this point see the examples.

Examples

The sample instructions below yield the following results:

Register 0 before	Sample instruction	Register 0 after	CC
0..0101 (+5)	SRA 0,2	0..0001 (+1)	2
1..1011 (-5)	SRA 0,2	1..1110 (-2)	1
0..0101 (+5)	SRA 0,3	0..0000 (0)	0
1..1011 (-5)	SRA 0,3	1..1111 (-1)	1
0..0001 (+1)	SRA 0,31	0..0000 (0)	0
1..1111 (-1)	SRA 0,31	1..1111 (-1)	1
01..111 ($+2^{31}-1$)	SRA 0,31	0..0000 (0)	0
10..001 ($-2^{31}+1$)	SRA 0,31	1..1111 (-1)	1

In each of the examples, right shifting of a positive fixed-point number is contrasted with right shifting of the negative pendant in order to point out the perhaps unfamiliar differences between them.

Shift Right Double

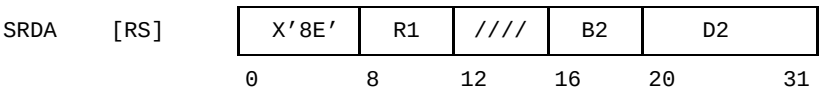
Function

The SRDA instruction shifts a 64-bit fixed-point number in a general-purpose register pair a specified number of binary positions to the right. The sign is taken into account. The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	SRDA	R1,D2(B2)	R1 even-numbered
* or also:			
	SRDA	R1,<number>	R1 even-numbered

Machine format



Description

The R1 field of the instruction defines a pair of general-purpose registers, consisting of registers R1 and R1+1; R1 must be even-numbered, otherwise a program interrupt will occur due to an addressing error.

Bit positions 12 to 15 of the instruction are ignored.

The address defined by D2(B2) is not used as the data address; instead, the rightmost 6 bits of this address form the number of binary positions by which the fixed-point number is to be shifted to the right. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

The contents of the general-purpose register pair R1 and R1+1 are treated as a 64-bit signed fixed-point number. The sign at bit position 0 of the (even-numbered) register R1 is left unchanged, but all other 63 bit positions are shifted. Bit positions freed from the left are padded with 0; binary positions shifted to the right beyond bit position 31 of R1+1 are lost.

Condition code

0~Zero	shifted fixed-point number = 0
1~Minus	shifted fixed-point number < 0 (bit 0 of R1 =1)
2~Plus	shifted fixed-point number > 0 (bit 0 of R1 =0)
3	Not used.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	R1 not even-numbered.

Programming notes

- If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.
- If the number of shifts is =0 mod 64, R1 and R1+1 are left unchanged, but the condition code is set.
- Shifting negative fixed-point numbers to the right causes "downward rounding" to the next lowest negative integer. Thus, for example, the number -1 again yields the number -1 no matter it is rightshifted; and the number -5, when shifted 2 binary positions to the right (i.e. when divided by 4), yields the number -2 (and not -1). For further information on this point see the examples.

Examples

The sample instructions below yield the following results:

Register 0,1 before	Sample instr.	Register 0,1 after	CC
0...0 0..0101 (+5)	SRDA 0,2	00...0 0..01 (+1)	2
1...1 1..1011 (-5)	SRDA 0,2	11...1 1..10 (-2)	1
0...0 0..0101 (+5)	SRDA 0,3	00...0 0..00 (0)	0
1...1 1..1011 (-5)	SRDA 0,3	11...1 1..11 (-1)	1
0...0 0...001 (+1)	SRDA 0,63	00...0 0..00 (0)	0
1...1 1.....1 (-1)	SRDA 0,63	11...1 1...1 (-1)	1
01..1 1.....1 ($+2^{63}-1$)	SRDA 0,63	00...0 0...0 (0)	0
10..0 0....01 ($-2^{63}+1$)	SRDA 0,63	11...1 1...1 (-1)	1

In each of the examples, right shifting of a positive fixed-point number is contrasted with right shifting of the negative pendant in order to point out the perhaps unfamiliar differences between them.

Shift Right Double Logical

Function

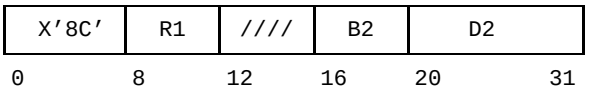
The SRDL instruction shifts a 64-bit binary number in a general-purpose register a specified number of binary positions to the right. The sign is not taken into account (logical shift).
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	SRDL	R1,D2(B2)	R1 even-numbered
* or also:			
	SRDL	R1,<number>	R1 even-numbered

Machine format

SRDL [RS]



Description

The R1 field of the instruction defines a pair of general-purpose registers, consisting of registers R1 and R1+1; R1 must be even-numbered, otherwise a program interrupt will occur due to an addressing error.

Bit positions 12 to 15 of the instruction are ignored.

The address defined by D2(B2) is not used as the data address; instead, the rightmost 6 bits of this address form the number of binary positions by which the fixed-point number is to be shifted to the left. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

The contents of the general-purpose register pair R1 and R1+1 are treated as a 64-bit unsigned fixed-point number. All 64 bits of this number are shifted. Bit positions freed from the left are padded with 0; binary positions shifted to the right beyond bit position 31 or R1+1 are lost.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	R1 not even-numbered.

Programming notes

– If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.

– If the number of shifts is 0 mod 64, R1 and R1+1 (and the condition code) are left unchanged.

Examples

The sample instructions below yield the following results

Register 0,1 before	Sample instr.	Register 0,1 after	CC
0...0 0..0101	SRDL 0,2	00...0 0..0001	unchanged
1...1 1..1011	SRDL 0,2	001..1 1..1110	unchanged
1...1 1.....1	SRDL 0,63	00...0 0..0001	unchanged
01..1 1.....1	SRDL 0,63	00...0 0..0000	unchanged

Unlike the SRDA instruction, with SRDL the value of bit position 0 in register R1 is not spread to the right; instead, bit positions freed from the left are always padded with 0.

Shift Right Single Logical

Function

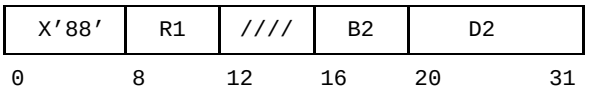
The SRL instruction shifts a 32-bit binary number in a general-purpose register a specified number of binary positions to the right. The sign is not taken into account (logical shift).
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	SRL	R1,D2(B2)	
* or also:			
	SRL	R1,<number>	

Machine format

SRL [RS]



Description

The contents of general-purpose register R1 are treated as a 32-bit unsigned fixed-point number.

The address defined by D2(B2) is not used as the data address; instead, the rightmost 6 bits of this address from the number of binary positions by which the fixed-point number is to be shifted to the right. This number lies between 0 and 63_{10}. The higher-order binary positions of D2(B2) are ignored.

All 32 binary positions are shifted to the right. Bit positions freed from the left are padded with 0. Binary positions shifted to the right beyond bit position 31 of R1 are lost.

Bit positions 12 to 15 of the instruction are ignored.

Condition code

Stays the same.

Program interrupts

None.

Programming notes

– If B2=0, D2 alone determines the number of shifts; in this case, the B2 entry may be omitted from the Assembler format.

– If the number of shifts is =0 mod 64, R1 (and the condition code) are left unchanged.

Examples

The sample instructions below yield the following results:

Register 0 before	Sample instr.	Register 0 after	CC
0..0101	SRL 0,2	0....01	unchanged
1..1011	SRL 0,2	001..10	unchanged
10....0	SRL 0,31	0....01	unchanged
01....1	SRL 0,31	0.....0	unchanged

Unlike the SRA instruction, with SRL the value of bit position 0 in register R1 is not spread to the right; instead, bit positions freed from the left are always padded with 0.

Store

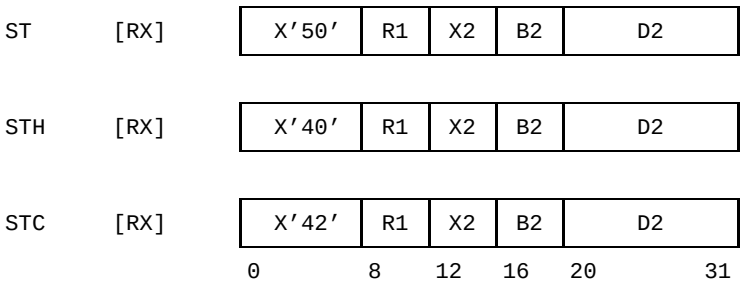
Function

The ST instruction moves the contents of a general-purpose register to a word in main memory.
The STH instructions moves bytes 2 and 3 of a general-purpose register to a halfword in main memory.
The STC instruction moves byte 3 of a general-purpose register to a byte in main memory.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
	ST STH STC	R1,D2(X2,B2) R1,D2(X2,B2) R1,D2(X2,B2)	D2(X2,B2): word boundary D2(X2,B2): halfword boundary

Machine formats



Description

ST: The contents of general-purpose register R1 are stored in the full word addressed by D2(X2,B2).

STH: Bit positions 16 to 31 of general-purpose register R1 are stored in the halfword addressed by D2(X2,B2).

STC: Bit positions 24 to 31 of general-purpose register R1 are stored in the byte addressed by D2(X2,B2).

Instr.	Operand1	Operand2
ST	Bytes 0 to 3 of register R1	Word addressed by D2(X2,B2)
STH	Bytes 2 and 3 of register R1	Halfword addressed by D2(X2,B2)
STC	Byte 3 of register R1	Byte addressed by D2(X2,B2)

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access to operand2 illegal.
Addressing error	X'5C'	STH: D2(X2,B2) not a halfword boundary ST : D2(X2,B2) not a word boundary

Store Clock

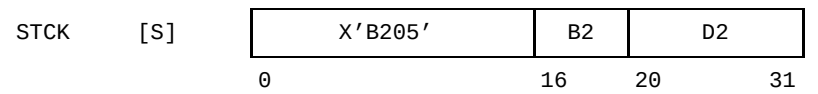
Function

The STCK instruction moves the current value of the clock to a doubleword in main memory.

Assembler format

Name	Operation	Operands	Remarks
	STCK	D2(B2)	D2(B2): doubleword boundary

Machine format



Description

The clock is a 64-bit unsigned binary number in an internal register of the central processing unit. After every microsecond, i.e. every 10^{-6} seconds, this binary number is incremented logically by 4096 (2^{12}). The STCK instruction causes the current value of this binary number to be moved to the doubleword addressed in main memory by D2(B2).

Many central processing units have a clock with a finer resolution. This is indicated by the fact that incrementation is more frequent and the increment is smaller than 4096. In any case, however, every 10^{-6} seconds bit positions 51 of the clock is incremented by 1.

Condition code

0~Zero	Clock time set relative to 1.1.1900, 0:00 o'clock.
1	Not used under BS2000.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand2 illegal.
Addressing error	X'5C'	D2(B2) not a doubleword boundary.

Programming notes

– The clock is useful for calculating the absolute time lapse between two events. To calculate the amount of time by a specific process, you should use the BS2000 macro GEPRT. The date and time of day are provided by the BS2000 macro GDATE.

– Every time the BS2000 operating system is initialized, the clock is set to a value relative to the base date January 1, 1900, 0:00 o'clock, so that the binary value 0 of the clock corresponds to this date. Thus, for example, on January 1, 1987, the clock would have the value $(87*365+21)*24*60*60*10^6*2^{12}$ $=(9C\ 0F\ 80\ D6\ C0\ 00\ 00\ 00)_{16}$.

– By incrementing the clock by 2^{12} after every microsecond, it follows that bit position 31 is incremented by 1 every 1.048576 seconds. Thus, unless a finer resolution is required, the first word of the doubleword stored by STCK is sufficient.

Example

```
Name        Operation   Operands
            .
TIMEBEF     DS          D
TIMEAFT     DS          D
TIMEDIFF    DC          PL8'0.00'
             .
             .
             .
            STCK        TIMEBEF
             .

   < Execution of operation to be measured >

             .
            STCK        TIMEAFT
            LM          0,1,TIMEAFT          Form diff TIMEAFT-TIMEBEF
            SL          1,TIMEBEF+4
            BNL         *+6                  Carry over handling
            BCTR        0,0
            S           0,TIMEBEF
            D           0,=F'40960000'       Scaling of 100th sec.
            CVD         1,TIMEDIFF           Quotient in register 1
             .
```

The example illustrates how to use the STCK instruction to measure time. The clock is read before and after the operation. Then the difference between the two is calculated. Following the S instruction, registers 0 and 1 contain the difference in multiples of 4''096'000.000th seconds as a 64-bit fixed-point number. (The right part of the difference is formed by logical subtraction, the left one by signed subtraction.) The D instruction supplies the time, rounded to 100th seconds, in general-purpose register 1, and the CVD instruction converts this result into packed decimal form.

Store Characters under Mask

Function

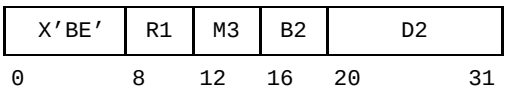
The STCM instruction moves selected bytes of a general-purpose register to a field in main memory.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	STCM	R1,M3,D2(B2)	B’0000’ ≤ M3 ≤ B’1111’

Machine format

STCM [RS]



Description

The 4 bits of the "mask" M3 correspond one-to-one with the 4 bytes of general-purpose register R1 (from left to right in both the mask and the register). Those bytes in R1 which lie opposite ones in the mask are moved to consecutive bytes in the main memory area addressed by D2(B2).

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X’48’	Write access of operand2 illegal.

Programming notes

– The length in bytes of the main memory area is equal to the number of ones in the mask.

– When a mask consisting entirely of ones is used (B'1111'), the STCM instruction has the same effect as the ST instruction, except that the main memory field does not have to be aligned on a word boundary.

Example

```
Name    Operation   Operands
        .
        STCM        15,B'0101',FIELD    Store second and fourth
*                                       bytes of register 15
*                                       in the main memory bytes
*                                       FIELD and FIELD+1
        .
```

Store Multiple

Function

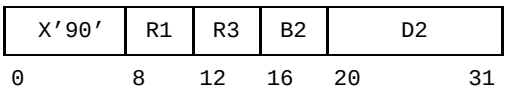
The STM instruction stores the contents of (up to 16) consecutive general-purpose registers in consecutive words in main memory.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	STM	R1,R3,D2(B2)	D2(B2): word boundary

Machine format

STM [RS]



Description

The contents of consecutive general-purpose registers, beginning with R1 and ending with R3, are moved to consecutive words in main memory. The first word is addressed by D2(B2).
If R1 > R3, storage takes place from general-purpose register R1 to general-purpose register 0 to register R3. If R1=R3, only one general-purpose register is stored.

Instr.	Operand1	Operand2
STM	Contents of registers R1 to R3	Word sequence addressed by D2(B2) No of words =R3-R1+1, if R3≥R1 =R3-R1+17, if R3<R1

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	Write access of operand2 illegal. D2(B2) not a word boundary.

Example

```
Name    Operation   Operands
        .
        STM         15,0,SAVE       The contents of general-purpose
*                                   registers 15 and 0 are stored
*                                   in the two consecutive words
*                                   SAVE and SAVE+4.
        .
```

Supervisor Call

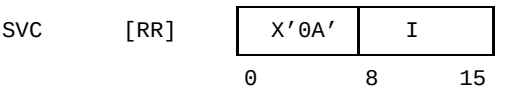
Function

The SVC instruction calls a (privileged) routine of the operating system.
The condition code is left unchanged

Assembler format

Name	Operation	Operands	Remarks
	SVC	I	X'00' ≤ I ≤ X'FF'

Machine format



Description

The SVC instruction activates a supervisor (Control System) call and also moves the contents of its I field to a privileged register of the central processing unit. The further execution of the instruction takes place in the privileged status of the central processing units, and is therefore described here.

Condition code

Stays the same.

Program interrupts

None.

Programming notes

The SVC instruction is part of the expansion of many BS2000 macros, e.g. the WROUT macro. It causes these macros to be transferred to the BS2000 system routines. By incorporating SVC into macros, the user is relieved of having to memorize the corresponding SVC code, among other things. This SVC instruction also facilitates portability, e.g. when converting to a new BS2000 version.

Test under Mask

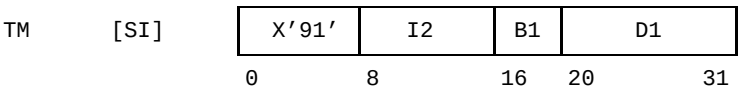
Function

The TM instruction tests selected bit positions of a byte in main memory.
The condition code is set in accordance with the test result.

Assembler format

Name	Operation	Operands	Remarks
	TM	D1(B1),I2	B'00000000' ≤ I2 ≤ B'11111111'

Machine format



Description

The direct operand I2 of the instruction is used as a mask to test selected bits of the byte addressed in main memory by D1(B1). The 8 bits of the mask correspond one-to-one with the 8 bits of the byte to be tested (from left to right in both operands). Each bit value of 1 in the mask selects the corresponding bit in the main memory byte and tests it. If all tested bits are =0, the condition code is set to 0~Zero; if they are all =1, the condition code is set to 3~Ones; and if bits of value 0 and bits of value 1 occur among the tested bits, the condition code is set to 1~Mixed.

Condition code

0~Zeroes	All tested bits are =0.
1~Mixed	The tested bits are neither all =0 nor all =1.
2	Not used.
3~Ones	all tested bits are =1.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand1 illegal.

Programming notes

The assembler [1] has "extended mnemonic operation codes" at its disposal for branch instructions following the T instruction:

- BZ or BRZ (Branch when Zeroes) for querying for pure 0_2 bit pattern
- BO or BRO (Branch when Ones) for querying for pure 1_2 bit pattern
- BM or BRM (Branch when Mixed) for querying for mixed 0_2-1_2 bit pattern

A complete list of all extended mnemonic operation codes can be found in the Appendix.

Examples

The sample instructions below set the following condition codes:

```
Name        Operation   Operands          CC      Query by e.g.:
            .
TESTBYTE    DC          X'87'
            .
            .
            .
Example1    TM          TESTBYTE,X'87'     3        BO, BRO
Example2    TM          TESTBYTE,X'70'     0        BZ, BRZ
Example3    TM          TESTBYTE,X'88'     1        BM, BRM
            .
```

Translate

Function

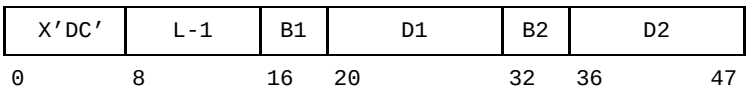
The TR instruction sets the bytes of a target field in accordance with a conversion table.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	TR	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine format

TR [SS]



Description

D1(B1) addresses the "target field" (L bytes long), and D2(B2) addresses the "conversion table". Prior to instruction execution, the target field contains the bytes to be converted; afterwards, it contains the converted bytes.

The target field is processed byte-by-byte from left to right. Each byte of the target field ("argument byte") is added individually to the start address of the conversion table. The sum addresses a byte ("function byte") in the conversion table. The function byte then replaces the original argument byte in the target field. The instruction terminates when all argument bytes have been replaced by function bytes. The addition of each argument byte to the start address of the conversion table takes place logically, with the argument byte being interpreted as an unsigned 8-bit number; the sum is either 24 or 31 bits long, depending on the addressing mode used.

The conversion table is not changed unless it overlaps with the target field. In case of overlap, no condition code is set, but earlier byte conversions are changed by subsequent ones, among other things.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.

Programming notes

– The TR instruction can be used to convert input data from one code to another, e.g. from ASCII to EBCDIC or from lowercase letters to uppercase (see example 1).

– The TR instruction can also be used to put the bytes of a target field into a different sequence. This is done by storing a "pattern" in the target field and using the source field, with the bytes to be put into a new sequence, as a "conversion table". Byte *i* in this pattern, (*i*=0,1,...,L-1) must contain the (binary) place number of that byte in the source field which is to be put on byte *i* in the target field. If, for example, a 3-byte target field is to be "cyclically permuted" (e.g. the sequence a-b-c is to be changed to c-a-b), byte 0 of the target field (i.e. the pattern) must contain the value X'02' (=place number of the source byte 'c'), byte 2 must contain the value X'00' and byte 2 the value X'01' (for further information see example 2).

– The conversion table is as long as the value of the largest argument byte plus 1. For safety's sake, the maximum length of 256 bytes is usually taken for every conversion table, and all bytes in the table are also defined with this length. Only if you are sure that the value of the largest argument byte is less than $(FF)_{16}$, or that individual argument bytes values will not occur, should you depart from this convention.

Examples

Example 1

The following TR instruction can be used to convert a character string CHARFLD from uppercase to lowercase:

```
Name        Operation   Operands

            .
CONVTB      DS          0C
            ORG         CONVTB+'a'
            DC          'ABCDEFGHI'
            ORG         CONVTB+'j'
            DC          'JKLMNOPQR'
            ORG         CONVTB+'s'
            DC          'STUVWXYZ'
            ORG
            .
            .
            .
            TR          CHARFLD,CONVTB
            .
```

Example 2

Below is a method of moving a character field ORGFIELD to a character field INVFIELD and at the same time "inverting" its contents, i.e. making the last byte in ORGFIELD the first byte in INVFIELD, the next-to-last byte in ORGFIELD the second byte in INVFIELD, and so on:

```
Name      Operation   Operands
          .
          LA          0,L'ORGFIELD          Pad INVFIELD with
          LA          1,0                   place numbers
MUSTERN   BCTR        0,0                   L-1, L-2,..., 0
          STC         0,INVFIELD(1)
          LA          1,1(1)
          LTR         0,0
          BNE         MUSTERN
          TR          INVFIELD(L'ORGFIELD),ORGFIELD
          .
```

These instructions will convert, for example,

```
ORGFIELD | DC | '0123456789'
```

into the inverted field

```
INVFIELD | DC | '9876543210'
```

Translate and Test

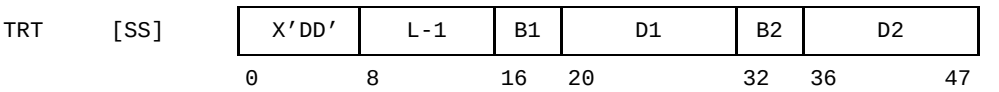
Function

The TRT instruction tests the bytes of a target field against a conversion table. The condition code is set in accordance with the results of the test.

Assembler format

Name	Operation	Operands	Remarks
	TRT	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine format



Description

D1(B1) addresses the "target field" (L bytes long); D2(B2) addresses the "conversion table".

The target field is processed byte-by-byte from left to right. Each byte of the target field ("argument byte") is added individually to the start address of the conversion table. The sum addresses a byte in the conversion table ("function byte"). If this function byte is $\neq 00_{16}$, the instruction is terminated; otherwise, the next argument byte is processed.

If all the function bytes in the conversion table are $=00_{16}$, the instruction is terminated with the condition code 0~Zero.

The first function byte that is $\neq 00_{16}$ terminates the instruction. The condition code is set 2~Plus if the associated argument byte was the last byte in the target field; otherwise, the condition code is set to 1~Minus. The address of the associated argument byte is entered in general-purpose register 1, and the non-zero function byte is entered in general-purpose register 2.

The addition of each argument byte to the start address of the conversion table proceeds logically, with the argument byte being interpreted as an unsigned 8-bit number; the sum is either 24 or 31 bits long, depending on the addressing mode used.

Neither the conversion table nor the target field is changed, not even in the case of overlapping (which is permitted).

General-purpose registers 1 and 2 are only changed when a function byte $\neq 00_{16}$ is encountered.
In 24-bit addressing mode, the 24-bit argument address is entered in bit positions 8 to 31 of general-purpose register 1, and bit positions 0 to 7 are left unchanged; in 31-bit addressing mode, the 31-bit argument byte address is entered in bit positions 1 to 31 of register 1, and bit position 0 is set to 0.
The value of the first non-zero function byte is entered in bit positions 24 to 31 of general-purpose register 2; bit positions 0 to 23 are left unchanged.

Condition code

0~Zero	All function bytes are $=00_{16}$.
1~Minus	A function byte not equal to 00_{16} was encountered before the last argument byte in the target field was processed.
2~Plus	The argument byte belonging to the last byte in the target field was $\neq 00_{16}$.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand1 or operand2 illegal.

Programming notes

- The conversion table is as long as the value of the largest processed argument byte, plus 1.
- Unlike the TR instruction, with the TRT instruction the target table is left unchanged.
- The TRT instruction uses general-purpose registers 1 and 2, although they are not specified in the instruction.
- Since general-purpose registers 1 and 2 are not always changed, and even when they are they are not completely replaced, we recommend setting these two registers explicitly *prior to* TRT, e.g. to the address of the first byte after the memory space to be processed.
- The TRT instruction can be used to check a target field for characters which have a special meaning, e.g. which are illegal. This is done by setting all of those function bytes in the conversion table that are identical to these characters to a value $\neq 00_{16}$, and setting all other bytes to the value $=00_{16}$.

Example

The target field DFIELD is to be checked for occurrences of "+" or "-". This can be done by declaring the following data fields:

```
Name      Operation   Operands
          .
CONVTB    DC          256X'00'              Function byte 00 for all
          ORG         CONVTB+'+'            other characters
          DC          X'01'                 Function byte 01 for +
          ORG         CONVTB+'-'
          DC          X'02'                 Function byte 02 for -
          ORG
          .
```

and entering the following instructions:

```
          .
          SR          1,1             Erase registers 1 and 2
          SR          2,2             see Programming Notes
          TRT         DFIELD,CONVTB
          BE          NOSIGN          No + or - in DFIELD
          BH          TRAILING        + or -  in final byte
          BL          EMBEDDED        + or -, but not in final byte
            .
```

In the case of TRAILING and EMBEDDED, general-purpose register 1 contains the address of the first "+" or "-" in DFIELD (either 24 bits or 31 bits long, depending on the addressing mode used), and general-purpose register 2 contains the associated function byte in its lowest-order byte, i.e. in this case either 01_{16} or 02_{16}. Note the concluding ORG instruction: it prevents any subsequent data declaration from extending into CONVTB.

Test and Set

Function

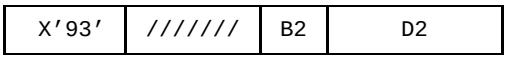
The TS instruction sets the condition code in accordance with the value of the highest-order of a main memory byte. It then overwrites this byte with $(FF)_{16}$. This instruction cannot be interrupted while it is being executed.

Assembler format

Name	Operation	Operands	Remarks
	TS	D2(B2)	

Machine format

TS [S]



Description

The highest-order bit in the main memory byte addressed by D2(B2) is tested. If it is =0, the condition code is set to 0~Zero; if not, the condition code is set to 1~Not Zero. Then all bits in the byte are set to 1, i.e. the byte is overwritten with FF_{16}. Bit positions 8 to 15 of the instruction are ignored.

The feature peculiar to the TS instruction is that in the time that passes between testing the highest-order bit of the byte addressed by the instruction and completion of overwriting with $(FF)_{16}$, no other central processing unit and no channel has read or write access to the byte in question. For this purpose, "serialization" takes place in the hardware before and after the instruction. During serialization, all outstanding memory access operations are processed. This mechanism predestines the TS instruction for synchronization problems in multiprocessor applications.

Condition code

0~Zero	Highest-order bit of D2(B2) was =0.
1~Not Zero	Highest-order bit of D2(B2) was =1.
2	Not used.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X’48’	Read/write access of operand2 illegal.

Programming notes

The TS instruction is less powerful than the CS and CDS instructions, and has been retained in the instruction set only for reasons of compatibility. For this reason, we refer to the description of the CS and CDS instructions and Appendix 7.6 for further information.

Unpack

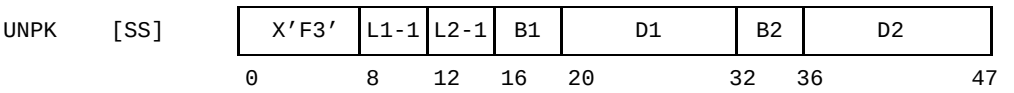
Function

The UNPK instruction turns a (packed) decimal number in the source field into an unpacked decimal number in the receive field.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	UNPK	D1(L1,B1),D2(L2,B2)	1 ≤ L1,L2 ≤ 16

Machine format



Description

D1(L1,B1) addresses the receive field, and D2(L2,B2) the source field (1 ≤ L1,L2 ≤ 16). The (packed) decimal number contained in the source field is moved to the receive field, where it is converted to unpacked (zoned) format.

The source field is *not* checked to see whether it really does contain a correct, packed decimal number; instead, it is treated as though it does contain one.

Both operands are processed byte-by-byte from right to left. First, the two halfbytes of the lowest-order byte in the source field are moved in reverse order to the lowest-order byte in the receive field. Then each additional halfbyte in the source field is moved to the right halfbyte of a byte in the receive field, with each left halfbyte ("zone") being set to F_{16}.

If the source field runs out of space before the receive field (i.e. if 2L2 < L1+1), the left-most (L1-2L2+1) bytes of the receive field will be padded with $(F0)_{16}$; if the receive field is too short to accommodate all halfbytes of the source field (i.e. if L1 < 2L2-1), the leftmost (2L2-L1-1) halfbytes in the source field will be ignored.

The receive and source fields may overlap. In this case, all subsequent byte operations will generally overwrite the result of earlier byte operations of the same instruction. The instruction is executed as though each byte in the receive field is stored the moment the necessary byte in the source field has been read.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand1 or read access of operand2 illegal.

Programming notes

The source field is only changed if it overlaps with the receive field.

Examples

The sample UNPK instructions below yield the following results:

FIELD before	Sample instruction		FIELD after	CC
any	UNPK	FIELD(1),=PL1'-3'	Z'-3'	unchanged
any	UNPK	FIELD(5),=PL2'12'	Z'00012'	unchanged
X'89'	UNPK	FIELD(1),FIELD(1)	X'98'	unchanged
X'23456789'	UNPK	FIELD(4),FIELD(4)	X'F6F6F798'	unchanged

The third example makes use of the fact that the source field (FIELD) is not checked for packed format (X'89' is not a correct, packed decimal number): all that happens is that the two halfbytes of FIELD change places. However, this permutation only takes place in the last (in this case only) byte, as is illustrated by example 4. Example 4 also illustrates a case of overlapping operands in which a subsequent byte operation (in this case the third) changes the result of an earlier byte operation (in this case the second). (The point of this example is to illustrate this case, not to recommend it.)

EXCLUSIVE OR

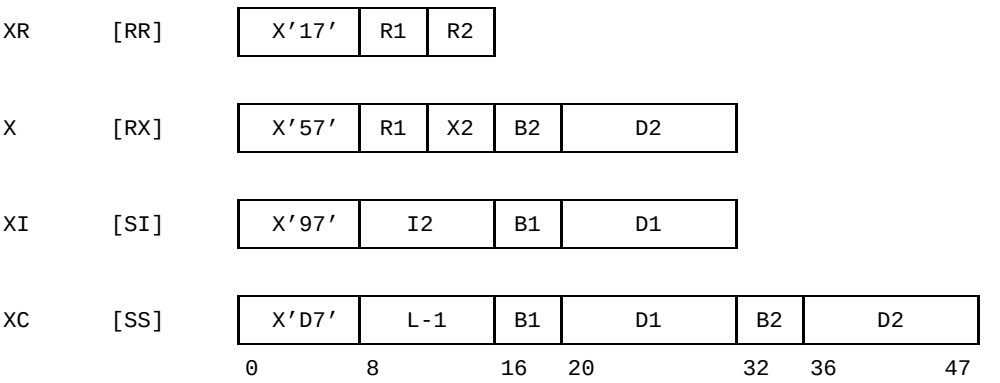
Function

The instructions XR, X, XI and XC cause two operands to be exclusively ORed bit-by-bit.
The condition code is set in accordance with the value of the results.

Assembler formats

Name	Operation	Operands	Remarks
	XR	R1,R2	
	X	R1,D2(X2,B2)	D2(X2,B2): word boundary
	XI	D1(B1),I2	X'00' ≤ I2 ≤ X'FF'
	XC	D1(L,B1),D2(B2)	1 ≤ L ≤ 256

Machine formats



Description

The bits in the first operand are changed by the opposing bits in the second operand according to the table below. The result replaces the first operand.

Table of EXCLUSIVE OR conjunctions

Bit value in first operand	Bit value in second operand	Bit value in result
0	0	0
0	1	1
1	0	1
1	1	0

Operands

Instr.	Operand1	Operand2
XR	Contents of register R1	Contents of register R2
X	Contents of register R1	Word addressed by D2(X2,B2)
XI	Byte addressed by D1(B1)	Direct operand I2
XC	Field addressed by D1(B1) with length L bytes	Field addressed by D2(B2) with length L bytes

Condition code

0~Zero result =0
1~Not Zero result ≠0
2 Not used.
3 Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	X: Read access of operand2 illegal. XI: Read/write access of operand1 illegal. XC: Read/write access of operand1 or read access of operand2 illegal.
Addressing error	X'5C'	X: D2(X2,B2) not a word boundary.

Programming notes

- EXCLUSIVE OR instructions invert all bit position in the first operand for which the opposing bit position in the second operand has the value 1, and leaves the remaining bit positions of the first operand unchanged.
- The operands are processed byte-by-byte from left to right.
- With XC, the operands may overlap. However, among other things, this means that subsequent byte operations will change earlier ones.
- If R1=R2 with the XR instruction, general-purpose register R1 and the condition code are set to zero.
- The XC instruction with operand1=operand2 sets all bytes of operand1 to X'00'.
- When using the XI and XC instructions in multiprocessor systems, note the following:
  Memory access operations of the first operand of the XI and XC instructions consist of reading a byte from memory and then writing the changed value into memory. These read and write operations on a single byte are not necessarily consecutive, if another processor or another application (or an input/output channel program) attempts to modify the memory location in question. A safe way of updating a shared word in memory is described in Appendix 7.6 and in the programming notes for the CS and CDS instructions.

Example

```
Name    Operation   Operands
        .
        XC          A,B                 This famous trick switches the
        XC          B,A                 main memory fields A and B
        XC          A,B                 without an auxiliary field.
        .
```

Incidentally, this pretty algorithm has an exception: when areas A and B overlap (or are identical), the part they share is padded with binary zeros instead of being left unchanged.
In the same way (using three XR instructions), it is possible to switch the contents of two (different) general-purpose registers; it is not possible, however, to switch a word in main memory with a general-purpose register because there is no X instruction whose *first* operand addresses a word in main memory.

4 Decimal instructions

Overview

Decimal instructions are used for

a) adding, subtraction, multiplying, dividing and comparing two decimal numbers (AP, SP, MP, DP, CP),

b) moving and/or rounding a decimal numbers (SRP),

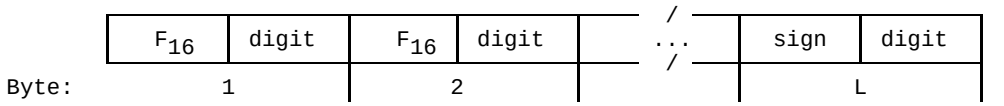
c) editing one or more consecutive decimal numbers (ED, EDMK).

The decimal numbers which are processed by decimal instructions are signed base 10 integers with up to 31 digits. With all decimal instructions, the numbers are taken from or created in main memory. There are no register operands, although some decimal arithmetic instructions use general-purpose registers for special condition codes.

With all decimal instructions, the length of a decimal number (in bytes) is specified in the instruction itself, and does not form part of the decimal number. (The assembler [1] considerably simplifies length computation.)

The decimal numbers processed by decimal instructions are stored in main memory in one of two formats: packed format or unpacked format. Unpacked (also known in other contexts as "zoned" format) is the format used following input (e.g. from the keyboard) or for output (e.g. to printer); packed format is the format used for the arithmetical handling or comparison of decimal numbers. (In Assembler, decimal numbers can be defined in either format by using the constant types Z and P; see below.) Both formats are explained in greater detail below.

Unpacked format



When decimal numbers are in unpacked (or "zoned") format, each decimal position is represented in one byte: the ones position in the final Lth byte, the tens position in the next-to-last (L-1)th byte, the hundreds position in the second-to-last (L-2)th byte, and so forth.

The numeric value of each decimal position is represented by means of its hexadecimal equivalent ($d_{10} = d_{16}$, d =0, 1, ... 9); in each case it forms the right halfbyte. The left halfbyte (the "zone") of the first L-1 bytes is a constant F_{16}; the left halfbyte of the final Lth byte contains the sign.

A positive sign is indicated by the hexadecimal values A_{16} or C_{16} or E_{16} or F_{16}; a negative sign is indicated by the hexadecimal values B_{16} or D_{16}.

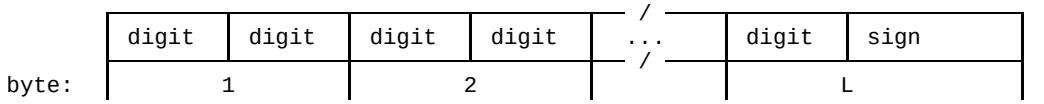
An unpacked format up to L bytes in length can accommodate one decimal number with up to L digits; the leftmost bytes contain the value $(F0)_{16}$ if the decimal number has fewer than L decimal positions. The maximum length of a decimal number is unpacked format is 16 bytes, so that up to 16-digit decimal numbers can be represented in this format.

Examples:

Decimal number	Unpacked format	
+12	F1 A2 or F1 C2 or F1 E2 or F1 F2	(2 bytes)
-5	B5 or D5	(1 byte)

In Assembler [1], the unpacked format of a decimal number is created with the Z-type constant, with the hexadecimal values C_{16} (for plus) and D_{16} (for minus) being used to represent the sign: e.g. Z'12' and Z'+12' each define two bytes with the contents $(F1\ C2)_{16}$ and Z'-5' defines one byte with the contents $(D5)_{16}$. The computation of the length of a decimal number can be left to the assembler (as in the examples just shown) or it can be entered explicitly: ZL3'12' defines 3 bytes with the contents $(F0\ F1\ C2)_{16}$. At the definition stage, the programmer can also add a decimal point, which, however, is not taken into account in the memory representation or in decimal instructions.

Packed format



When decimal numbers are given in packed format, each decimal position is represented in a halfbyte. The ones position is stored in the left halfbyte of the final Lth byte, the tens position in the right halfbyte of the next-to-last (L-1)th byte, the hundreds position in the same byte but in the left halfbyte, and so forth.

The numeric value of each decimal position is represented by its hexadecimal equivalent ($d_{10} = d_{16}$, d =0, 1, ... 9). The sign of the decimal number is represented in the right halfbyte of the last (lower-order) byte. A positive sign is determined by the hexadecimal values A_{16} or C_{16} or E_{16} or F_{16}, and a negative sign by the hexadecimal values B_{16} or D_{16}.

A packed format which is L bytes long can accommodate a decimal number of up to 2L-1 digits, where the upper half-bytes contain $=0_{16}$ when the decimal number includes fewer than 2L-1 significant decimal positions. The maximum length of the packed format of a decimal number is 16 bytes, so that decimal numbers up to 31 digits long can be represented in this format. The absolute value range W of packed decimal numbers is therefore $0 \leq W \leq 10^{31}-1$.

Examples:

Decimal number	Packed format
+12	01 2A or 01 2C or 01 2E or 01 2F (2 bytes)
-5	5B or 5D (1 byte)

In Assembler, the packed format of a decimal number is created by the P-type constant, with the sign being represented by the hexadecimal values C_{16} (for plus) and D_{16} (for minus): e.g. P'12' or P'+12' each define two bytes with the contents $(01\ 2C)_{16}$ and P'-5' defines one byte with the contents $(5D)_{16}$. As with the unpacked format, the assembler determines the length implicitly whenever an explicit length specification is omitted.

Table of sign codes

Code	Sign
A_{16}	positive
B_{16}	negative
C_{16}	positive
D_{16}	negative
E_{16}	positive
F_{16}	positive

Add Decimal

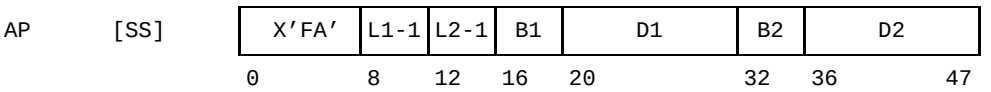
Function

The AP instruction adds two packed decimal numbers. The sum replaces the first addend.
The condition code is set in accordance with the value of the sum.

Assembler format

Name	Operation	Operands	Remarks
	AP	D1(L1,B1),D2(L2,B2)	1 ≤ L1,L2 ≤ 16

Machine format



Description

The packed decimal number in the field of the second operand (D2(L2,B2)) is added to the packed decimal number in the field of the first operand (D1(L1,B1)), with the sign being taken into account; the packed sum then replaces the first operand. The first operand and the sum are both L1 bytes long, and the second operand is L2 bytes long, where 1 ≤ L1, L2 ≤ 16.

Both operands are checked for correct packed format; in case of error, a program interrupt occurs due to a data error.

A decimal overflow takes place when the sum has more significant decimal positions than will fit into the field of the first operand; in this case, the instruction terminates normally, but only the 2L1-1 lowest-order decimal positions are stored and the highest-order decimal positions are lost. The condition code is set to 3~Overflow. If the bit for decimal overflow in the program mask is set to 1 (default value in BS2000), a program interrupt will also occur due to a decimal overflow.

A genuine sum of =0 always has a positive sign (C_{16}); however, a 0 sum which results from a decimal overflow may also have a negative sign (D_{16}).

Condition code

0~Zero sum = 0 (with sign C_{16}).
1~Minus sum < 0 (with sign D_{16}).
2~Plus sum > 0 (with sign C_{16}).
3~Overflow Sum too large.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access operand1 or read access of operand2 illegal.
Data error	X'60'	Incorrect format in addend
Decimal overflow	X'74'	Sum too large for first operand.

Programming notes

- Both operands are processed as integers.
- A positive sign in the result is represented by C_{16}, a negative sign by D_{16}.
- The two operands may overlap, but in this case the addresses of their lowest-order bytes must be identical (D1(B1)+L1-1 = D2(B2)+L2-1); otherwise, a program interrupt will occur due to a data error.
- The second operand is only changed if it overlaps with the first operand.
- The operands are processed from right to left.
- If a decimal overflow occurs, the result has the sign of the correct sum.

Examples

The sample AP instructions shown below yield the following results:

DFIELD before	Sample instructions	DFIELD after	CC
PL1'+1'	AP DFIELD,=PL1'-1'	PL1'+0'	0
PL1'+1'	AP DFIELD,=PL16'-2'	PL1'-1'	1
PL1'+1'	AP DFIELD,DFIELD	PL1'+2'	2
PL1'+1'	AP DFIELD,=PL8'-11'	PL1'-0'	3

Note that the decimal overflow in the fourth example is not caused by the excessive length of the second operand, but rather because the sum (-10) does not fit into the first operand. With decimal overflow, it may happen (as in this example) that a resultant zero is given a negative sign.

Compare Decimal

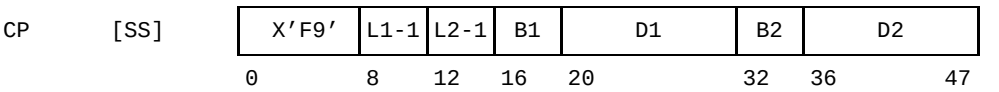
Function

The CP instruction compares two packed decimal numbers.
The condition code is set in accordance with the comparison results.

Assembler format

Name	Operation	Operands	Remarks
	CP	D1(L1,B1),D2(L2,B2)	1 ≤ L1,L2 ≤ 16

Machine format



Description

The packed decimal number in the field of the first operand (D1(L1,B1)) is compared with the packed decimal number in the field of the second operand (D2(L2,B2)). The signs are taken into account, and the condition code is set in accordance with the comparison result.

Both operands are checked for correct packed format; in case of error, a program interrupt occurs due to a data error.

Decimal overflow cannot occur.

Condition code

0~Equal	operand1 = operand2
1~Low	operand1 < operand2
2~High	operand1 > operand2
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read access of operand1 or operand2 illegal.
Data error	X'60'	Incorrect format in operand.

Programming notes

- Both operands are processed as integers.
- Both operands are always left unchanged.
- The operands are processed from right to left.
- When different sign codes with identical meanings appear in a compare operation (e.g. C_{16} and F_{16}), they are always treated in accordance with their meaning.
- When negative zero is compared to positive zero, the result is the condition code 0~Equal.
- The two operands may overlap, but in this case the addresses of their lowest-order bytes must be identical (D1(B1)+L1-1=D2(B2)+L2-1); otherwise, a program interrupt will occur due to a data error.

Examples

The sample CP instructions shown below yield the following results:

CFIELD	Sample instructions	CC
PL1'+0'	CP CFIELD,=PL16'-0'	0
PL1'+1'	CP CFIELD,=PL1'2'	1
PL1'+1'	CP CFIELD,=PL16'-2'	2

In the first example, a positive zero is compared with a negative zero, which also happens to be overlength: nevertheless, the comparison results are "equal". Similarly a comparison of X'1B' with X'001D' would set the condition code to 0~Equal.

Divide Decimal

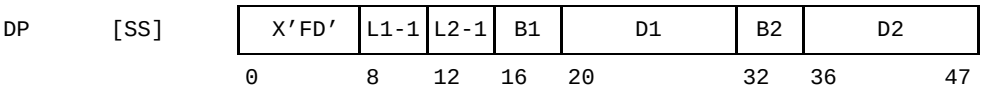
Function

The DP instruction divides two packed decimal numbers. The quotient and the remainder replace the dividend.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
*	DP	D1(L1,B1),D2(L2,B2)	L2 < L1 ≤ 16 and 1 ≤ L2 ≤ min(8, L1-1)

Machine format



Description

The DP instruction performs signed division of the dividend (D1(L1,B1)) by the divisor (D2(L2,B2)), and creates the (integer part of the) quotient as well as the division remainder as packed decimal numbers in the field of the dividend.

The length of the dividend (L1) must be greater than that of the divisor (L2), otherwise a program interrupt will occur due to an addressing error (L2 < L1 ≤ 16). At least the first decimal position in the dividend must be $=0_{16}$, otherwise a program interrupt will occur due to a division error.

The length of the divisor (L2) must be less than that of the dividend (L1), and must not be greater than 8 bytes, otherwise a program interrupt will occur due to an addressing error. The divisor can therefore include a maximum of 15 decimal positions (L2 ≤ Min (L1-1,8)).

The resultant quotient is L1-L2 bytes long, and is stored as a packed decimal number (integer) in the leftmost L1-L2 bytes of the field of the dividend; the resultant remainder is L2 bytes long, and is stored in the rightmost L2 bytes of the field of the dividend. The quotient and the remainder thus completely replace the dividend.

Both operands must represent valid packed decimal numbers, otherwise a program interrupt will occur due to a data error.

The sign of the quotient is formed according to the usual algebraic rules, even if the dividend is =0; the division remainder always has the sign of the dividend, even if the remainder is =0. A positive sign is represented as C_{16}, a negative sign as D_{16}.

If the divisor is =0 or the quotient is longer than L1-L2 bytes, a program interrupt will occur due to a data error (i.e. not due to a decimal overflow). This also applies to the case where both the dividend *and* the divisor are =0. In the case of division error, both the initial operands are left unchanged.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 and read access of operand2 illegal.
Addressing error	X'5C	L1 or L2 incorrect.
Data error	X'60'	Incorrect format in operand.
Division error	X'68'	Divisor =0 or quotient too large.

Programming notes

- All operands (dividend, divisor, quotient and remainder) are interpreted as integers.
- The quotient can be at most 15 bytes long, i.e. it can include a maximum of 29 decimal positions.
- Dividend and divisor may overlap, but in this case the addresses of their lowest-order bytes must be identical (D1(B1)+L1-1=D2(B2)+L2-1), otherwise a program interrupt will occur due to a data error. Moreover, L1 must be greater than L2, i.e. D1(B1)≠D2(B2).
- The dividend must have at least one leading halfbyte with the value 0_{16}. This is a necessary but not sufficient condition for preventing a division error.
- The following instruction sequence is equivalent to a division error check running within DP:

```
Name    Operation   Operands
        MVO         TEMP(L'DIVISOR+1),DIVISOR
        CLC         TEMP(L'DIVISOR),DIVIDEND
        BNH         DIVERROR
```

This instruction sequence may be executed prior to a DP instruction to ensure that no program interrupt due to a division error will occur. It requires a temporary field TEMP which is (L'DIVISOR + 1) long.

Examples

Following execution of

```
Name    Operation   Operands
        .
        DP          DFIELD,DIVISOR
        .
```

the examples below yield the following results:

DFIELD before Dividend	DIVISOR Divisor	DFIELD after Quotient	DFIELD after Remainder
PL5'1001'	PL2'10'	PL3'+100'	PL2'+1'
PL5'1001'	PL2'-10'	PL3'-100'	PL2'+1'
PL5'-1001'	PL2'-10'	PL3'+100'	PL2'-1'
PL5'1000'	PL2'-10'	PL3'-100'	PL2'+0'
PL3'-1000'	P'1'	unchanged, dividend error	
PL8'-1000'	PL8'10'	unchanged, addressing error	

In the next-to-last example a division error occurs because the quotient (P'-1000') requires 3 bytes for storage purpose but only 2 bytes are available; the third byte is reserved for the remainder.

In the final example, an addressing error occurs because the condition L2=8 ≤ Min(L1-1,8)=Min(7,8)=7 is not satisfied.

Edit

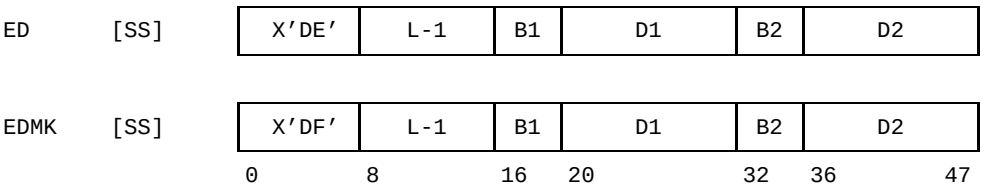
Function

The instructions ED (Edit) and EDMK (Edit and Mark) edit one or more packed decimal numbers into printable form. The EDMK instruction also enters the address of the most significant decimal position into general-purpose register 1.
The condition code is set in accordance with the most recently edited decimal number.

Assembler formats

Name	Operation	Operands	Remarks
	ED EDMK	D1(L,B1),D2(B2) D1(L,B1),D2(B2)	1 ≤ L ≤ 256 1 ≤ L ≤ 256

Machine formats



Description

The instructions ED and EDMK edit one or more consecutive packed decimal numbers into printable form. The editing takes place with the aid of masks.

The EDMK instruction does the same as the ED instruction except that it also "marks" the most significant decimal position in the receive field by entering its address in general-purpose register 1.

Operand1 (addressed by D1(B1)) has two purposes: before the instruction is executed it contains the editing mask, and following execution it contains the edited result. Depending on the purpose served, operand1 is therefore referred to either as "mask" or as "receive field". Any character may be included in the mask, but three characters have special meaning as control characters. These are the codes X'20' (digit selector), X'21' (significance starter) and X'22' (field separator).

Operand2 (addressed by D2(B2)) is the "source field". It must contain one or more consecutive packed decimal numbers.

If the two operands overlap, the results will be unpredictable.

Both operands are processed from left to right. The mask (and the receive field) are processed byte-by-byte, the source field halfbyte-by-halfbyte.

Every character in the mask is replaced during instruction execution either by an unpacked decimal digit of the source field or by the filler character (see below), or it is left unchanged. Which of these options is actually taken depends on

– the mask character,

– the switch setting of the significance indicator,

– the value of the decimal digit lying opposite the mask character in the source field (=0 or ≠0).

Mask character

There are four kinds of mask character:

Mask character	Coding
Digit selector	X'20'
Significance starter	X'21'
Field separator	X'22'
Text character	any other

The occurrence of a digit selector or a significance starter in the mask causes the next decimal digit in the source field to be read. Depending on its value and on the setting of the significance indicator, the character entered in the receive field in place of the mask character is either this decimal digit (unpacked and supplied with zone F_{16}) or the filler character.

The field separator indicates the beginning of a new decimal number in the source field, whenever an ED or EDMK instruction is to edit two or more decimal numbers. The field separator is always replaced by the filler character, and causes the significance indicator to be set to "off".

All text characters in the mask either remain unchanged or are replaced by the filler character, depending on whether the significance indicator is set to "on" or "off".

Filler character

The first character in the mask, i.e. the byte at the address D1(B1), is used as a filler character. Any character may be selected as a filler character, even the digit selector, the significance starter or the field separator.

The filler character is entered in the receive field depending on the significance indicator setting, the mask character and the value of the opposing digit in the source field. For further details, consult the tabular summary below.

The filler character is buffered at the start of instruction execution, and remains available from the buffer for the entire time that the instruction is being executed, even if it itself is replaced in the receive field. Only after the filler character has been buffered does interpretation of the mask character begin, starting with the filler character itself.

The filler character itself is only replaced if it is a digit selector or significance starter, and if it lies opposite a decimal digit $\neq 0$.

Source field digits

Each time a digit selector or significance starter is encountered in the mask, the next decimal digit in the source field is read and the significance indicator is checked. This determines whether the decimal digits are stored in the right halfbyte of the receive field and the zone F_{16} is entered in the left halfbyte, or whether the decimal digits are skipped and the filler character is stored in the receive field.

The source field is processed one halfbyte at a time, from left to right. All left halfbytes must contain decimal digits, i.e. hexadecimal digits $\leq 9_{16}$; otherwise, a program interrupt will occur due to a data error. Each time a left halfbyte is read, the right halfbyte is checked to see whether it is a sign, i.e. contains a hexadecimal digit $\geq A_{16}$. If so, the system makes sure (after switching the significance indicator, if necessary) that with the next digit selector or significance starter the next adjoining left halfbyte is read; otherwise, the right halfbyte remains available for this purpose.

Significance indicator

The significance indicator is an internal toggle switch. When it is switched to the "on" position, "significance" applies: i.e. digit selectors and significance starters in the mask are replaced by their opposing decimal digits and text characters are left unchanged. If the significance indicator is switched to the "off" position, "nonsignificance" applies: i.e. digit selectors, significance starters and text characters are replaced by the filler character.

The position of the significance indicator also indicates whether the decimal number to be edited contains a minus sign or a plus sign. It thus determines, among other things, the condition code of the ED and EDMK instructions.

The significance indicator is switched to the "off" position

– at the start of instruction execution or

– when a field separator is encountered or

– when a decimal digit in a left halfbyte of the source field is followed by a plus sign (A_{16}, C_{16}, E_{16} or F_{16}) in the associated right halfbyte.

The significance indicator is switched "on" whenever it is in the "off" position and

– a significance starter lies opposite a source field digit which is not followed by a plus sign or

– a digit selector lies opposite a source field digit which is ≠0 and is not followed by a plus sign.

In all other cases the significance indicator is left unchanged (for further information see the tabular summary below).

Receive field characters

The editing results of the ED and EDMK instructions are stored in the receive field and replace the mask: they have the length of the mask (L bytes), and consist of text characters, filler characters and zoned source field digits.

A text character in the mask either remains unchanged or it is replaced by the filler character, depending on whether the significance indicator is in the "on" or "off" position when the text character occurs.

A digit selector or significance starter in the mask is replaced by the filler character when the significance indicator is in the "off" position and the opposing source field digit is =0. In contrast, a digit selector or significance starter in the mask is replaced by the opposing zoned source field digit when this digit is ≠0 or the significance indicator is in the "on" position.

Summary

The table below summarizes the functions of the individual mask characters. The leftmost 4 columns show the 4 possible condition constellations; the rightmost 2 columns show the associated result characters in the receive field and the setting of the significance indicator.

Effect of ED and EDMK mask characters					
Conditions				Result	
Mask character	Significance indicator before	Source field digit	followed by plus sign ?	Receive field character	Significance indicator after
Digit selector (X'20')	off off off on on	0 1...9 1...9 0...9 0...9	irrelevant no yes no yes	filler ch SF digit SF digit SF digit SF digit	off on off on off
Significance starter (X'21')	off off off off on on	0 0 1...9 1...9 0...9 0...9	no yes no yes no yes	filler ch filler ch SF digit SF digit SF digit SF digit	on off on off on off
Field separator (X'22')	irrelevant	irrelevant	irrelevant	filler character	off
Text character (any other)	off on	irrelevant irrelevant	irrelevant irrelevant	filler character text character	off on

Condition code

0~Zero — All processed digits in the most recently edited decimal number were $=0_{16}$ or neither a significance starter nor a digit selector occurred in the mask for the most recently edited decimal number, or the last character in the mask was a field separator.

1~Minus — The most recently edited decimal number was non-zero and the significance indicator was most recently in the "on" position.

2~Plus — The most recently edited decimal number was non-zero and the significance indicator was most recently in the "off" position.

3 — Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.
Data error	X'60'	A left halfbyte in the source field does not contain a decimal digit.

Additional marking with EDMK

The EDMK instruction is identical to the ED instruction with regard to the execution described above and to its result as well as the resulting condition code. However, EDMK also stores, in general-purpose register 1, the highest-order non-zero decimal digit in the receive field, provided the significance indicator was previously set to "on" during the store operation. When two or more decimal numbers are edited with an EDMK instruction, this applies to the final decimal number for which significance has been switched on in this way. If significance is not switched on at all, or is switched on using the significance starter, general-purpose register 1 is left unchanged.

In 24-bit addressing mode, the address is entered in bit positions 8 to 31 of general-purpose register 1; in 31-bit addressing mode it is entered in positions 1 to 31. Bit positions 0 to 7 (24-bit mode) or bit position 0 (31-bit mode) are left unchanged.

Programming notes

- The ED and EDMK instructions are provided for those cases where edited decimal numbers are to be given additional characters such as thousands commas or currency signs, or where leading zeros are to be replaced by filler characters, e.g. by blanks or "protective asterisks" (see examples).
- The EDMK instruction simplifies the insertion of a sign immediately in front of the highest-order non-zero decimal digit of the final edited decimal number. If a significance starter was used in the mask to edit this number, it is advisable, before calling EDMK, to load its address into general-purpose register 1, having previously incremented it by 1. Then general-purpose register 1 will contain the address of the highest-value zoned decimal digit, whether significance was switched on by the significance starter or by the highest-order non-zero decimal digit. Hence, when reduced by 1, register 1 will point to the position of the floating sign (for further information see also example 3).
- With certain data constellations, the EDMK instruction changes general purpose register 1 even though it is not specified in the instruction.

– EDMK does not always change general-purpose register 1, and even when it does change the register it does not fully replace it. It is therefore advisable in all cases (not just in the above case) to set register 1 to a meaningful value *before* issuing EDMK.

Examples

```
Name        Operation   Operands
            MVC         DFIELD1,MASKE
Example1    ED          DFIELD1,=P'123456789'
*                                   DFIELD1 after: 1'234.567,89
*                                   CC: 2~Plus
MASKE       DC          C' '            Filler character
            DC          X'20'           Digit selector
            DC          C''''           Apostrophe
            DC          3X'20'          3 digit selectors
            DC          C'.'            Point
            DC          X'202120'       Digit selector,
            DC          X'202120'       significance starter,
*                                       digit selector
            DC          C','            Comma
            DC          2X'20'          2 digit selectors
            .
Example2    ED          DFIELD2(11),SFIELD
*                                   DFIELD2 after: ***1002***3
*                                   CC: 1~Minus
DFIELD2     DC          C'*'            Filler character
            DC          2X'202120'      First 2 masks
            DC          X'22'           Field separator
            DC          X'202120'       Third mask
SFIELD      DC          PL2'-1,-2,-3'   3 numbers to be edited
            .
Example3    MVC         DFIELD3(7),=X'402021206B2020'
*                                   X'40'=blank, X'6B'=point
            LA          1,DFIELD3+3     Register 1 to
*                                       Significance starter +1
            EDMK        DFIELD3(7),ACCOUNT ACCOUNT = 3 bytes long
            BE          READY           Packed decimal number
            BCTR        1,0
            MVI         0(1),'+'
            BH          READY
            MVI         0(1),'-'
READY       EQU         *
```

If, in example 1, PL5'-1' is specified instead of P'123456789' as the number to be edited, this would produce the character string ␣␣␣␣␣␣␣␣␣0,01.

Example 2 illustrates the effect of a minus sign and field separator on the significance indicator. If the first decimal number to be edited is PL2'-1' instead of PL2'+1', the character string ***1**2***3 will be produced in FIELD2 since in this case the significance indicator is switched on by the plus sign.

Example 3 illustrates an instruction sequence for editing a monetary amount which is to be preceding by a "+" sign, a "-" sign or by no sign, depending on whether it is less than, greater than, or equal to 0. Thus, for example, the ACCOUNT value PL3'-.12' is to be edited to ␣␣-0,12 and the value PL3'0' ␣␣␣0,00. Prior to EDMK, the instruction sequence loads general-purpose register 1 "as a precaution" with the address of the character that follows the significance starter (X'21') in the mask. If the first non-zero in ACCOUNT is located to the right of the significance starter, register 1 is left unchanged by EDMK; otherwise, EDMK enters its address in register 1. In both cases, following EDMK register 1 contains the address of the first zoned digit, and hence the contents of register 1 decremented by 1 via BCTR 1,0, point to the decimal position where the "floating" plus sign or minus sign (or no sign in the case of 0) belongs.

Multiply Decimal

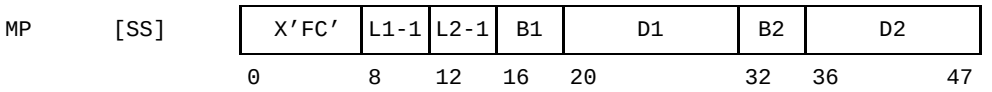
Function

The MP instruction performs signed multiplication of two packed decimal numbers. The product replaces the first operand.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
*	MP	D1(L1,B1),D2(L2,B2)	L2 < L1 ≤ 16 and 1 < L2 ≤ min(8, L1-1)

Machine format



Description

The packed multiplier is multiplied by the packed multiplicand, with the sign being taken into account. The packed product replaces the multiplicand.

The multiplicand and the product are addressed in main memory by D1(B1), the multiplier by D2(B2). The length of the multiplier (L2) must be at least 1 and no more than 8 bytes; it must also be less than the length of the multiplicand ($1 \leq L2 \leq Min(L1-1,8)$). The length of the multiplicand (L1) must be greater than the length of the multiplier ($L2 < L1 \leq 16$). If these conditions are not satisfied, a program interrupt will occur due to an addressing error.

The multiplicand must contain at least as many leading bytes with the contents 00_{16} as the length of the multiplier; otherwise, a program interrupt will occur due to a data error. A program interrupt will also occur when one or both operands do not represent correctly packed decimal numbers.

The sign of the product is determined according to the algebraic rules, even if one or both of the operands are =0.

A decimal overflow cannot occur.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.
Addressing error	X'5C'	L1 or L2 wrong.
Data error	X'60'	1. Incorrect format in operand. 2. Multiplicand does not have at least L2 leading zero bytes.

Programming notes

- Both operands are processed as integers; the product is in integer.
- In the product, a positive sign is represented by C_{16}, and a negative sign by D_{16}.
- The multiplicand and multiplier may overlap, but in this case the addresses of their lowest-order bytes must be identical (D1(B1)+L1-1= D2(B2)+L2-1); otherwise, a program interrupt will occur due to a data error. Furthermore, L1 must be greater than L2, i.e. D1(B1)<D2(B2), and the uppermost L2 bytes in the multiplicand must be $=00_{16}$ (see example).
- A product of =0 may have a negative sign.
- The result of an MP instruction can be at least one leading zero (0_{16}).
- The result of an MP instruction can be at most $10^{30}-2*10^{15}+1$, i.e. it can have at most 29 digits.

Examples

The sample MP instructions below yield the following results:

DFIELD before	Sample instruction	DFIELD after
PL2’-9’	MP DFIELD,=P’2’	PL2’-18’
PL2’-9’	MP DFIELD,=P’0’	PL2’-0’
PL2’-9’	MP DFIELD,DFIELD+1(1)	PL2’+81’

Note that the instruction MP DFIELD,DFIELD would be incorrect when used instead of example 3, since it does not satisfy the initial condition L2<L1.

Subtract Decimal

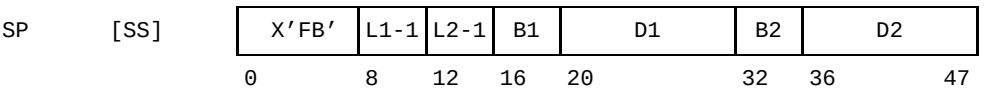
Function

The SP instruction subtracts two packed decimal numbers. The resulting difference replaces the first operand.
The condition code is set in accordance with the value of the difference.

Assembler format

Name	Operation	Operands	Remarks
	SP	D1(L1,B1),D2(L2,B2)	1 ≤ L1,L2 ≤ 16

Machine format



Description

The packed decimal number in the field of the second operand (D2(L2,B2)) is subtracted from the packed decimal number in the field of the first operand (D1(L1,B1)), with the sign being taken into account. The packed difference replaces the first operand.

The first operand and the difference have a length of L1 bytes; the second operand has a length of L2 bytes, where $1 \leq L1,L2 \leq 16$.

Both operands are checked for a correct packed format; in case of error, a program interrupt will occur due to a data error.

A decimal overflow will occur if the result has more significant decimal positions than will fit into the field of the first operand. The instruction is terminated normally, but only the lowest-order 2L1-1 decimal positions in the difference are stored and the highest-order decimal positions are lost; the condition code is then set to 3~Overflow. If the bit for decimal overflow is set to 1 in the program mask (default value in BS2000), a program interrupt will also occur.

A genuine difference of =0 always has a positive sign (C_{16}). However, a difference of =0 resulting from a decimal overflow can also have a negative sign (D_{16}).

Condition code

0~Zero The difference is =0 (it has the sign C_{16}).
1~Minus The difference is <0 (it has the sign D_{16}).
2~Plus The difference is >0 (it has the sign C_{16}).
3~Overflow The difference is more significant decimal positions than will fit into the field of the first operand.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 or read access of operand2 illegal.
Data error	X'60'	Operand has incorrect format.
Decimal overflow	X'74'	Difference too large for first operand.

Programming notes

- Both operands are processed as integers.
- The operands are processed from right to left.
- The operands may overlap, but in this case the addresses of their lowest-order bytes must be identical (D1(B1)+L1-1 = D2(B2)+L2-1); otherwise, a program interrupt will occur due to a data error.
- The second operand is only changed if it overlaps with the first operand.
- If a decimal overflow occurs, the result has the sign of the correct difference. For this reason, it may happen with decimal overflow that a result of =0 has a negative sign.

Examples

The sample SP instructions below yield the following results:

DFIELD before	Sample instruction	DFIELD after	CC
PL1’-2’	SP DFIELD,=P’-10’	PL1’+8’	2
PL1’-2’	SP DFIELD,=PL16’-2’	PL1’+0’	0
PL1’-2’	SP DFIELD,=P’-13’	PL1’+1’	3
PL1’-2’	SP DFIELD,DFIELD	PL1’+0’	0

Note that the second operand may well be longer than the first (see first three examples). Decimal overflow will not occur unless the result of the SP instruction is too long to be stored in the first operand.

Shift and Round Decimal

Function

The SRP instruction shifts a packed decimal number a specified number of decimal positions to the left or right; when shifted to the right, a decimal number will then be rounded in accordance with a specified rounding digit.
The condition code is set in accordance with the value of the result.

Assembler formats

Name	Operation	Operands	Remarks
	SRP	D1(L1,B1),D2(B2),I3	0 ≤ I3 ≤ 9
* or also:			
	SRP	D1(L1,B1),64-r,rz	1 ≤ r ≤ 32; 0 ≤ rz ≤ 9
	SRP	D1(L1,B1),l,rz	0 ≤ l ≤ 31; 0 ≤ rz ≤ 9

Where:

`r` the number of decimal positions to be shifted to the right

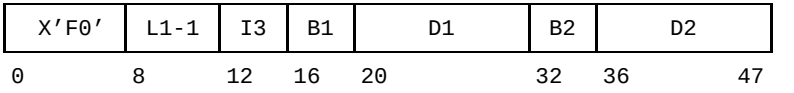
`rz` the rounding digit (≤9) and

`l` the number of decimal positions to be shifted to the left.

With shift left, the direct operand I3 likewise must be specified in the Assembler format, even though it is ignored when the instruction is executed.

Machine format

SRP [SS]



Description

The first operand (addressed by D1(L1,B1)) must be a packed decimal number with a length of L1 bytes (1 ≤ L1 ≤ 16). This decimal number is shifted in the direction and by the number of decimal positions indicated by the address D2(B2). The direct operand I3 must be a decimal digit, i.e. ≤9; it is used as a rounding digit for final rounding following a shift right.

The address determined by D2(B2) is not used as a data address; instead, the rightmost 6 binary positions of this address (bits 26-31) form the shift information: if the highest-order bit in this binary number is =0 (i.e. if bit 26 =0), the shift will take place to the left, namely, by the number of decimal positions specified by the binary number.

Otherwise, if the highest-order bit in the binary number is =1, the shift will take place to the right, namely, by the number of decimal positions specified by the twos complement of the binary number.

With both shift left and shift right the sign position of the first operand remains unchanged. Any digit positions freed will be padded with 0_{16}; this takes place from the right with shift left, and from the left with shift right.

If a significant decimal digit is lost due to a shift left, a decimal overflow occurs. The condition code is then set to 3~Overflow and a program interrupt occurs, provided the bit for decimal overflow is set to =1 in the program mask (default value in BS2000).

Rounding:

After a shift right, the shifted decimal number is then rounded. This is done by adding the most recently shifted decimal digit to the direct operand I3 (rounding digit) and adding any overflow one to the shifted decimal number in accordance with its sign.

The first operand is checked for a correct format. The value of I3 must be a correct decimal number, i.e. ≤ 9, even if the shift is to the left. If an error is detected, a program interrupt will occur due to a data error.

Condition code

0~Zero	Result is = 0 (+0 or -0).
1~Minus	Result is < 0.
2~Plus	Result is > 0.
3~Overflow	decimal overflow.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Read/write access of operand1 illegal.
Data error	X'60'	1. First operand has incorrect format. 2. Rounding digit (I3) not ≤ 9
Decimal overflow	X'74'	Loss of significant decimal digits

Programming notes

– The shift information (direction and number) is obtained from the 6 lowest-order bits of the address D2(B2). These 6 bits form a binary number b in the value range $0 \leq b \leq 63$. If $b \leq 31$ the shift takes place b decimal positions to the left; if $32 \leq b \leq 63$, the shift takes place 64-b decimal places to the right.

  This yields the following limit values:

6 lowest-order bits of D2(B2)	Binary number b	Direction of shift	Number of positions
000000	0	no shift	
000001	1	left	1 decimal pos.
011111	31	left	31 decimal pos.
100000	32	right	32 decimal pos.
111111	63	right	1 decimal pos.

– If B2=0, the shift information is obtained entirely from the 6 lowestorder bits of the D2 field. In this case, "(B2)" may be omitted from the Assembler format.

– If rounding occurs, positive numbers are rounded up and negative numbers are rounded down.

– Standard commercial rounding is obtained by entering the rounding digit 5.

– A pure shift right is obtained by entering a rounding digit 0.

– The maximum value 32 for shift right is sufficient to "zero" the longest possible decimal number (31 would also be sufficient).

– With shift left, too, a third operand I3 must be specified in the Assembler format and must be $\leq 9_{16}$ even though it is ignored when the instruction is executed.

– A packed decimal number GZAHL can be multiplied with a *variable* power of ten 10^{x+k} by entering the variable "x" in general-purpose register B1 and the constant "k" in the D1 field of an SRP. For example, this can be done as follows:

Name	Operation	Operands
	LH	6,=H'x'
	SRP	GZAHL,k(6),0

The special feature of this technique is that it also works if "x+k" is negative, i.e. when dividing by $10^{-(x+k)}$.

Examples

The sample SRP instructions below yield the following results:

DFIELD before	Sample instruction	DFIELD after	CC
PL2’995’	SRP DFIELD,64-1,5	PL2’100’	2
PL2’994’	SRP DFIELD,64-1,5	PL2’99’	2
PL2’-995	SRP DFIELD,64-1,5	PL2’-100’	1
PL2’-1’	SRP DFIELD,3,0	PL2’-0’	3
PL2’-1’	SRP DFIELD,1,0	PL2’-10’	1
PL2’-1’	SRP DFIELD,64-1,9	PL2’-1’	1

Zero and Add

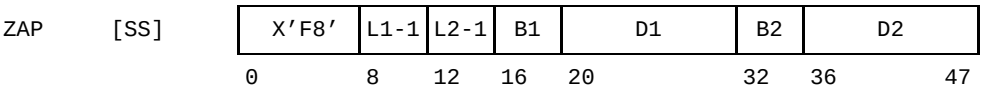
Function

The ZAP instruction moves a packed decimal number to a specified storage area in main memory. The instruction is equivalent to the AP instruction when its first operand is =0.
The condition code is set in accordance with the value of the moved decimal number.

Assembler format

Name	Operation	Operands	Remarks
	ZAP	D1(L1,B1),D2(L2,B2)	1 ≤ L1,L2 ≤ 16

Machine format



Description

D1(L1,B1) addresses the receive field, and D2(L2,B2) the source field (1≤L1,L2≤16).

Only the source field is checked for a correct packed format. If an error is detected, a program interrupt occurs due to a data error.

A decimal overflow occurs when the source field has more significant decimal positions than will fit into the receive field. In this case, the instruction is terminated normally, but only the lowest-order 2L1-1 decimal positions of the source field are moved and the highest-order decimal positions are lost. The condition code is set to 3~Overflow. If the bit for decimal overflow is set to 1 in the program mask (default value in BS2000), a program interrupt will then occur as well.

Once the instruction has been executed, a genuine zero in the source field will always have a positive sign (C_{16}); however, a result of =0 caused by a decimal overflow may also have a negative sign.

Condition code

0~Zero Receive field = 0 (sign C_{16}).
1~Minus Receive field < 0 (sign D_{16}).
2~Plus Receive field > 0 (sign C_{16}).
3~Overflow Source field has more significant decimal positions than will fit into the receive field.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access of operand1 or read access of operand2 illegal.
Data error	X'60'	Source field has incorrect format.
Decimal overflow	X'74'	Second operand too large.

Programming notes

- The source field is only changed if it overlaps with the receive field.
- The move operation takes place from right to left.
- The two operands may overlap. However, a correct result can only be expected if the lowest-order byte in the source field does not lie to the right of the lowest-order byte in the receive field, i.e. if D1(B1)+L1-1 ≤ D2(B2)+L2-1.
- The sign of the receive field is set to $=C_{16}$ or $=D_{16}$ even if it is coded differently in the source field.

Examples

The sample ZAP instructions below yield the following results:

FIELD before	Sample instruction	FIELD after	CC
any	ZAP FIELD(1),=PL1'-1'	PL1'-1'	1
any	ZAP FIELD(1),=PL16'0'	PL1'+0'	0
any	ZAP FIELD(1),=PL8'-10'	PL1'-0'	3

Note that the decimal overflow in the third example did not occur because the second operand is longer than the first, but because its value -10 will not fit in the first operand. With decimal overflow, it may happen (as in this case) that a resultant zero is given a negative sign.

5 Floating-point instructions

Overview

The floating-point instructions are used for processing numbers with large value ranges.

There are floating-point instructions for loading, addition, subtraction, multiplication, division, comparison and sign handling. The instructions process floating-point numbers in three different formats: short, long and extended format.

Most floating-point instructions process two floating-point numbers. Either both of these numbers are located in floating-point registers, or one of them is in main memory and the other in a floating-point register.

Most floating-point instructions create normalized results which represent the highest possible precision; for addition and subtraction, however, there are also instructions which yield unnormalized results, since this can be desirable in many applications (e.g. for subtotals).

Floating-point numbers (sign, characteristic, mantissa)

Each floating-point numbers consist of the sign, the characteristic, and the mantissa.

The sign is a 1-bit number. A zero stands for a positive sign, and a 1 for a negative sign.

The characteristic is an unsigned 7-bit number that represents a base 16 exponent. The exponent itself is obtained by subtracting 64 from the characteristic. The value range of the characteristic extends from 0 to 127, that of the exponent from -64 to +63.

The mantissa is a hexadecimal fraction consisting of 6, 14 or 28 hexadecimal digits, depending on the format used (see below). The (implicit) hexadecimal point of this fraction is located to the left of the highest-order hexadecimal digit.

The value of a floating-point number is obtained from the sign and the product of the mantissa and 16 raised to the power of the exponent:

Value of a floating-point number $= (-1)^{Sign} * mantissa * 16^{exponent}$

$= (-1)^{Sign} * mantissa * 16^{characteristic-64}$

Exponent overflow and underflow

If the resultant exponent in a floating-point operation is less than -64 (i.e. the characteristic is less than 0), an **exponent underflow** occurs. The operation is carried out to the end. If the bit for exponent underflow is set to 1 in the program mask (default value BS2000), a program interrupt then takes place; the mantissa and the sign will be correct, but the characteristic of the result will be 128 too large. If, however, exponent underflow takes place and the associated bit in the program mask is =0, no program interrupt will take place; instead, a so-called genuine zero will be created as a result.

If the resultant exponent in a floating-point operation is greater than 63 (i.e. the characteristic is greater than 127), an **exponent overflow** occurs. The operation is carried out to the end and a program interrupt takes place. (This program interrupt will always take place since there is *no* bit for exponent overflow in the mask.) The characteristic of the result will be 128 too small. The mantissa and the sign, however, will be correct.

Handling the zero, significance

If the mantissa of an addition, subtraction, multiplication or division operation is =0, the sign will always be set to positive; with other operations, however, the sign of a =0 mantissa depends on the sign of the initial operand or operands (the same applies to results with a non-zero mantissa).

If a floating-point addition or subtraction operation produces as a subtotal a mantissa consisting entirely of $=0_{16}$ in its hexadecimal positions, **significance** occurs. The instruction is carried out to the end. If the bit for significance is set to =1 in the program mask, a program interrupt will occur, with the characteristic being correct but the sign and the mantissa being set to =0. If, however, significance occurs and the associated bit is =0, a genuine zero is created and no program interrupt takes place.

A **genuine zero** is a floating-point number whose sign, characteristic and mantissa are all =0. A genuine zero may arise as a normal arithmetic result if the operands have the appropriate values, but it can also be created explicitly, namely in the following cases:

1. Exponent underflow has occurred and the corresponding bit in the program mask is =0.
2. An addition or subtraction operation has resulted in a mantissa of =0 and the mask bit for significance is =0.
3. The operand of a halve instruction, or one or both operands of a multiplication instruction, or the dividend of a division instruction has a mantissa of =0.

Normalization

An amount can be represented with greatest precision by using a floating-point number that is "normalized". A normalized floating-point number is one whose highest-order mantissa position is not equal to 0_{16}. If the highest-order hexadecimal digit in the mantissa is $=0_{16}$, the floating-point number is referred to as unnormalized.

Unnormalized floating-point numbers are normalized by shifting the mantissa to the left by the number of leading hexadecimal zeros, and reducing the characteristic by this same number.

A floating-point number whose mantissa is =0 cannot be normalized; its characteristic is either set to =0 or it is left unchanged, depending on whether the floating-point operation determines that a genuine zero should or should not be created in this case.

The extended addition and subtraction instructions with floating-point operands, as well as all multiplication, division and halving instructions, normalize their results automatically. Floating-point addition and subtraction in short or long format can be activated both with normalized and with unnormalized results. None of the other instructions normalizes its results.

In instructions which do not perform normalization, leading hexadecimal zeros are not eliminated. The result may be either normalized or unnormalized, depending on the initial operands involved.

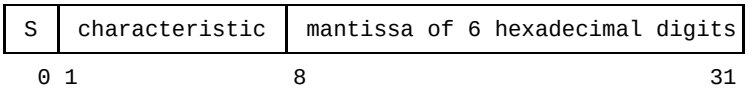
With all floating-point instructions, the initial operands may be either normalized or unnormalized. In the case of multiplication and division instructions, the operands are normalized prior to the actual multiplication or division; other instructions that perform normalization do so only after producing the final result.

If the mantissa overflows when forming the subtotal of an addition, subtraction or rounding operating, it is shifted one hexadecimal position to the right; a one (1_{16}) is entered in the freed hexadecimal position and the characteristic is increased by one. These steps also take place with those instructions which otherwise do not perform normalization.

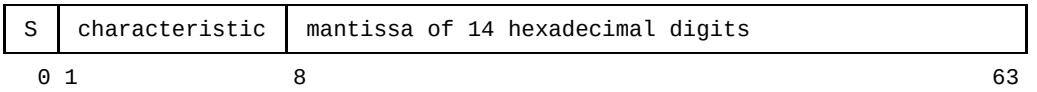
Floating-point formats

There are three formats of floating-point numbers: "short", "long" and "extended". Short format refers to floating-point numbers which are 32 bits long (i.e. one "word"); long format refers to 64-bit floating-point numbers (one "doubleword") and extended format to 128 bit floating-point numbers (two "doublewords"). Floating-point numbers in short or long format may be addressed both in main memory and in floating-point registers; extended floating-point numbers, however, can only be addressed in floating-point registers - or, more precisely, in floating-point register pairs.

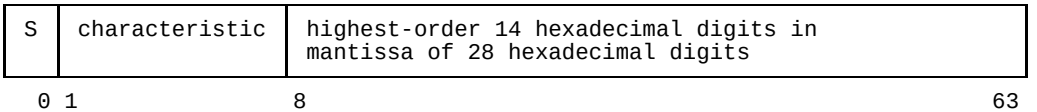
Short format:



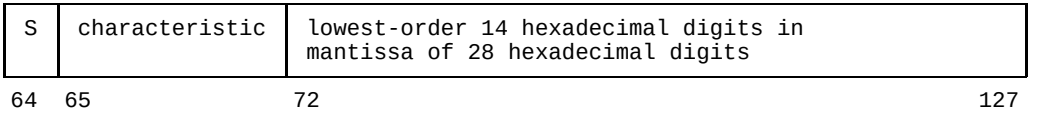
Long format:



Extended format, upper portion:



Extended format, lower portion:



In all three formats, bit 0 forms the sign (S). The next 7 bits (bits 1 to 7) represent the characteristic. With short and long format, the next 24 or 56 bits (bits 8 to 31 or bits 8 to 63) form the mantissa, which consists either of 6 or 14 hexadecimal digits.

A floating-point number in extended format is represented by two floating-point numbers in long format. These are referred to as the "upper portion" and the "lower portion" of the extended floating-point number.

The upper portion of an extended floating-point number may be any floating-point number in long format; its sign and characteristic determine the sign and characteristic of the entire floating-point number. Its mantissa determines the 14 highest-order hexadecimal digits of the 28 hexadecimal digits in the mantissa of the extended floating-point number. If the upper portion is normalized, the entire number is considered to be normalized.

The mantissa of the lower portion of an extended floating-point number determines the 14 lowest-order hexadecimal digits of the 28 hexadecimal digits in the mantissa of the extended floating-point number. The sign and the characteristic of the lower portion are ignored by instructions that process extended floating-point numbers; however, instructions that create extended floating-point numbers also create a sign and a characteristic in the lower portion: the sign is identical to the sign of the upper portion, and the characteristic is 14 less than that of the upper portion.

When an extended floating-point number is created in a floating-point register pair, the lower portion is given the same sign as the upper portion, and its characteristic is set to 14 less than that of the upper portion, unless a genuine zero was created. If, by subtracting 14, the characteristic of the lower portion becomes less than zero, it is set to a value which is 128 too large. The "exponent underflow" state will only occur, however, if there is also an underflow in the characteristic of the upper portion.

When an extended floating-point number is turned into a genuine zero, both the upper and the lower portion are turned into a genuine zero.

Floating-point registers

There are 4 floating-point registers with the numbers (addresses) 0, 2, 4 and 6. These floating-point registers exist alongside the general-purpose registers, which are used in many of the remaining instructions (and in some floating-point instructions for base and index addressing).

Each floating-point register is 64 bits long. Short and long floating-point numbers fit into a single floating-point register, extended floating-point numbers require a **pair** of floating-point registers: either the pair with number 0, consisting of floating-point registers 0 and 2, or the pair with number 4, consisting of floating-point registers 4 and 6.

A short floating-point number occupies only the leftmost 32 bits of the 64 bits in a floating-point register. All floating-point instructions with short operands ignore the rightmost 32 bits or leave them unchanged when short floating-point numbers are created in a register.

If the R1 or R2 field of a floating-point number is given a register number other than 0, 2, 4 or 6, or (in the case of instructions with extended format) a register pair number other than 0 or 4, a program interrupt will occur due to an addressing error.

Value range of floating-point numbers

The absolute value range V of normalized floating-point numbers depends on their format:

Short format:

$$16^{-65} \leq V \leq (1 - 16^{-6}) * 16^{63}$$

Long format:

$$16^{-65} \leq V \leq (1 - 6^{-14}) * 16^{63}$$

Extended format:

$$16^{-65} \leq V \leq (1 - 16^{-28}) * 16^{63}$$

In all three formats, the (absolute) value range V is approximately as follows:

$$5.4 * 10^{-79} \leq V \leq 7.2 * 10^{75}$$

Guard digits

The final result of a floating-point instruction has 6 hexadecimal digits in the case of short format, 14 hexadecimal digits in the case of long format, and 28 hexadecimal digits in the case of extended format. During instruction execution, however, interim results have one extra hexadecimal digit. This (lowest-order) hexadecimal digit is known as the **guard digit**. The guard digit normally increases the accuracy of the result. Its precise effect, however, is specific to the instruction used, and is therefore described for each individual floating-point instruction.

Instruction set

There are 52 floating-point instructions. These cause two floating-point numbers to be added, subtracted, multiplied or divided, or one floating-point number to be loaded, stored, rounded or halved. All instructions use either one floating-point register and one main memory operand, or two floating-point registers.

For short and long floating-point numbers there are floating-point instructions for all the above-named tasks, while for extended floating-point numbers there are only instructions for addition, subtraction, multiplication and division.

Most instructions create as their results a floating-point number in the same format as their initial operands.
However, multiplication instructions create a long (or extended) product from short (or long) operands, and some division instructions create short (or long) quotients from long (or extended) dividends. Finally, two rounding instructions make it possible to round from extended to long format and from long to short format.

Most instructions normalize their results. However, there are addition and subtraction instructions which do not normalize their results. Many instructions leave their results unchanged, so that whether the result is normalized or not depends on the initial operands used.

The instruction for extended division (DXR) is only available on central processing units which have 31-bit addressing mode at their disposal.

The mnemonic operation code of each floating-point instruction contains, at its second or third position, an identifier for the format of the floating-point numbers which it processes. The letters below generally have the following meanings:

E short floating-point format, normalized
U short floating-point format, unnormalized
D long floating-point format, normalized
W long floating-point format, unnormalized
X extended floating-point format, normalized

Programming notes

- A long floating-point number can be converted into an extended floatingpoint number by appending to it a long floating-point number with a mantissa of =0. In particular, this number can be genuine zero. The reverse conversion, i.e. from an extended to a long floating-point number, is accomplished either by means of the LRDR instruction or simply by leaving out the lower portion.
- When exponent overflow or underflow occurs, the second long floating-point number of an extended floating-point number represents the correct lower portion of that number whenever its characteristic is at least 14 less than that of the first long floating-point number. If the difference of the characteristics of both long floating-point numbers is less than 14 and the extended floating-point number is not a genuine zero, then the lower portion is incorrect.
- Up to three leading bits of a normalized floating-point number may be = 0 since normalization takes place one hexadecimal digit at a time, i.e. in 4-bit units.
- BS2000 presets all four mask bits of the program mask to 1. For this reason, a program interrupt will normally occur under the conditions described above for exponent underflow and significance. However, it is possible for the application program to change the presetting using the instruction SPM (Set Program Mask).
- When an extended floating-point number is normalized, the entire 28-digit mantissa is used. The lower portion does not have to be normalized although it constitutes the extended floating-point number.

– To convert a 32-bit fixed-point number into a long floating-point number, and vice versa a long floating-point number into a 32-bit fixed-point number, the following instruction sequences may be used:

```
Name        Operation   Operands
            .
FPTOFL      EQU         *                  Fixed-point no. from GP reg. 0
            ST          0,TMPDWORD+4
            XI          TMPDWORD+4,X'80'
            LD          0,TMPDWORD
            LE          0,TWOEX31
            SD          0,TWOEX31          Floating-point no. in GP reg. 0

FLTOFP      EQU         *                  Floating-point no. in GP reg. 0
            AW          0,TWOEX31
            BM          TOOSMALL           Error: <-2^31
            CE          0,TWOEX31
            BNE         TOOBIG             Error: ≥+2^31
            STD         0,TMPDWORD
            XI          TMPDWORD+4,X'80'
            L           0,TMPDWORD+4       Fixed-point no. in GP reg. 0
            .

* Requisite data:

TMPDWORD    DS          D
TWOEX31     DC          X'4E00000080000000'  8*16^-7*16^14 =2^31
            .
```

The FPTOFL routine first transforms the fixed-point number to be converted from the area $-2^{31}...+2^{31}-1$ to the area $0...2^{32}-1$ by adding the value 2^{31} (-modulo 2^{32}-). This is done by inverting the sign position by means of XI. The routine then makes this number the right portion of the mantissa of a long floating-point number with the exponent 14, i.e. the characteristic $(64+14)_{10} = (4E)_{16}$. Finally, it subtracts from this number the previously added 2^{31} and normalizes the result.

The FLTOFP routine first performs unnormalized addition of the value 2^{31} to the floating-point number to be converted and shifts any non-integer hexadecimal digits to the right. No rounding takes place. If the sum is <0, the floating-point number was $<-2^{31}$, i.e. it was too small for the value range of fixed-point numbers, if the sum is $\geq 2^{32}$, the number was too large. This latter case is shown by the fact that the leftmost 6 hexadecimal digits $\neq 0$. The previously added 2^{31} must now be subtracted from the right portion of the mantissa; as with FPTOFL, this is done by means of an XI instruction. The final L instruction loads the finished fixed-point number into general-purpose register 0.

Add Normalized

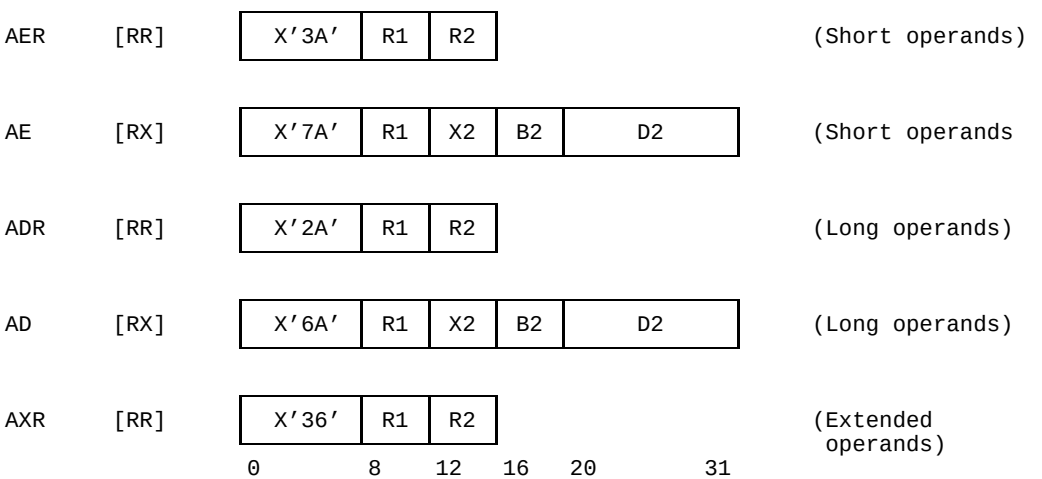
Function

The instructions AER, AE, ADR, AD and AXR add two floating-point numbers. The normalized sum replaces the first operand.
The condition code is set in accordance with the value of the sum.

Assembler formats

Name	Operation	Operands	Remarks
* short addends, short sum:			
	AER	R1,R2	R1,R2 =0, 2, 4 or 6
	AE	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* long addends, long sum:			
	ADR	R1,R2	R1,R2 =0, 2, 4 or 6
	AD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* extended addends, extended sum:			
	AXR	R1,R2	R1,R2 =0 or 4

Machine formats



Description

First the characteristics of both operands are compared; the mantissa of the operand with the smaller characteristic is shifted to the right by the difference of the characteristics, and its characteristic is increased by the same amount, so that the characteristics are equal. The last hexadecimal digit to be shifted beyond the boundary is preserved as a guard digit. The guard digit of the other operand - or of both operands if the characteristics were identical prior to the addition operation - is set to =0.

Next, both mantissas, including the guard digits, are added, with their signs being taken into account. Their sum forms a subtotal consisting of 7 hexadecimal digits in the case of short format, 15 hexadecimal digits in the case of long format, and 29 hexadecimal digits in the case of extended format.

If an overflow occurred, the subtotal is shifted to the right by one hexadecimal position; then a 1_{16} is entered in the hexadecimal position freed to the left, and the characteristic is increased by 1.

Significance occurs when the subtotal, including guard digit, is =0. If in this case the significance bit in the program mask has been set to 1 (default value in BS2000), a program interrupt will occur; otherwise, no program interrupt occurs and a genuine zero will be created as the final result.

If the subtotal, including guard digit, is not equal to 0, it will be normalized, i.e. shifted to the left until the highest-order hexadecimal digit is other than 0_{16}. Any hexadecimal positions freed from the right will be padded with 0_{16}. The characteristic is reduced by the number of shifted hexadecimal positions.

Finally, the normalized subtotal is truncated to 6 or 14 or 28 hexadecimal digits and made into the final result together with the previously calculated characteristic. With extended format, a characteristic which is 14 less than the characteristic of the upper portion is created in the lower portion of the floating-point sum, and the sign of the lower portion is made identical to that of the upper portion.

Exponent overflow occurs when the characteristic of the final result is greater than 127. A program interrupt then takes place: the sign and the mantissa are correct, but the result characteristic(s) are 128 too small.

Exponent underflow occurs when the characteristic of the final result is less than 0. If, in this case, the exponent underflow bit in the program mask has been set to =1 (default value in BS2000), a program interrupt takes place: the sign and the mantissa are correct, but the result characteristic(s) are 128 too large. Otherwise, no program interrupt takes place and a genuine zero is created as final result.

With the AXR instruction, exponent underflow does not occur when only the lower portion of the final result has a characteristic less than 0. In this case, its characteristic is set 128 too large.

Condition code

0~Zero	The mantissa of the final result is = 0; the sign is positive.
1~Minus	Result is < 0.
2~Plus	Result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	AE, AD: Read access of operand2 illegal.
Addressing error	X'5C'	Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.
Exponent overflow	X'64'	Sum characteristic > 127
Significance	X'6C'	Mantissa =0, characteristic 0 and mask bit for significance =1
Exponent underflow	X'70'	Sum characteristic < 0

Programming notes

- Switching the two operands in a normalized addition operation does not in any way change the result.
- Normalized addition normalizes the sum but not the addends.
- BS2000 presets the bits for exponent underflow and significance in the program mask to 1, so that in the above-mentioned cases a program interrupt will occur. An application program can change the presetting by means of the instruction SPM (Set Program Mask).
- With AE and AER, the rightmost 32 bits of the floating-point register involved are ignored or left unchanged.
- R2 may be identical to =R1.

Example

```
Name      Operation   Operands
          .
          DS          0D
FLNO1     DC          X'3F11111111111111'
FLNO2     DC          X'C001111111111111'
          .
          .
          .
          LD          2,FLNO1
          AD          2,FLNO2
          .
```

The final result in floating-point register 2 is X'3210000000000000' together with the condition code 2~Plus. After the characteristics have been unified, the first operand has the value X'4001111111111111' and the guard digit $=1_{16}$. The subtotal is X'40000000000000001'.

Add Unnormalized

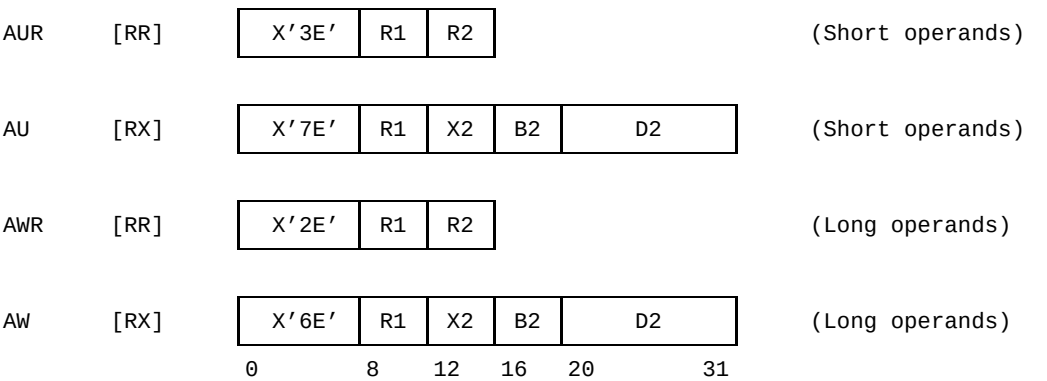
Function

The instructions AUR, AU, AWR and AW add two floating-point numbers. The sum replaces the first operand; it is not normalized.
The condition code is set in accordance with the value of the sum.

Assembler formats

Name	Operation	Operands	Remarks
* short addends, short sum:			
	AUR	R1,R2	R1,R2 =0, 2, 4 or 6
	AU	R1,D2(X2,B2)	R1 =0, 2, 4 or 6
* long addends, long sum:			
	AWR	R1,R2	R1,R2 =0, 2, 4 or 6
	AW	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and

Machine formats



Description

First, the characteristics of both operands are compared; the mantissa of the operand with the smaller characteristic is shifted to the right by the difference of the characteristics, and its characteristic is increased by the same amount, so that the characteristics are equal. The last hexadecimal digit to be shifted beyond the boundary is preserved as a guard digit. The guard digit of the other operand - or of both operands if the characteristics were identical prior to the addition operation - is set to $=0_{16}$.
Next, both mantissas, including the guard digits, are added, with their signs being taken into account. Their sum forms a subtotal consisting of 7 hexadecimal digits in the case of short format and 15 hexadecimal digits in the case of long format.

If an overflow occurred, the subtotal is shifted to the right by one hexadecimal position, then a 1_{16} is entered in the hexadecimal position freed to the left, and the characteristic is increased by 1.

Significance occurs when the subtotal, **including guard digit**, is =0. If in this case the significance bit in the program mask has been set to =1 (default value in BS2000), a program interrupt will occur; otherwise, no program interrupt occurs and a genuine zero will be created as the final result.
The subtotal (without being normalized beforehand) is truncated to 6 or 14 hexadecimal digits and made into the final result together with the previously calculated characteristic.

Exponent overflow occurs when the characteristic of the final result is greater than 127. A program interrupt then takes place: the sign and the mantissa are correct, but the result characteristic(s) are 128 too small.

Exponent underflow cannot occur.

Condition code

0~Zero	The mantissa of the result is = 0; the sign is positive.
1~Minus	The result is < 0.
2~Plus	The result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	AU, AW: Read access of operand2 illegal.
Addressing error	X'5C'	Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.
Exponent overflow	X'64'	Sum characteristic > 127
Significance	X'6C'	Mantissa =0, characteristic 0 and mask bit for significance =1

Programming notes

- Switching the two operands in a normalized addition operation does not in any way change the result.
- BS2000 presets the bit for significance in the program mask to 1, so that in the above-mentioned case a program interrupt will occur. An application program can change the presetting by means of the instruction SPM (Set Program Mask).
- With AU and AUR, the rightmost 32 bits of the floating-point register involved are ignored or left unchanged.
- Unnormalized addition is equivalent to normalized addition except for the following differences:
  - The result is not normalized.
  - Exponent underflow cannot occur.
  - The guard digit is not used for determining significance.
- For extended floating-point operands, there does exist an instruction for normalized addition (AXR), but not for unnormalized addition.

Example

See the example for unnormalized subtraction (SU).

Compare

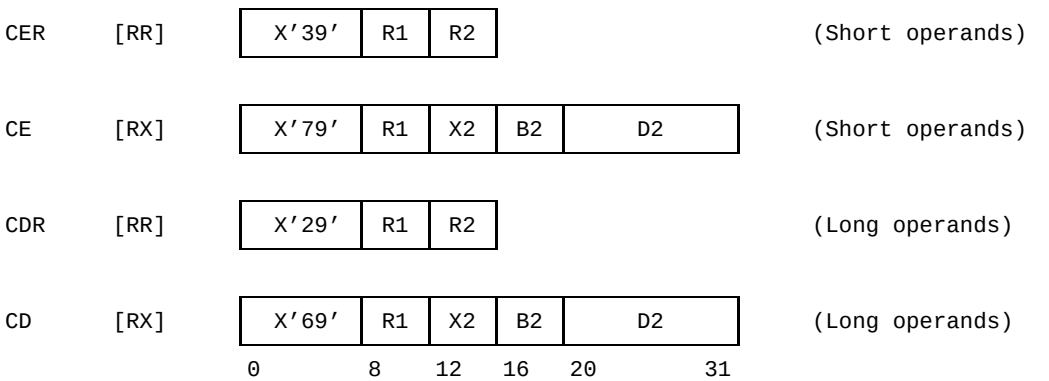
Function

The instructions CER, CE, CDR and CD compare two floating-point numbers. The condition code is set in accordance with the comparison result.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	CER	R1,R2	R1,R2 =0, 2, 4 or 6
	CE	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* long operands:			
	CDR	R1,R2	R1,R2 =0, 2, 4 or 6
	CD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6

Machine formats



Description

The comparison is performed as though a normalized subtraction operation were to take place in which the difference is not stored. The condition code is set to 0~Equal if both operands, including guard digit, are identical; it is set to 1~Low (or 2~High) if the first operand is smaller (larger) than the second operand.

Exponent underflow, exponent overflow and significance cannot occur.

Condition code

0~Equal Operand1 incl. guard digit = operand2.
1~Low Operand1 is < operand2.
2~High Operand1 is > operand2.
3~Overflow Not used.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	CE, CD: Read access of operand2 illegal. Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.

Programming notes

- Two operands, both with mantissa 0, produce the condition code 0~Equal even when they have different signs or characteristics.
- It is not a sufficient condition for inequality when the characteristics of the two operands are different.
- The instructions CE and CER compare only the leftmost 32 bits of their operands; for this reason, it is possible for CE and CER to indicate equality where CD and CDR would not.
- There is no instruction to compare two floating-point operands in extended format.

Example

```
Name     Operation   Operands
         .
         DS          0F
FLNO1    DC          X'48001000'      =16**5+0
FLNO2    DC          X'47010001'      =16**5+16
         .
         .
         .
         LE          6,FLNO1
         CE          6,FLNO2          yields CC 1~Low
         .
```

With this data the instructions above set the condition code to 1~Low because after the characteristics are unified the guard digits are unequal. In contrast, the condition code would already be set to 0~Equal if FLNO2 were only one smaller, i.e. had the value X'4610000F' =16^5+15, since in this case the two guard digits would be identical.

Divide

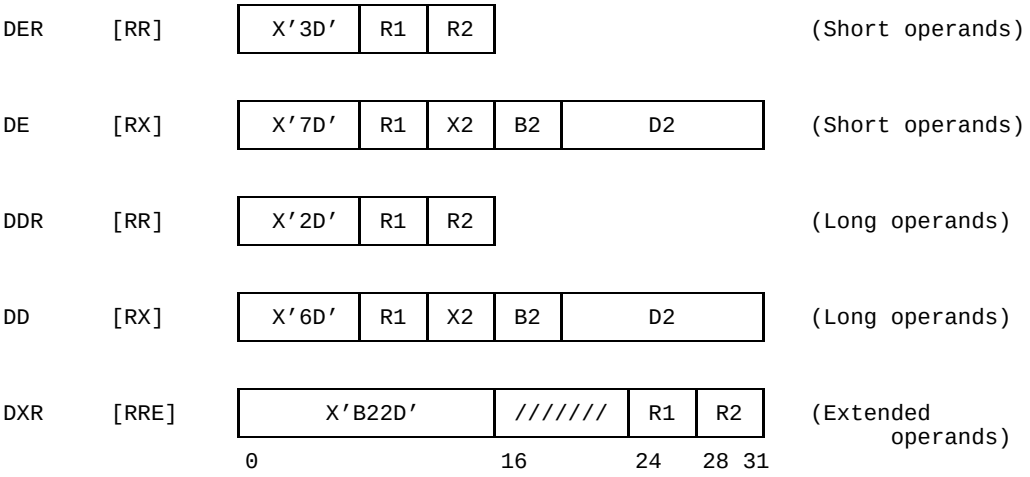
Function

The instructions DER, DE, DDR, DD and DXR divide two floating-point numbers. The normalized quotient replaces the first operand.
The condition code is left unchanged.
The DXR instruction is only available in the instruction set of central processing units that have 31-bit addressing mode at their disposal.

Assembler formats

Name	Operation	Operands	Remarks
*	short operands, short quotient:		
	DER	R1,R2	R1,R2 =0, 2, 4 or 6
	DE	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
*	long operands, long quotient:		
	DDR	R1,R2	R1,R2 =0, 2, 4 or 6
	DD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
*	extended operands, extended quotient:		
	DXR	R1,R2	R1,R2 =0 or 4

Machine formats



Description

Floating-point operand1 is the dividend, floating-point operand2 the divisor. The normalized quotient replaces the first operand. No remainder is created.

First, the dividend and the divisor are normalized so that they do not have any hexadecimal zeros, and their characteristics are adapted (reduced) accordingly. The normalization operation takes place internally and the initial operands are left unchanged; if the dividend (i.e. its amount) is larger than the divisor, it is shifted to the right by one hexadecimal position and its characteristic is increased by 1.

The characteristics of both operands are subtracted and the two (normalized) mantissas are divided. The resultant quotient forms an interim result together with the difference of the characteristics (plus 64) and the algebraic computed sign. All hexadecimal digits in both mantissas are taken into account during the division operation.

Finally, the interim result is truncated to 6 hexadecimal digits in the case of DER and DE, to 14 hexadecimal digits in the case of DDR and DD, and to 28 hexadecimal digits in the case of DXR. It is then made into the result, which is always normalized.

Exponent overflow occurs when the characteristic of the final result is greater than 127 and its mantissa is non-zero. A program interrupt then takes place: the mantissa and sign are correct, but the characteristic of the final result is 128 too small. With DXR, it may happen that the characteristic of the lower portion is also 128 too small.

Exponent underflow occurs when the characteristic of the final result is less than 0 and its mantissa is non-zero. If, in this case, the bit for exponent underflow is set to =1 in the program mask (default value with BS2000) a program interrupt will occur: the mantissa and the sign are correct, but the characteristic is 128 too large; otherwise, no program interrupt will occur and a genuine zero is created as quotient. With DXR, exponent underflow does not occur if the characteristic is less than 0 in the lower portion only. In this case, its characteristic is 128 too large.

Exponent overflow or underflow can only occur with the final result, not when a characteristic overflows or underflows during intermediate computations.

A division error occurs when the mantissa of the divisor is =0 (even if the dividend is likewise =0). A program interrupt takes place.

If the mantissa of the dividend is =0 but the divisor ≠0, a genuine zero is created as a final result.

The sign of the quotient is computed according to the usual algebraic rules; a genuine zero, however, always has a positive sign.

Bit positions 16 to 23 in the DXR instruction are ignored.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	DE, DD: Read access of operand2 illegal.
Invalid operation code	X'58'	DXR attempted on a central processing unit without 31-bit capability
Addressing error	X'5C'	Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.
Exponent overflow	X'64'	Quotient characteristic > 127
Division error	X'68'	Divisor mantissa =0
Exponent underflow	X'70'	Quotient characteristic < 0

Programming notes

– With DER and DE, the rightmost 32 bits of floating-point register R1 are ignored for the mantissa division, and are left unchanged. The same applies to the rightmost 32 bits of floating-point register R2 in the case of DER.

– BS2000 presets the bit for exponent underflow to 1 in the program mask so that under the conditions described above a program interrupt will occur by default. However, the application program can change the presetting using the SPM instruction.

– R2 may be equal to R1.

Example

Name	Operation	Operands
	.	
	DS	0D
DIVIDEND	DC	X’00100000’ $16^{-64}*(16^{-1}+8*16^{-7}) = 1*16^{-71}*(16^{6}+8)$
	DC	X’80000000’
DIVISOR	DC	X’80020000’ $-16^{-64}*(2*16^{-2}+16^{-7}) = -8^{-1}*16^{-71}*(16^{6}+8)$
	DC	X’10000000’
	.	
	.	
	.	
	LD	6,DIVIDEND
	DD	6,DIVISOR The final result in floating-point
	.	register 6 is:
		X’C180000000000000’=$-16^{+1}*8*16^{-1}$=-8.

Note that exponent underflow "actually" occurs when the divisor is normalized. Since, however, it only occurs with the interim result and not with the final result, no program interrupt takes place.

Halve

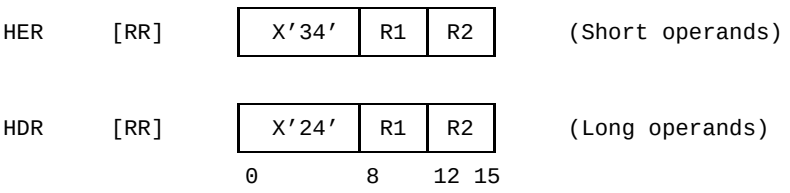
Function

The instructions HER and HDR divide the floating-point number in floating-point register R2 by +2 and store the normalized result in floating-point register R1.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	HER	R1,R2	R1,R2 =0, 2, 4 or 6
* long operands:			
	HDR	R1,R2	R1,R2 =0, 2, 4 or 6

Machine formats



Description

The 6-digit (with HER) or 14-digit (with HDR) mantissa of the floating-point number in floating-point register R2 is shifted one bit to the right and the bit position freed to the left is padded with 0. The bit position right-shifted out of the mantissa is placed to the left in the guard digit and the remaining three bits of the guard digit are set to =0.

The interim result thus produced, including the guard digit, is normalized and the final result is stored in floating-point register R1.

Exponent underflow occurs when the characteristic of the final result is less than 0 and its mantissa is non-zero. If, in this case, the bit for exponent underflow is set to =1 in the program mask (default value in BS2000), a program interrupt takes place: the mantissa and the sign are correct, but the characteristic is 128 too large; otherwise, no program interrupt takes place and a genuine zero is created as the final result.

If the mantissa of the initial operand (in floating-point register R2) is =0, a genuine zero will be created as the final result. Significance or exponent underflow do not occur in this case.

The sign of the result is equal to that of the initial operand; however, a genuine zero always has a positive sign.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	Wrong floating-point register
Exponent underflow	X'70'	Result characteristic < 0

Programming notes

- With HER, the rightmost 32 bits of floating-point register R2 are ignored and the rightmost 32 bits of floating-point register R1 are left unchanged.
- The result of an HER or HDR instruction is always identical to the result of floating-point division using DER or DDR and a divisor of 2.
- BS2000 presets the mask bit for exponent underflow to 1 in the program mask, so that under the conditions described above a program interrupt will occur. However, the application program can change the presetting by means of the SPM instruction.
- A genuine zero can only occur if the initial operand has a mantissa of =0 or when an exponent underflow occurs and the bit for exponent underflow is set to =0 in the program mask.
- R2 may be equal to R1.

Example

```
Name    Operation   Operands
        .
        LE          6,FLNO
        HER         4,6
        .
        .
        .
        DS          0F
FLNO    DC          X'86000001'          -1*16**-6*16**-58
        .
```

Following instruction execution, floating-point register 4 has the value: X'80800000' =-0,5*16^{-64}.

The condition code and the rightmost portion of floating-point register 4 are left unchanged.

Load Complement

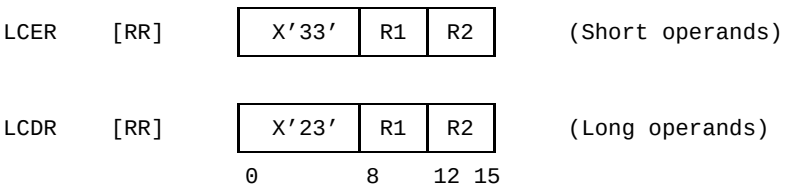
Function

The instructions LCER and LCDR load the floating-point number in floating-point register R2 into floating-point register R1, reversing its sign and setting the condition code in accordance with the value in R1.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	LCER	R1,R2	R1,R2 =0, 2, 4 or 6
* long operands:			
	LCDR	R1,R2	R1,R2 =0, 2, 4 or 6

Machine formats



Description

The short (LCER) or long (LCDR) floating-point number in floating-point register R2 is moved to floating-point register R1 after reversing its sign. No normalization takes place.

The sign is also reversed when the mantissa of the initial operand is =0; however, in this case the condition code is set to 0~Zero.

Condition code

0~Zero	Mantissa of result = 0.
1~Minus	Result is < 0.
2~Plus	Result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	Wrong floating-point register specified.

Programming notes

– R1 may be equal to R2.

– The LCER instruction moves only the leftmost 32 bits of floating-point register R2 and leaves the rightmost 32 bits of floating-point register R1 unchanged.

Load

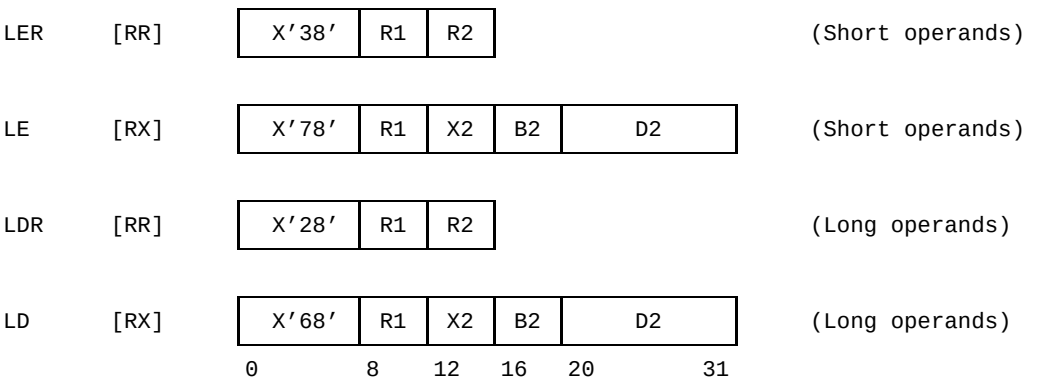
Function

The instructions LER, LE, LDR and LD load a floating-point number into a floating-point register.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	LER	R1,R2	R1,R2 =0, 2, 4 or 6
	LE	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* long operands:			
	LDR	R1,R2	R1,R2 =0, 2, 4 or 6
	LD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6

Machine formats



Description

The short (LE and LER) or long (LD and LDR) floating-point number in floating-point register R2 (LER and LDR) or in the main memory word (LE) or main memory doubleword (LD) is loaded into floating-point register R1. No normalization takes place.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X'48' X'5C'	LE, LD: Read access of operand2 illegal. Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.

Programming notes

The instructions LE and LER leave the rightmost 32 bits of floating-point register R1 unchanged.

Load Negative

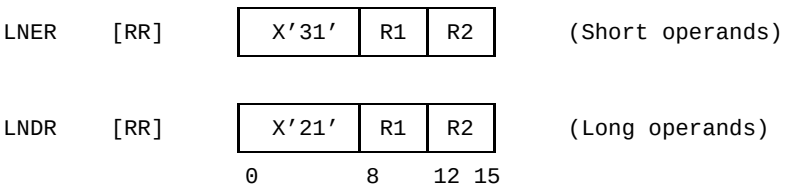
Function

The instructions LNER and LNDR load the floating-point number in floating-point register R2 with a negative sign into floating-point register R1 and set the condition code in accordance with the value in R1.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	LNER	R1,R2	R1,R2 =0, 2, 4 or 6
* long operands:			
	LNDR	R1,R2	R1,R2 =0, 2, 4 or 6

Machine formats



Description

The short (LNER) or long (LNDR) floating-point number in floating-point register R2 is moved with a negative sign into floating-point register R1. No normalization takes place.

The sign is also set to negative if the mantissa of the initial operand is =0; however, in this case the condition code is set to 0~Zero.

Condition code

0~Zero	Mantissa of result = 0.
1~Minus	Result is < 0.
2	Not used.
3	Not used.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	Wrong floating-point register specified.

Programming notes

- R1 may be equal to R2
- The LNER instruction moves only the leftmost 32 bits of floating-point register R2 and leaves the rightmost 32 bits of floating-point register R1 unchanged.

Load Positive

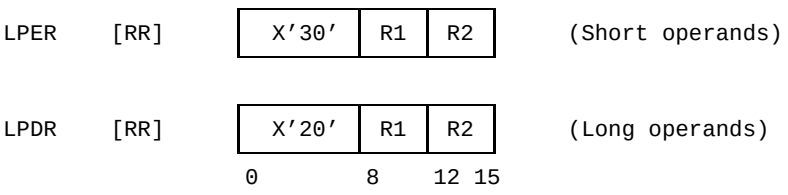
Function

The instructions LPER and LPDR load the floating-point number in floating-point register R2 with a positive sign into floating-point register R1 and set the condition code in accordance with the value in R1.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	LPER	R1,R2	R1,R2 =0, 2, 4 or 6
* long operands:			
	LPDR	R1,R2	R1,R2 =0, 2, 4 or 6

Machine formats



Description

The short (LPER) and long (LPDR) floating-point number in floating-point register R2 is moved with a positive sign to floating-point register R1. No normalization takes place.

The sign is also set to positive if the mantissa of the initial operation is =0; however, in this case the condition code is set to 0~Zero.

Condition code

0~Zero	Mantissa of result = 0.
1	Not used.
2~Plus	Result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Addressing error	X’5C’	Wrong floating-point register specified.

Programming notes

– R1 may be equal to R2

– The LPER instruction moves only the leftmost 32 bits of floating-point register R2 and leaves the rightmost 32 bits of floating-point register R1 unchanged.

Load Rounded

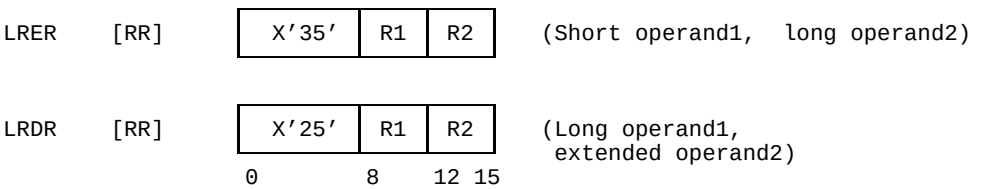
Function

The instructions LRER and LRDR load the floating-point number in floating-point register (pair) R2 into floating-point register R1, rounding it to the next lowest floating-point format.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
* short operand1, long operand2:			
	LRER	R1,R2	R1,R2 =0, 2, 4 or 6
* long operand1, extended operand2:			
	LRDR	R1,R2	R1 =0, 2, 4 or 6 and R2 =0 or 4

Machine formats



Description

The long (LRER) or extended (LRDR) floating-point number in floating-point register R2 or in floating-point register pair R2 and R2+2 is rounded to short (LRER) or long (LRDR) floating-point format and moved to floating-point register R1. No normalization takes place. The sign is left unchanged.

Rounding consists of adding a one to bit position 32 or 72 of the mantissa of the second operand and passing any carry over to the higher-order mantissa positions.

If a carry over beyond the highest-order hexadecimal position of the mantissa occurs during rounding, this position is shifted one position to the right, a 1_{16} is entered in the freed position, and the characteristic is increased by one.

Exponent overflow occurs when the result characteristic is greater than 127. In this case a program interrupt takes place, with the mantissa and the sign being correct but the characteristic being 128 too small.

Exponent underflow and significance cannot occur.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Addressing error	X'5C'	Wrong floating-point register specified.
Exponent overflow	X'64'	Result characteristic > 127

Programming notes

- R1 may be equal to R2. However, note that once the operation has taken place the right portion of general-purpose register R1 (with LRER) or the contents of floating-point register R1+2 (with LRDR) may no longer be used for interpreting the results (see example).
- The LRER instruction leaves the rightmost 32 bits of floating-point register R1 unchanged.

Example

Name	Operation	Operands
	.	
	LD	0,=XL8'C6FFFFFF890ABCDE'
	LRER	0,0
	CD	0,=XL8'C7100000890ABCDE' yields CC 0~Equal
	.	

The above instructions set the condition code to 0~Equal. The initial operand has the value $-(16^7-1+0.5...)$; the result has the value -16^7. This result is a *short* floating-point number. Since the right portion of result register 0 remains unchanged, the interpretation of the result as a *long* floating-point number (shown here for demonstration purposes only) is arithmetically incorrect.

Load and Test

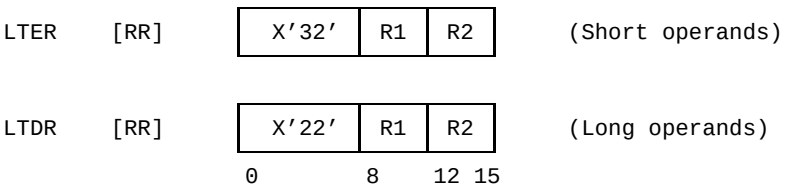
Function

The instructions LTER and LTDR load the floating-point number in floating-point register R2 into floating-point register R1 and set the condition code in accordance with the value in R1.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	LTER	R1,R2	R1,R2 =0, 2, 4 or 6
* long operands:			
	LTDR	R1,R2	R1,R2 =0, 2, 4 or 6

Machine formats



Description

The short (LTER) or long (LTDR) floating-point number in floating-point register R2 is moved to floating-point register R1 without being changed. The condition code is set in accordance with the value of the moved number. No normalization takes place.

Condition code

0~Zero	Mantissa of result is = 0.
1~Minus	Result is < 0.
2~Plus	Result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Addressing error	X’5C’	Wrong floating-point register specified.

Programming notes

– R1 may be equal to R2.

– The LTER instruction moves and tests only the leftmost 32 bits of floating-point register R2 and leaves the rightmost 32 bits of floating-point register R1 unchanged. Accordingly, it may happen that an LTER shows equality where an LTDR does not.

Multiply

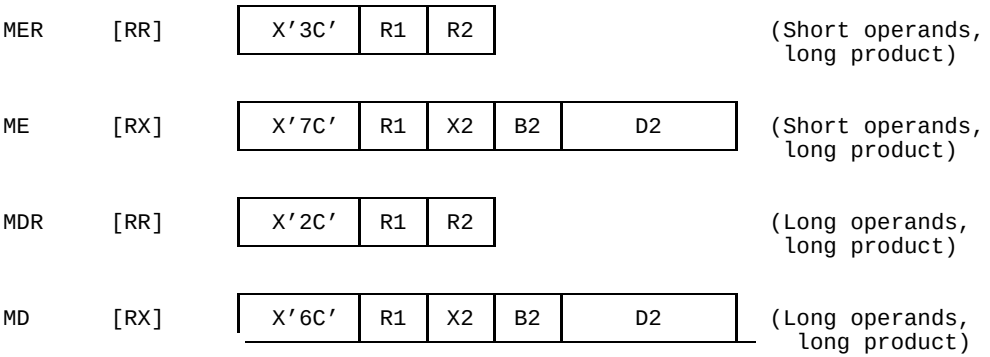
Function

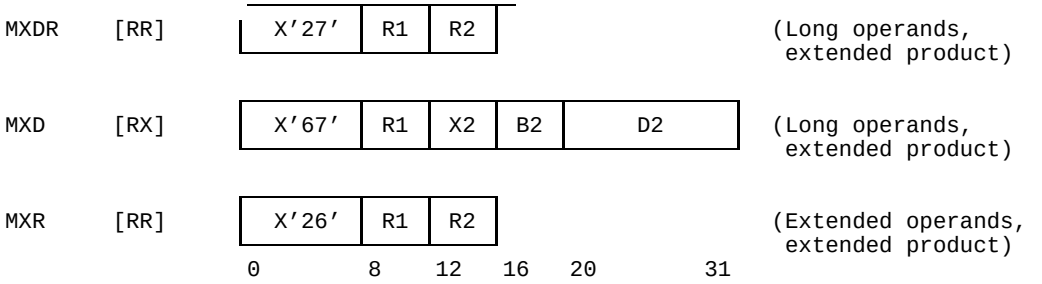
The instructions MER, ME, MDR, MD, MXDR, MXD and MXR multiply two floating-point numbers. The normalized product replaces the first operand.
The condition code is left unchanged.

Assembler formats

Name	Operation	Operands	Remarks
* short multiplier and multiplicand, long product:			
	MER	R1,R2	R1,R2 =0, 2, 4 or 6
	ME	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* short multiplicand and multiplier, long product:			
	MDR	R1,R2	R1,R2 =0, 2, 4 or 6
	MD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* long multiplicand and multiplier, extended product:			
	MXDR	R1,R2	R1 =0 or 4 and
*			R2 =0, 2, 4 or 6
	MXD	R1,D2(X2,B2)	R1 =0 or 4 and
* extended multiplicand and multiplier, extended product:			
	MXR	R1,R2	R1,R2 =0 or 4

Machine formats





Description

The first floating-point operand is the multiplicand; the second floating-point operand is the multiplier. The (normalized) product replaces the first operand.

With the MER and ME instructions, the multiplicand and the multiplier have 6 hexadecimal digits; with MDR, MD, MXDR and MXD they have 14 hexadecimal digits, and with MXR they have 28 hexadecimal digits. The product has 14 hexadecimal digits for MER, ME, MDR and MD, and 28 hexadecimal digits for MXDR, MXD and MXR.

First, the multiplicand and the multiplier are normalized. Normalization takes place internally; the initial operands are left unchanged.

The characteristics of the two operands are added; the two (normalized) mantissas are multiplied; the resultant product forms an interim result together with the characteristic sum minus 64 and the algebraic computed sign. The mantissa of the interim result is exact. If it contains a leading hexadecimal zero, it is shifted one hexadecimal position to the left and the characteristic is reduced by 1. Lastly, the final result is created by enlarging the interim result with two hexadecimal zeros to 14 hexadecimal positions (with ME and MER), or shortening it to 14 or 28 hexadecimal positions (with the remaining instructions).

Exponent overflow occurs when the characteristic of the final result is greater than 127 and its mantissa is non-zero. A program interrupt then takes place: the mantissa and the sign are correct, but the characteristic of the result is 128 too small. With MXDR, MXD and MXR it may happen that the characteristic of the lower portion is also 128 too small.

Exponent underflow occurs when the characteristic of the final result is less than 0, and its mantissa is non-zero. If in this case the bit for exponent underflow is set to =1 in the program mask (default value in BS2000) a program interrupt will take place: the mantissa and the sign will be correct, but the characteristic of the result will be 128 too large; otherwise no program interrupt will take place and the genuine zero will be created as the final result. With MDXR, MXD and MXR, exponent underflow is not acknowledged when only the lower portion underflows.

Exponent overflow only occurs with the final result, not when a characteristic overflows in an interim result.

The sign of the final result is computed according to the usual algebraic rules; however, a genuine zero always has a positive sign.

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	ME, MD, MXD: Read access of operand2 illegal.
Addressing error	X'5C'	Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.
Exponent overflow	X'64'	Product characteristic > 127
Exponent underflow	X'70'	Product characteristic < 0

Programming notes

- Switching the multiplicand and the multiplier does not change the result in any way.
- With MER and ME, the rightmost 32 bits of the floating-point registers involved are ignored when the mantissas are multiplied. On the other hand, with these instructions the rightmost 32 bits are overwritten by the product.
- With MXDR and MXD the contents of the floating-point register R1+2 are ignored when the mantissas are multiplied. However, its contents are overwritten by the lower portion of the product. With MXDR, the contents of floating-point register R2+2 are also ignored.
- BS2000 presets the mask bit for exponent underflow to 1 in the program mask, so that under the conditions described above a program interrupt will occur. However, the application program can change the presetting by using the SPM instruction.
- R2 may be equal to R1.
- With the instructions MER, ME, MXDR and MXD the result is exact; with MDR, MD and MXR hexadecimal digits located to the right may be lost as a result of truncation.

Example

```
Name      Operation   Operands
          .
FLNO1     DC          EE2'2.56'              =X'43100000'
FLNO2     DC          ES4'-16'               =X'C6000010'
FLNO3     DC          D'-4096'               =X'C410000000000000'
          .
          .
          .
          LE          6,FLNO1
          ME          6,FLNO2
          CD          6,FLNO3                yields CC 0~Equal
          .
```

The ME instruction creates the value D'4096'= X'C410000000000000'; the CD instruction sets the condition code to 0~Equal.
This example makes use of the Assembler options for the data declaration of floating-point numbers. The E-type constants in the data declarations for FLNO1 and FLNO2 cause the assembler to generate short floating-point numbers; the D-type constant causes generation of a long floating-point number. The exponent factor "E2" (for FLNO1) causes the argument 2.56 to be multiplied by 10^2, and the scaling factor "S4" (for FLNO2) creates a mantissa shifted 4 hexadecimal positions to the right. Further options for floating-point data declarations can be found in the ASSEMBH Reference Manual [1].

Subtract Normalized

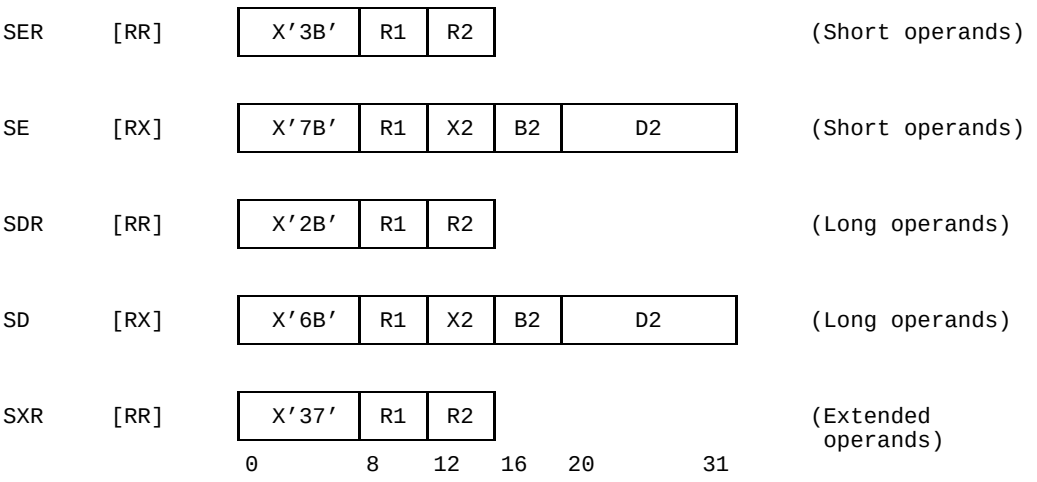
Function

The instructions SER, SE, SDR, SD and SXR subtract two floating-point numbers. The normalized difference replaces the first operand.
The condition code is set in accordance with the value of the difference.

Assembler formats

Name	Operation	Operands	Remarks
*	short operands, short difference:		
	SER	R1,R2	R1,R2 =0, 2, 4 or 6
	SE	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
*	long operands, long difference:		
	SDR	R1,R2	R1,R2 =0, 2, 4 or 6
	SD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
*	extended operands, extended difference:		
	SXR	R1,R2	R1,R2 =0 or 4

Machine formats



Description

First, the characteristics of both operands are compared; the mantissa of the operand with the smaller characteristic is shifted to the right by the difference of the characteristics, and its characteristic is increased by the same amount, so that the characteristics are equal. The last hexadecimal digit to be shifted beyond the boundary is preserved as a guard digit. The guard digit of the other operand - or of both operands if the characteristics were identical prior to the subtraction operation - is set to =0.

Next, both mantissas, including the guard digits, are subtracted, with the signs being taken into account (operand1 mantissa minus operand2 mantissa). Their difference forms an interim result consisting of 7 hexadecimal digits in the case of short format, 15 hexadecimal digits in the case of long format, and 29 hexadecimal digits in the case of extended format.
If an overflow occurred, the interim result is shifted to the right by one hexadecimal position; then a 1_{16} is entered in the hexadecimal position freed to the left, and the characteristic is increased by 1.

Significance occurs when the interim result, including guard digit, is =0. If in this case the significance bit in the program mask has been set to =1 (default value in BS2000), a program interrupt will occur; otherwise, no program interrupt occurs and a genuine zero will be created as a final result.

If the interim result, including guard digit, is ≠0, it will be normalized, i.e. shifted to the left until the highest-order hexadecimal digit is other than 0_{16}. Any hexadecimal positions freed from the right will be padded with 0_{16}. The characteristic is reduced by the number of shifted hexadecimal positions.
Finally, the normalized interim result is truncated to 6 or 14 or 28 hexadecimal digits and made into the final result together with the previously calculated characteristic. With extended format, a characteristic which is 14 less than the characteristic of the upper portion is created in the lower portion of the floating-point difference, and the sign of the lower portion is made identical to that of the upper portion.

Exponent overflow occurs when the characteristic of the final result is greater than 127. A program interrupt then takes place: the sign and the mantissa are correct, but the result characteristic(s) are 128 too small.

Exponent underflow occurs when the characteristic of the final result is less than 0. If, in this case, the exponent underflow bit in the program mask has been set to =1 (default value in BS2000), a program interrupt takes place: the sign and the mantissa are correct, but the result characteristic(s) are 128 too large. Otherwise, no program interrupt takes place and a genuine zero is created as the final result.
With the SXR instruction, exponent underflow does not occur when only the lower portion of the final result has a characteristic less than 0. In this case, its characteristic is set 128 too large.

Condition code

0~Zero	The mantissa of the final result is = 0; the sign is positive.
1~Minus	Result is < 0.
2~Plus	Result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	SE, SD: read access of operand2 illegal.
Addressing error	X'5C'	Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.
Exponent overflow	X'64'	Characteristic of difference > 127.
Significance	X'6C'	Mantissa =0, characteristic 0 and mask bit for significance =1.
Exponent underflow	X'70'	Characteristic of difference < 0.

Programming notes

- Normalized subtraction normalizes the difference, but not the initial operands.
- BS2000 presets the bits for exponent underflow and significance to 1 in the program mask, so that in the aforementioned cases a program interrupt will occur. An application program can change the presetting using the instruction SPM (Set Program Mask).
- With SE and SER the rightmost 32 bits of the floating-point register involved are ignored and are left unchanged.
- R2 may be equal to R1; in this case, the result will be a genuine zero.

Example

```
Name      Operation   Operands
          .
          DS          0F
FLNO1     DC          X'46100000'
FLNO2     DC          X'40200000'
          .
          .
          .
          LE          0,FLNO1
          SE          0,FLNO2
          .
```

The final result in the floating-point register is X'45FFFFFE' and the condition code is set to 2~Plus.
After the characteristics were unified, the second operand had the value X'46000000' and the guard digit was $=2_{16}$. The interim result was X'460FFFFFE'.

Store

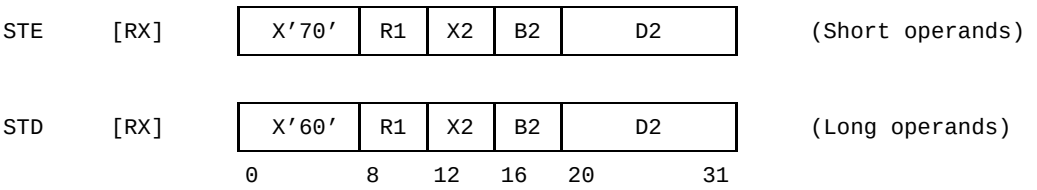
Function

The instructions STE and STD store the floating-point number located in floating-point register R1 in a main memory field.

Assembler formats

Name	Operation	Operands	Remarks
* short operands:			
	STE	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* long operands:			
	STD	R1,D2(X2,B2)	R1 =0, 2, 4 or 6

Machine formats



Description

The short (or long) floating-point number in floating-point register R1 is stored in the word (or doubleword) at the main memory location indicated by D2(X2,B2).

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error Addressing error	X’48’ X’5C’	Write access of operand2 illegal. Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.

Subtract Unnormalized

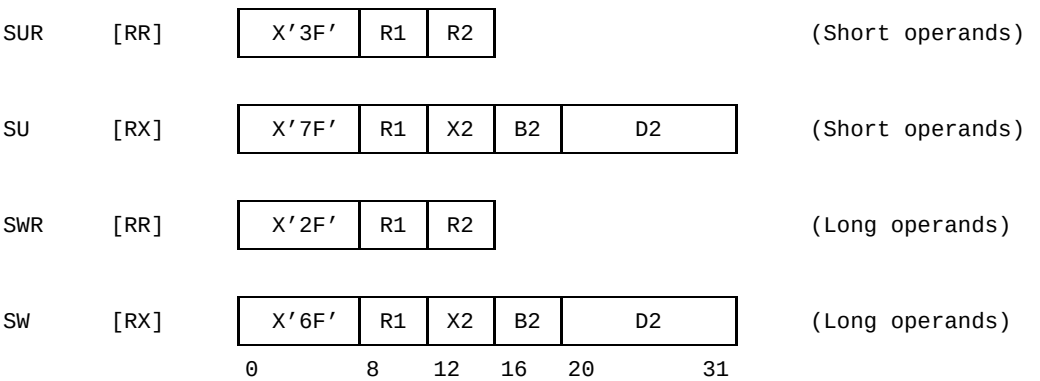
Function

The instructions SUR, SU, SWR and SW subtract two floating-point numbers. The difference replaces the first operand; it is not normalized. The condition code is set in accordance with the value of the difference.

Assembler formats

Name	Operation	Operands	Remarks
* short operands, short difference:			
	SUR	R1,R2	R1,R2 =0, 2, 4 or 6
	SU	R1,D2(X2,B2)	R1 =0, 2, 4 or 6 and
* long operands, long difference:			
	SWR	R1,R2	R1,R2 =0, 2, 4 or 6
	SW	R1,D2(X2,B2)	R1 =0, 2, 4 or 6

Machine formats



Description

First, the characteristics of both operands are compared; the mantissa of the operand with the smaller characteristic is shifted to the right by the difference of the characteristics, and its characteristic is increased by the same amount, so that the characteristics are equal. The last hexadecimal digit to be shifted beyond the boundary is preserved as a guard digit. The guard digit of the other operand - or of both operands if the characteristics were identical prior to the subtraction operation - is set to =0.

Next, both mantissas, including the guard digits, are subtracted, with the signs being taken into account (operand1 mantissa minus operand2 mantissa). Their difference forms an interim result consisting of 7 hexadecimal digits in the case of short format and 15 hexadecimal digits in the case of long format.

If an overflow occurred, the interim result is shifted to the right by one hexadecimal position; then a 1_{16} is entered in the hexadecimal position freed to the left, and the characteristic is increased by 1.

Significance occurs when the interim result, including guard digit, is =0. If in this case the significance bit in the program mask has been set to =1 (default value in BS2000), a program interrupt will occur; otherwise, no program interrupt occurs and a genuine zero will be created as a final result.

The unnormalized interim result is truncated to 6 or 14 hexadecimal digits and made into the final result together with the previously calculated characteristic.

Exponent overflow occurs when the characteristic of the final result is greater than 127. A program interrupt then takes place: the sign and the mantissa are correct, but the result characteristic(s) are 128 too small.

Exponent underflow cannot occur.

Condition code

0~Zero	The mantissa of the final result is = 0; the sign is positive.
1~Minus	Result is < 0.
2~Plus	Result is > 0.
3	Not used.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	SU, SW: Read access of operand2 illegal.
Addressing error	X'5C'	Wrong floating-point reg. specified or D2(X2,B2) not full (double) word boundary.
Exponent overflow	X'64'	Difference characteristic > 127.
Significance	X'6C'	Mantissa =0, characteristic 0 and mask bit for significance =1

Programming notes

- BS2000 presets the significance bit in the program mask to 1, so that in the aforementioned case a program interrupt will occur. An application program can change the presetting by using the instruction SPM (Set Program Mask).
- With SU and SUR the rightmost 32 bits in the floating-point registers involved are ignored and remain unchanged.
- Unnormalized subtraction is equivalent to normalized subtraction except for the following differences:
  - The result is not normalized.
  - Exponent underflow cannot occur.
  - The guard digit is not used for determining significance.
- With extended floating-point operations there is an instruction for normalized subtraction (SXR), but not for unnormalized subtraction.

Example

```
Name    Operation   Operands
        .
        DS          0D
FLNO1   DC          X'4001111111111111'
FLNO2   DC          X'3F1111111111101'
        .
        .
        .
        LD          2,FLNO1
        SW          2,FLNO2
        .
```

The result of the above instructions depends on the value of the significance bit in the program mask: if this bit is =1 (default value in BS2000) the final result is X'4000000000000000' and a program interrupt takes place due to significance; otherwise, the final result is X'0000000000000000' (genuine zero) and no program interrupt takes place. In both cases the condition code is set to 0~Zero.
After the characteristics were unified, the second operand had the value X'4001111111111110' and the guard digit was $=1_{16}$. The interim result was X'400000000000000F'. Note that even with unnormalized subtraction (and addition) the guard digits are subtracted (or added).

6 ESA instructions

Overview

The ESA instructions support the extended virtual address space available on ESA systems.

a) Read/write operations on access registers (CPYA, EAR, SAR, LAM, STAM, LAE).

b) Query AR/ASC mode (IAC).
Set or reset AR/ASC mode (SAC).

c) Check access-register address translations (TAR).

ESA systems (Enterprise System Architecture) support both **program space** (corresponding to conventional address space) and extended address space for data. Like program space, the **data spaces** have virtual addresses (address 0 to 2 gigabytes). Data spaces may contain only data (or program code stored as data) - program code cannot be executed in a data space. A data space is addressed unambiguously by means of the **SPID** (space identification) or one or more **ALETs** (access list entry token). The SPID is assigned when a data space is created and has global validity. ALETs point unambiguously to a data space only within a program. Addressing with ALETs entailed the introduction of **access registers** (see also 2.2.2) as an additional set of registers parallel to the general-purpose registers. The access registers contain the ALETs. When **AR mode** (access register mode) is active, address translation in a machine instruction involves evaluation of the access registers, which means that data in a data space is addressed.

Only programs running on ESA systems under a BS2000 version ≥ V11 and using ESA instructions can store data in a data space of this type. See the "Executive Macros" manual [3].

ESA systems support the 24-bit and 31-bit addressing modes, data spaces and program space. Consequently, ESA systems offer an additional addressing mode known as the AR mode. The XS capability of ESA systems remains available regardless of the AR mode. The program space and each data space created can use either only the lower address space or the lower and higher address spaces. Support for XS programs is described in the manual "Introduction to XS Programming" [2].

AR mode

The AR mode (access register mode) is part of the **ASC mode** (address space control mode). It defines how the access registers are evaluated in address translation:

– If AR mode is switched on, the access registers are evaluated as part of addressing. This enables addresses in the data spaces to be accessed.
  A value 0 in an access register has a special meaning:
  A value 0 in an access register enables the program space to be addressed in AR mode. This is the default value of the access registers when a program starts.

– If AR mode is switched off, the access registers are not evaluated and only addresses in the program space can be accessed. The program runs like a conventional program on a non-ESA system.

For information on the AR mode, use the IAC instruction to query the ASC mode.
The SAC instruction switches the AR mode on and off.

Notes

ESA programming is supported by the following BS2000 macros:
(see manual "Executive Macros" [3])

– DSPSRV creates and releases data space
– ALESRV links task to data space and revokes link
– ALINF requests information on access lists

When addressing with access registers, note the following:

The general-purpose register associated with the access register must be used as the base register. If, say, a general-purpose register is used as an index register, the corresponding access register is ignored. If a general-purpose register is specified as the base register in an instruction, the corresponding access register must be supplied with values correctly in AR mode.
On account of the strict relationship between access registers and base registers, it is important not to switch index and base registers in AR mode.

Copy Access Register

Function

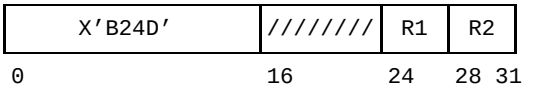
The CPYA instruction transfers the contents of an access register to an access register. The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	CPYA	R1,R2	

Machine format

CPYA [RRE]



Description

The contents of access register R2 are transferred to access register R1.

Bit positions 16 to 23 of the instruction are ignored.

Condition code

Stays the same.

Program interrupts

None.

Extract Access Register

Function

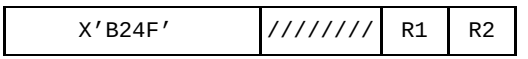
The EAR instruction transfers the contents of an access register to a general-purpose register.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	EAR	R1,R2	

Machine format

EAR [RRE]



Description

The contents of access register R2 are transferred to general-purpose register R1.

Bit positions 16 to 23 of the instruction are ignored.

Condition code

Stays the same.

Program interrupts

None.

Insert Address Space Control

Function

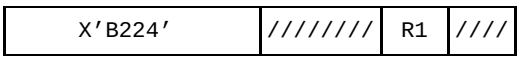
The IAC instruction transfers the current value of the ASC mode to a general-purpose register.
The condition code is set in accordance with the value of the ASC mode.

Assembler format

Name	Operation	Operands	Remarks
	IAC	R1	

Machine format

IAC [RRE]



Description

The ASC mode (primary space mode or access register mode) can be queried either in register R1 or by means of the condition code.

Bit 16 and bit 17 (address space control bits ≙ ASC mode) of the current PSW are reversed and transferred to general-purpose register R1 as bits 22 and 23, in other words bit 16 becomes bit 23 and bit 17 becomes bit 22 of register R1. Bit positions 16 to 21 of register R1 are set to 0, bit positions 0 to 15 and 24 to 31 of the register remain unchanged.

Bit positions 16 to 23 and 28 to 31 of the instruction are ignored.

Condition code

0 primary space mode (PSW bit 16 and PSW bit 17 =0)
2 access register mode (PSW bit 16 =0 and PSW bit 17 =1)

Program interrupts

None.

Load Address Extended

Function

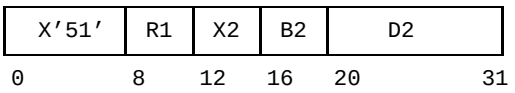
The LAE instruction loads a general-purpose register with an address and the corresponding access register with a value.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	LAE	R1,D2(X2,B2)	

Machine format

LAE [RX]



Description

The address D2(X2,B2) is loaded into general-purpose register R1. The address is the logical sum of the addresses in general-purpose registers X2 and B2 and the binary value of the 12-bit long D2 field. In this calculation the sign is ignored and any carry over past the highest-order binary position is ignored. If X2=0, the content of register X2 is *not* taken into account. If B2=0, the content of register B2 is *not* taken into account.

In the 24-bit addressing mode, only the lowest-order 24 bits of general-purpose registers B2 and X2 are used to calculate the sum. The sum is written into general-purpose register R1 at bit positions 8 to 31 and bit positions 0 to 7 of R1 are set to 0.
In the 31-bit addressing mode only the lowest-order 31 bits of general-purpose registers B2 and X2 are used to calculate the sum. The sum is written into general-purpose register R1 at bit positions 1 to 31 and bit 0 is set to 0.

The corresponding access register R1 receives a value that depends on the AR mode, the current value of bit positions 16 and 17 (address space control bits) of the PSW. If bit positions 16 and 17 have the binary value 01, indicating that AR mode (access register mode) is switched on, the value in the access register is further dependent on whether the B2 field =0 or ≠0 (see table below).

PSW bits 16 and 17	Mode	Value in access register R1
00	primary space mode	X'00000000' i.e. bit positions 0 to 31 =0
01	access register mode	X'00000000', if B2 field =0. If B2 field 0, the content of access register B2 is transferred to access register R1. Bit positions 0 to 6 of access register B2 must be =0, otherwise the results in general-purpose register R1 and in access register R1 are undefined.

The address derived as a result of this operation is not accessed.

Condition code

Stays the same.

Program interrupts

None.

Load Access Multiple

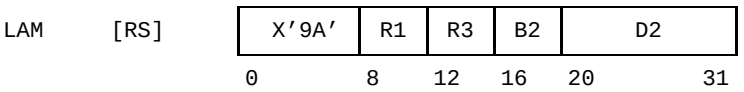
Function

The LAM instruction loads up to 16 consecutive words from main memory into consecutive access registers.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	LAM	R1,R3,D2(B2)	D2(B2): word boundary

Machine format



Description

The consecutive access registers, beginning with R1 and ending with R3, receive consecutive words, the first of which is addressed by D2(B2).

If R1=R3, only one register (R1) receives a value. If R3 is less than R1, loading begins at R1 and continues upwards to access register 15 and from access register 0 up to and including R3.

Name	Operand1	Operand2
LAM	Contents of access registers R1 to R3	Word sequence addressed by D2(B2) No. of words =R3-R1+1, if R3≥R1 =R3-R1+17, if R3<R1

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X’48’	Read access of operand2 illegal
Addressing error	X’5C’	D2(B2) not a word boundary

Set Address Space Control

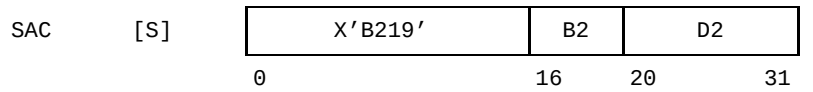
Function

The SAC instruction switches AR mode (access register mode) on and off.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	SAC	D2(B2)	

Machine format



Description

Bit positions 20 to 23 of the D2 field or register B2 set the address space control bits in the PSW (bits 16 and 17), thus switching the AR mode (access register mode) on or off:

- Directly in the D2 field
  Set bits 20 to 23 as shown in the table below. Bit positions 20 and 21 must be =0. Bit positions 24 to 31 are ignored.
- The B2 field defines a general-purpose register (GPR).
  Load the values shown in the table below into the register (bits 20 to 23). Bit positions 20 and 21 must be =0. Bit positions 0 to 19 and 24 to 31 of the register are ignored.

D2 field /GPR reg., bit position 20,21,22,23	Mode	PSW bits 16 and 17
0000	primary space mode	00
0010	access register mode	01

Condition code

Stays the same.

Program interrupts

None.

Programming notes

– The values of bit positions 20 to 23 of the D2 field or the general-purpose register (B2) correspond to those that the IAC instruction stores in a general-purpose register.

Example

```
Name        Operation   Operands
            .
            SAC         512                 switch on AR mode
            SAC         0                   switch off AR mode
            .
            .
```

Set Access Register

Function

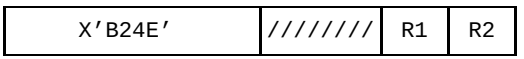
The SAR instruction transfers the contents of a general-purpose register to an access register.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	SAR	R1,R2	

Machine format

SAR [RRE]



Description

The contents of general-purpose register R2 are transferred to access register R1.

Bit positions 16 to 23 of the instruction are ignored.

Condition code

Stays the same.

Program interrupts

None.

Store Access Multiple

Function

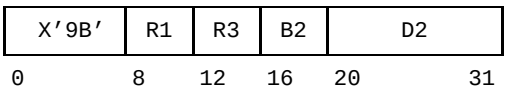
The STAM instruction stores the contents of up to 16 consecutive access registers in consecutive words in main memory.
The condition code is left unchanged.

Assembler format

Name	Operation	Operands	Remarks
	STAM	R1,R3,D2(B2)	D2(B2): word boundary

Machine format

STAM [RS]



Description

The contents of consecutive access registers, the first being R1 and the last R3, are transferred to consecutive words in main memory. The first word is addressed by D2(B2).
If R1 > R3, transfer begins at access register R1 and continues to register 15, and then from access register 0 to register R3. If R1=R3, only one access register (R1) is stored.

Name	Operand1	Operand2
STAM	Contents of access registers R1 to R3	Sequence of words addressed by D2(B2) No. of words =R3-R1+1, if R3≥R1 =R3-R1+17, if R3<R1

Condition code

Stays the same.

Program interrupts

Type	Weight	Causes
Address trans. error	X'48'	Write access to operand2 illegal
Addressing error	X'5C'	D2(B2) not a word boundary

Test Access Register

Function

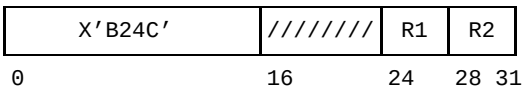
The TAR instruction checks for the occurrence of an exception during address translation involving an access register (ART, access register translation).
The condition code is set in accordance with the ALET value.

Assembler format

Name	Operation	Operands	Remarks
	TAR	R1,R2	

Machine format

TAR [RRE]



Description

The content of access register R1 (ALET, access list entry token) is tested for exceptions detected in the course of the ART (access register translation).
The ALET is tested to ascertain whether it references a valid entry in the access list or contains X'00000000'.

If R1 =0, the content of access register 0 is used in the ART, instead of the conventional value X'00000000'.

BS2000 versions ≥ V11 currently ignore bit positions 0 to 15 of general-purpose register R2. Bit positions 16 to 31 of the register are ignored.

Bit positions 16 to 23 of the instruction are ignored.

Condition code

0 ALET (access list entry token) is X'00000000'.
1 ALET causes no exceptions in the ART (access register translation).
2 ALET causes no exceptions in the ART.
3 ALET causes exceptions in the ART.

Program interrupts

Type	Weight	Causes
Address trans. error Special operation Exception	X'48' X'54'	ESA functions not available

7 Appendix

7.1 EBCDIC table (SRV.10)

EBCDIC.SRV.10

A (zone)

B (digit)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	TC7			SP	&	-						{	}	\	0
1	TC1	DC1					/		a	j	~		A	J		1
2	TC2	DC2		TC9			~		b	k	s		B	K	S	2
3	TC3	DC3					[		c	l	t		C	L	T	3
4							]		d	m	u		D	M	U	4
5	FE1	NL	FE2						e	n	v		E	N	V	5
6		FE0	TCA						f	o	w		F	O	W	6
7	DEL		ESC	TC4			ß		g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM							i	r	z		I	R	Z	9
A						!	^	:								
B	FE3				.	$	,	#	Ä		ä					
C	FE4	IS4		DC4	<	*	%	@	Ö		ö					
D	FE5	IS3	TC5	TC8	(	)	_	’	Ü		ü					
E	SO_1	IS2	TC6		+	;	>	=								
F	SI_1	IS1	BEL	SUB	|		?	"								

1) The control characters SI and SO will probably be dropped from the 8-bit code.

7.2 Instructions listed by mnemonic code

Mnemo. code	Op. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
A	5A	RX	4	yes	Gen	R1,D2(X2,B2)
AD	6A	RX	4	yes	Flpt	R1,D2(X2,B2)
ADR	2A	RR	2	yes	Flpt	R1,R2
AE	7A	RX	4	yes	Flpt	R1,D2(X2,B2)
AER	3A	RR	2	yes	Flpt	R1,R2
AH	4A	RX	4	yes	Gen	R1,D2(X2,B2)
AL	5E	RX	4	yes	Gen	R1,D2(X2,B2)
ALR	1E	RR	2	yes	Gen	R1,R2
AP	FA	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
AR	1A	RR	2	yes	Gen	R1,R2
AU	7E	RX	4	yes	Flpt	R1,D2(X2,B2)
AUR	3E	RR	2	yes	Flpt	R1,R2
AW	6E	RX	4	yes	Flpt	R1,D2(X2,B2)
AWR	2E	RR	2	yes	Flpt	R1,R2
AXR	36	RR	2	yes	Flpt	R1,R2
BAL	45	RX	4	no	Gen	R1,D2(X2,B2)
BALR	05	RR	2	no	Gen	R1,R2
BAS	4D	RX	4	no	Gen	R1,D2(X2,B2)
BASR	0D	RR	2	no	Gen	R1,R2
BASSM	0C	RR	2	no	Gen	R1,R2
BC	47	RX	4	no	Gen	M1,D2(X2,B2)
BCR	07	RR	2	no	Gen	M1,R2
BCT	46	RX	4	no	Gen	R1,D2(X2,B2)
BCTR	06	RR	2	no	Gen	R1,R2
BSM	0B	RR	2	no	Gen	R1,R2
BXH	86	RS	4	no	Gen	R1,R3,D2(B2)
BXLE	87	RS	4	no	Gen	R1,R3,D2(B2)
C	59	RX	4	yes	Gen	R1,D2(X2,B2)
CD	69	RX	4	yes	Flpt	R1,D2(X2,B2)
CDR	29	RR	2	yes	Flpt	R1,R2
CDS	BB	RS	4	yes	Gen	R1,R3,D2(B2)
CE	79	RX	4	yes	Flpt	R1,D2(X2,B2)
CER	39	RR	2	yes	Flpt	R1,R2
CH	49	RX	4	yes	Gen	R1,D2(X2,B2)
CL	55	RX	4	yes	Gen	R1,D2(X2,B2)
CLC	D5	SS	6	yes	Gen	D1(L,B1),D2(B2)
CLCL	0F	RR	2	yes	Gen	R1,R2
CLI	95	SI	4	yes	Gen	D1(B1),I2
CLM	BD	RS	4	yes	Gen	R1,M3,D2(B2)
CLR	15	RR	2	yes	Gen	R1,R2
CP	F9	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
CPYA	B24D	RRE	4	no	ESA	R1,R2
CR	19	RR	2	yes	Gen	R1,R2
CS	BA	RS	4	yes	Gen	R1,R3,D2(B2)
CVB	4F	RX	4	no	Gen	R1,D2(X2,B2)
CVD	4E	RX	4	no	Gen	R1,D2(X2,B2)
D	5D	RX	4	no	Gen	R1,D2(X2,B2)
DD	6D	RX	4	no	Flpt	R1,D2(X2,B2)
DDR	2D	RR	2	no	Flpt	R1,R2
DE	7D	RX	4	no	Flpt	R1,D2(X2,B2)
DER	3D	RR	2	no	Flpt	R1,R2

Mnemo. code	Op. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
DP	FD	SS	6	no	Dcpt	D1(L1,B1),D2(L2,B2)
DR	1D	RR	2	no	Gen	R1,R2
DXR	B22D	RRE	4	no	Flpt	R1,R2
EAR	B24F	RRE	4	no	ESA	R1,R2
ED	DE	SS	6	yes	Dcpt	D1(L,B1),D2(B2)
EDMK	DF	SS	6	yes	Dcpt	D1(L,B1),D2(B2)
EX	44	RX	4	ja*	Gen	R1,D2(X2,B2) *instr.-specific
HDR	24	RR	2	no	Flpt	R1,R2
HER	34	RR	2	no	Flpt	R1,R2
IAC	B224	RRE	4	yes	ESA	R1
IC	43	RX	4	no	Gen	R1,D2(X2,B2)
ICM	BF	RS	4	yes	Gen	R1,M3,D2(B2)
IPM	B222	RRE	4	no	Gen	R1
L	58	RX	4	no	Gen	R1,D2(X2,B2)
LA	41	RX	4	no	Gen	R1,D2(X2,B2)
LAE	51	RX	4	no	ESA	R1,D2(X2,B2)
LAM	9A	RS	4	no	ESA	R1,R3,D2(B2)
LCDR	23	RR	2	yes	Flpt	R1,R2
LCER	33	RR	2	yes	Flpt	R1,R2
LCR	13	RR	2	yes	Gen	R1,R2
LD	68	RX	4	no	Flpt	R1,D2(X2,B2)
LDR	28	RR	2	no	Flpt	R1,R2
LE	78	RX	4	no	Flpt	R1,D2(X2,B2)
LER	38	RR	2	no	Flpt	R1,R2
LH	48	RX	4	no	Gen	R1,D2(X2,B2)
LM	98	RS	4	no	Gen	R1,R3,D2(B2)
LNDR	21	RR	2	yes	Flpt	R1,R2
LNER	31	RR	2	yes	Flpt	R1,R2
LNR	11	RR	2	yes	Gen	R1,R2
LPDR	20	RR	2	yes	Flpt	R1,R2
LPER	30	RR	2	yes	Flpt	R1,R2
LPR	10	RR	2	yes	Gen	R1,R2
LR	18	RR	2	no	Gen	R1,R2
LRDR	25	RR	2	no	Flpt	R1,R2
LRER	35	RR	2	no	Flpt	R1,R2
LTDR	22	RR	2	yes	Flpt	R1,R2
LTER	32	RR	2	yes	Flpt	R1,R2
LTR	12	RR	2	yes	Gen	R1,R2
M	5C	RX	4	no	Gen	R1,D2(X2,B2)
MC	AF	SI	4	yes	Gen	D1(B1),I2
MD	6C	RX	4	no	Flpt	R1,D2(X2,B2)
MDR	2C	RR	2	no	Flpt	R1,R2
ME	7C	RX	4	no	Flpt	R1,D2(X2,B2)
MER	3C	RR	2	no	Flpt	R1,R2
MH	4C	RX	4	no	Gen	R1,D2(X2,B2)
MP	FC	SS	6	no	Dcpt	D1(L1,B1),D2(L2,B2)
MR	1C	RR	2	no	Gen	R1,R2
MVC	D2	SS	6	no	Gen	D1(L,B1),D2(B2)
MVCL	0E	RR	2	yes	Gen	R1,R2
MVI	92	SI	4	no	Gen	D1(B1),I2
MVN	D1	SS	6	no	Gen	D1(L,B1),D2(B2)

Mnemo. code	Op. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
MVO	F1	SS	6	no	Gen	D1(L1,B1),D2(L2,B2)
MVZ	D3	SS	6	no	Gen	D1(L,B1),D2(B2)
MXD	67	RX	4	no	Flpt	R1,D2(X2,B2)
MXDR	27	RR	2	no	Flpt	R1,R2
MXR	26	RR	2	no	Flpt	R1,R2
N	54	RX	4	yes	Gen	R1,D2(X2,B2)
NC	D4	SS	6	yes	Gen	D1(L,B1),D2(B2)
NI	94	SI	4	yes	Gen	D1(B1),I2
NR	14	RR	2	yes	Gen	R1,R2
O	56	RX	4	yes	Gen	R1,D2(X2,B2)
OC	D6	SS	6	yes	Gen	D1(L,B1),D2(B2)
OI	96	SI	4	yes	Gen	D1(B1),I2
OR	16	RR	2	yes	Gen	R1,R2
PACK	F2	SS	6	no	Gen	D1(L1,B1),D2(L2,B2)
S	5B	RX	4	yes	Gen	R1,D2(X2,B2)
SAC	B219	S	4	no	ESA	D2(B2)
SAR	B24E	RRE	4	no	ESA	R1,R2
SD	6B	RX	4	yes	Flpt	R1,D2(X2,B2)
SDR	2B	RR	2	yes	Flpt	R1,R2
SE	7B	RX	4	yes	Flpt	R1,D2(X2,B2)
SER	3B	RR	2	yes	Flpt	R1,R2
SH	4B	RX	4	yes	Gen	R1,D2(X2,B2)
SL	5F	RX	4	yes	Gen	R1,D2(X2,B2)
SLA	8B	RS	4	yes	Gen	R1,D2(B2)
SLDA	8F	RS	4	yes	Gen	R1,D2(B2)
SLDL	8D	RS	4	no	Gen	R1,D2(B2)
SLL	89	RS	4	no	Gen	R1,D2(B2)
SLR	1F	RR	2	yes	Gen	R1,R2
SP	FB	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
SPM	04	RR	2	yes	Gen	R1
SR	1B	RR	2	yes	Gen	R1,R2
SRA	8A	RS	4	yes	Gen	R1,D2(B2)
SRDA	8E	RS	4	yes	Gen	R1,D2(B2)
SRDL	8C	RS	4	no	Gen	R1,D2(B2)
SRL	88	RS	4	no	Gen	R1,D2(B2)
SRP	F0	SS	6	yes	Dcpt	D1(L1,B1),D2(B2),I3
ST	50	RX	4	no	Gen	R1,D2(X2,B2)
STAM	9B	RS	4	no	ESA	R1,R3,D2(B2)
STC	42	RX	4	no	Gen	R1,D2(X2,B2)
STCK	B205	S	4	yes	Gen	D2(B2)
STCM	BE	RS	4	no	Gen	R1,M3,D2(B2)
STD	60	RX	4	no	Flpt	R1,D2(X2,B2)
STE	70	RX	4	no	Flpt	R1,D2(X2,B2)
STH	40	RX	4	no	Gen	R1,D2(X2,B2)
STM	90	RS	4	no	Gen	R1,R3,D2(B2)
SU	7F	RX	4	yes	Flpt	R1,D2(X2,B2)
SUR	3F	RR	2	yes	Flpt	R1,R2
SVC	0A	RR	2	no	Gen	I
SW	6F	RX	4	yes	Flpt	R1,D2(X2,B2)
SWR	2F	RR	2	yes	Flpt	R1,R2
SXR	37	RR	2	yes	Flpt	R1,R2
TAR	B24C	RRE	4	yes	ESA	R1,R2
TM	91	SI	4	yes	Gen	D1(B1),I2

Mnemo. code	Op. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
TR	DC	SS	6	no	Gen	D1(L,B1),D2(B2)
TRT	DD	SS	6	yes	Gen	D1(L,B1),D2(B2)
TS	93	S	4	yes	Gen	D2(B2)
UNPK	F3	SS	6	no	Gen	D1(L1,B1),D2(L2,B2)
X	57	RX	4	yes	Gen	R1,D2(X2,B2)
XC	D7	SS	6	yes	Gen	D1(L,B1),D2(B2)
XI	97	SI	4	yes	Gen	D1(B1),I2
XR	17	RR	2	yes	Gen	R1,R2
ZAP	F8	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)

7.3 Instructions listed by operation code

Op. code	Mnemo. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
04	SPM	RR	2	yes	Gen	R1
05	BALR	RR	2	no	Gen	R1,R2
06	BCTR	RR	2	no	Gen	R1,R2
07	BCR	RR	2	no	Gen	M1,R2
0A	SVC	RR	2	no	Gen	I
0B	BSM	RR	2	no	Gen	R1,R2
0C	BASSM	RR	2	no	Gen	R1,R2
0D	BASR	RR	2	no	Gen	R1,R2
0E	MVCL	RR	2	yes	Gen	R1,R2
0F	CLCL	RR	2	yes	Gen	R1,R2
10	LPR	RR	2	yes	Gen	R1,R2
11	LNR	RR	2	yes	Gen	R1,R2
12	LTR	RR	2	yes	Gen	R1,R2
13	LCR	RR	2	yes	Gen	R1,R2
14	NR	RR	2	yes	Gen	R1,R2
15	CLR	RR	2	yes	Gen	R1,R2
16	OR	RR	2	yes	Gen	R1,R2
17	XR	RR	2	yes	Gen	R1,R2
18	LR	RR	2	no	Gen	R1,R2
19	CR	RR	2	yes	Gen	R1,R2
1A	AR	RR	2	yes	Gen	R1,R2
1B	SR	RR	2	yes	Gen	R1,R2
1C	MR	RR	2	no	Gen	R1,R2
1D	DR	RR	2	no	Gen	R1,R2
1E	ALR	RR	2	yes	Gen	R1,R2
1F	SLR	RR	2	yes	Gen	R1,R2
20	LPDR	RR	2	yes	Flpt	R1,R2
21	LNDR	RR	2	yes	Flpt	R1,R2
22	LTDR	RR	2	yes	Flpt	R1,R2
23	LCDR	RR	2	yes	Flpt	R1,R2
24	HDR	RR	2	no	Flpt	R1,R2
25	LRDR	RR	2	no	Flpt	R1,R2
26	MXR	RR	2	no	Flpt	R1,R2
27	MXDR	RR	2	no	Flpt	R1,R2
28	LDR	RR	2	no	Flpt	R1,R2
29	CDR	RR	2	yes	Flpt	R1,R2
2A	ADR	RR	2	yes	Flpt	R1,R2
2B	SDR	RR	2	yes	Flpt	R1,R2
2C	MDR	RR	2	no	Flpt	R1,R2
2D	DDR	RR	2	no	Flpt	R1,R2
2E	AWR	RR	2	yes	Flpt	R1,R2
2F	SWR	RR	2	yes	Flpt	R1,R2
30	LPER	RR	2	yes	Flpt	R1,R2
31	LNER	RR	2	yes	Flpt	R1,R2
32	LTER	RR	2	yes	Flpt	R1,R2
33	LCER	RR	2	yes	Flpt	R1,R2
34	HER	RR	2	no	Flpt	R1,R2
35	LRER	RR	2	no	Flpt	R1,R2
36	AXR	RR	2	yes	Flpt	R1,R2
37	SXR	RR	2	yes	Flpt	R1,R2
38	LER	RR	2	no	Flpt	R1,R2

Op. code	Mnemo. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format	
39	CER	RR	2	yes	Flpt	R1,R2	
3A	AER	RR	2	yes	Flpt	R1,R2	
3B	SER	RR	2	yes	Flpt	R1,R2	
3C	MER	RR	2	no	Flpt	R1,R2	
3D	DER	RR	2	no	Flpt	R1,R2	
3E	AUR	RR	2	yes	Flpt	R1,R2	
3F	SUR	RR	2	yes	Flpt	R1,R2	
40	STH	RX	4	no	Gen	R1,D2(X2,B2)	
41	LA	RX	4	no	Gen	R1,D2(X2,B2)	
42	STC	RX	4	no	Gen	R1,D2(X2,B2)	
43	IC	RX	4	no	Gen	R1,D2(X2,B2)	
44	EX	RX	4	ja*	Gen	R1,D2(X2,B2)	*instr.-specific
45	BAL	RX	4	no	Gen	R1,D2(X2,B2)	
46	BCT	RX	4	no	Gen	R1,D2(X2,B2)	
47	BC	RX	4	no	Gen	M1,D2(X2,B2)	
48	LH	RX	4	no	Gen	R1,D2(X2,B2)	
49	CH	RX	4	yes	Gen	R1,D2(X2,B2)	
4A	AH	RX	4	yes	Gen	R1,D2(X2,B2)	
4B	SH	RX	4	yes	Gen	R1,D2(X2,B2)	
4C	MH	RX	4	no	Gen	R1,D2(X2,B2)	
4D	BAS	RX	4	no	Gen	R1,D2(X2,B2)	
4E	CVD	RX	4	no	Gen	R1,D2(X2,B2)	
4F	CVB	RX	4	no	Gen	R1,D2(X2,B2)	
50	ST	RX	4	no	Gen	R1,D2(X2,B2)	
51	LAE	RX	4	no	ESA	R1,D2(X2,B2)	
54	N	RX	4	yes	Gen	R1,D2(X2,B2)	
55	CL	RX	4	yes	Gen	R1,D2(X2,B2)	
56	O	RX	4	yes	Gen	R1,D2(X2,B2)	
57	X	RX	4	yes	Gen	R1,D2(X2,B2)	
58	L	RX	4	no	Gen	R1,D2(X2,B2)	
59	C	RX	4	yes	Gen	R1,D2(X2,B2)	
5A	A	RX	4	yes	Gen	R1,D2(X2,B2)	
5B	S	RX	4	yes	Gen	R1,D2(X2,B2)	
5C	M	RX	4	no	Gen	R1,D2(X2,B2)	
5D	D	RX	4	no	Gen	R1,D2(X2,B2)	
5E	AL	RX	4	yes	Gen	R1,D2(X2,B2)	
5F	SL	RX	4	yes	Gen	R1,D2(X2,B2)	
60	STD	RX	4	no	Flpt	R1,D2(X2,B2)	
67	MXD	RX	4	no	Flpt	R1,D2(X2,B2)	
68	LD	RX	4	no	Flpt	R1,D2(X2,B2)	
69	CD	RX	4	yes	Flpt	R1,D2(X2,B2)	
6A	AD	RX	4	yes	Flpt	R1,D2(X2,B2)	
6B	SD	RX	4	yes	Flpt	R1,D2(X2,B2)	
6C	MD	RX	4	no	Flpt	R1,D2(X2,B2)	
6D	DD	RX	4	no	Flpt	R1,D2(X2,B2)	
6E	AW	RX	4	yes	Flpt	R1,D2(X2,B2)	
6F	SW	RX	4	yes	Flpt	R1,D2(X2,B2)	
70	STE	RX	4	no	Flpt	R1,D2(X2,B2)	
78	LE	RX	4	no	Flpt	R1,D2(X2,B2)	
79	CE	RX	4	yes	Flpt	R1,D2(X2,B2)	
7A	AE	RX	4	yes	Flpt	R1,D2(X2,B2)	
7B	SE	RX	4	yes	Flpt	R1,D2(X2,B2)	
7C	ME	RX	4	no	Flpt	R1,D2(X2,B2)	
7D	DE	RX	4	no	Flpt	R1,D2(X2,B2)	

Op. code	Mnemo. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
7E	AU	RX	4	yes	Flpt	R1,D2(X2,B2)
7F	SU	RX	4	yes	Flpt	R1,D2(X2,B2)
86	BXH	RS	4	no	Gen	R1,R3,D2(B2)
87	BXLE	RS	4	no	Gen	R1,R3,D2(B2)
88	SRL	RS	4	no	Gen	R1,D2(B2)
89	SLL	RS	4	no	Gen	R1,D2(B2)
8A	SRA	RS	4	yes	Gen	R1,D2(B2)
8B	SLA	RS	4	yes	Gen	R1,D2(B2)
8C	SRDL	RS	4	no	Gen	R1,D2(B2)
8D	SLDL	RS	4	no	Gen	R1,D2(B2)
8E	SRDA	RS	4	yes	Gen	R1,D2(B2)
8F	SLDA	RS	4	yes	Gen	R1,D2(B2)
90	STM	RS	4	no	Gen	R1,R3,D2(B2)
91	TM	SI	4	yes	Gen	D1(B1),I2
92	MVI	SI	4	no	Gen	D1(B1),I2
93	TS	S	4	yes	Gen	D2(B2)
94	NI	SI	4	yes	Gen	D1(B1),I2
95	CLI	SI	4	yes	Gen	D1(B1),I2
96	OI	SI	4	yes	Gen	D1(B1),I2
97	XI	SI	4	yes	Gen	D1(B1),I2
98	LM	RS	4	no	Gen	R1,R3,D2(B2)
9A	LAM	RS	4	no	ESA	R1,R3,D2(B2)
9B	STAM	RS	4	no	ESA	R1,R3,D2(B2)
AF	MC	SI	4	yes	Gen	D1(B1),I2
B205	STCK	S	4	yes	Gen	D2(B2)
B219	SAC	S	4	no	ESA	D2(B2)
B222	IPM	RRE	4	no	Gen	R1
B224	IAC	RRE	4	yes	ESA	R1
B22D	DXR	RRE	4	no	Flpt	R1,R2
B24C	TAR	RRE	4	yes	ESA	R1,R2
B24D	CPYA	RRE	4	no	ESA	R1,R2
B24E	SAR	RRE	4	no	ESA	R1,R2
B24F	EAR	RRE	4	no	ESA	R1,R2
BA	CS	RS	4	yes	Gen	R1,R3,D2(B2)
BB	CDS	RS	4	yes	Gen	R1,R3,D2(B2)
BD	CLM	RS	4	yes	Gen	R1,M3,D2(B2)
BE	STCM	RS	4	no	Gen	R1,M3,D2(B2)
BF	ICM	RS	4	yes	Gen	R1,M3,D2(B2)
D1	MVN	SS	6	no	Gen	D1(L,B1),D2(B2)
D2	MVC	SS	6	no	Gen	D1(L,B1),D2(B2)
D3	MVZ	SS	6	no	Gen	D1(L,B1),D2(B2)
D4	NC	SS	6	yes	Gen	D1(L,B1),D2(B2)
D5	CLC	SS	6	yes	Gen	D1(L,B1),D2(B2)
D6	OC	SS	6	yes	Gen	D1(L,B1),D2(B2)
D7	XC	SS	6	yes	Gen	D1(L,B1),D2(B2)
DC	TR	SS	6	no	Gen	D1(L,B1),D2(B2)

Op. code	Mnemo. code	Inst. Type	Length in bytes	CC	Inst. class	Assembler format
DD	TRT	SS	6	yes	Gen	D1(L,B1),D2(B2)
DE	ED	SS	6	yes	Dcpt	D1(L,B1),D2(B2)
DF	EDMK	SS	6	yes	Dcpt	D1(L,B1),D2(B2)
F0	SRP	SS	6	yes	Dcpt	D1(L1,B1),D2(B2),I3
F1	MVO	SS	6	no	Gen	D1(L1,B1),D2(L2,B2)
F2	PACK	SS	6	no	Gen	D1(L1,B1),D2(L2,B2)
F3	UNPK	SS	6	no	Gen	D1(L1,B1),D2(L2,B2)
F8	ZAP	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
F9	CP	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
FA	AP	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
FB	SP	SS	6	yes	Dcpt	D1(L1,B1),D2(L2,B2)
FC	MP	SS	6	no	Dcpt	D1(L1,B1),D2(L2,B2)
FD	DP	SS	6	no	Dcpt	D1(L1,B1),D2(L2,B2)

7.4 Extended mnemonic operation code

To make work easier for programmers, the assembler has extended mnemonic operation codes at its disposal. These make it possible to represent conditioned branches mnemonically, including their branch masks. The assembler breaks down these extended mnemonic operation codes into the instructions BC or BCR and sets the mask accordingly.

For instructions with two meanings (e.g. minus/mixed), the instruction with the second meaning (mixed) is to be used following the TM instruction.

Assembler format with extended mnemonic operation code		yields inst.	yields mask,operand	Meaning
B	D2(X2,B2)	BC	15,D2(X2,B2)	Branch
BE	D2(X2,B2)	BC	8,D2(X2,B2)	Branch when Equal
BH	D2(X2,B2)	BC	2,D2(X2,B2)	Branch when High
BL	D2(X2,B2)	BC	4,D2(X2,B2)	Branch when Low
BM	D2(X2,B2)	BC	4,D2(X2,B2)	Branch when Minus/Mixed
BNE	D2(X2,B2)	BC	7,D2(X2,B2)	Branch when Not Equal
BNH	D2(X2,B2)	BC	13,D2(X2,B2)	Branch when Not High
BNL	D2(X2,B2)	BC	11,D2(X2,B2)	Branch when Not Low
BNM	D2(X2,B2)	BC	11,D2(X2,B2)	Branch when Not Minus/Mixed
BNO	D2(X2,B2)	BC	14,D2(X2,B2)	Branch when Not Overflow/Ones
BNP	D2(X2,B2)	BC	13,D2(X2,B2)	Branch when Not Plus
BNZ	D2(X2,B2)	BC	7,D2(X2,B2)	Branch when Not Zero/Zeroes
BO	D2(X2,B2)	BC	1,D2(X2,B2)	Branch when Overflow/Ones
BP	D2(X2,B2)	BC	2,D2(X2,B2)	Branch when Plus
BR	R2	BCR	15,R2	Branch Register
BRE	R2	BCR	8,R2	Branch Register when Equal
BRH	R2	BCR	2,R2	Branch Register when High
BRL	R2	BCR	4,R2	Branch Register when Low
BRM	R2	BCR	4,R2	Branch Register when Minus/Mixed
BRNE	R2	BCR	7,R2	Branch Register when Not Equal
BRNH	R2	BCR	13,R2	Branch Register when Not High
BRNL	R2	BCR	11,R2	Branch Register when Not Low
BRNM	R2	BCR	11,R2	Branch Register when Not Minus/Mixed
BRNO	R2	BCR	14,R2	Branch Register when Not Overflow/Ones
BRNP	R2	BCR	13,R2	Branch Register when Not Plus
BRNZ	R2	BCR	7,R2	Branch Register when Not Zero/Zeroes
BRO	R2	BCR	1,R2	Branch Register when Overflow/Ones
BRP	R2	BCR	2,R2	Branch Register when Plus
BRZ	R2	BCR	8,R2	Branch Register when Zero/Zeroes
BZ	D2(X2,B2)	BC	8,D2(X2,B2)	Branch when Zero/Zeroes
NOP	D2(X2,B2)	BC	0,D2(X2,B2)	No Operation
NOPR	R2	BCR	0,R2	No Operation Register

7.5 Powers of base 2

Value	Decimal representation	Hexadecimal representation	
2^{0}	1	1	
2^{1}	2	2	
2^{2}	4	4	
2^{3}	8	8	
2^{4}	16	10	
2^{5}	32	20	
2^{6}	64	40	
2^{7}	128	80	
2^{8}	256	1 00	
2^{9}	512	2 00	
2^{10}	1 024	4 00	
2^{11}	2 048	8 00	
2^{12}	4 096	10 00	
2^{13}	8 192	20 00	
2^{14}	16 384	40 00	
$2^{15}-1$	32 767	7F FF	
2^{15}	32 768	80 00	
$2^{16}-1$	65 535	FF FF	
2^{16}	65 536	1 00 00	
2^{17}	131 072	2 00 00	
2^{18}	262 144	4 00 00	
2^{19}	524 288	8 00 00	
2^{20}	1 048 576	10 00 00	
2^{21}	2 097 152	20 00 00	
2^{22}	4 194 304	40 00 00	
2^{23}	8 388 608	80 00 00	
$2^{24}-1$	16 777 215	FF FF FF	
2^{24}	16 777 216	1 00 00 00	
2^{25}	33 544 432	2 00 00 00	
2^{26}	67 108 864	4 00 00 00	
2^{27}	134 217 728	8 00 00 00	
2^{28}	268 435 456	10 00 00 00	
2^{29}	536 870 912	20 00 00 00	
2^{30}	1 073 741 824	40 00 00 00	
$2^{31}-1$	2 147 483 647	7F FF FF FF	
2^{31}	2 147 483 648	80 00 00 00	*)
$2^{32}-1$	4 294 976 295	FF FF FF FF	*)
-1	-1	FF FF FF FF	
-2	-2	FF FF FF FE	
-2^{15}	-32 768	FF FF 80 00	
-2^{31}	-2 147 483 648	80 00 00 00	

*) Unsigned 32-bit binary number.

7.6 Access to shared data in multiprocessor systems

Competing access by a number of programs (tasks/contingency processes) to shared data in memory must be carefully programmed to reduce the risk of concurrent access in multiprocessor systems. A **lock** is required for both read and write operations on shared data in order to prevent inconsistencies due to competing write accesses (a data item can also be treated as a lock).
The lock may be a byte, a word or a doubleword in length.
A **binary lock** that is one word long (also known as a lockword) may have the value 0 for 'free' or another value such as X'FFFFFFFF' for 'reserved'.
In a **count lock** a counter is incremented or decremented.

The terms **'safe read'**, **'safe write'**, **'safe instructions'** and **'safe operation'** are used below to mean
that no other processor can change a byte in the word or doubleword or a bit in the byte or word or doubleword while the read or write operation is in progress.

Only certain instructions are 'safe instructions' and therefore suitable for setting, resetting or querying locks.
Absolutely **safe instructions** on all systems are: **CS, CDS and TS**; see Chapter 3 and Appendix 7.6.1
Note, however, that these instructions take ten times longer to execute than an ST, for example, and even longer on some types of system.

Resetting a byte with **MVI** is a 'safe operation'.
Resetting a word or doubleword with the instructions **MVC, ST, STM, and STD** is a 'safe operation' only on 7.590 systems and higher. On downward-compatible systems (≤ 7.580), use the CS or CDS instruction to reset locks or clear shared memory spaces (not protected by a lock) longer than 1 byte (i.e. spaces of word or doubleword length).

Querying a byte with **CLI** is a 'safe operation'.
Querying a word or doubleword with **C, CL, CLC, IC, L, LM, LD, MVC** is a 'safe operation' on 7.590 systems and higher. On downward-compatible systems (≤ 7.580) queries on locks or shared memory spaces (not protected by a lock) longer than 1 byte (i.e. spaces of word or doubleword length) must be repeated, in order to ensure that no incorrect values have been read.

7.6.1 Setting locks

The following instructions are suitable for setting locks:

TS (1 byte, but only 2 values, = X'FF' and ≠ X'FF',
so suitable only for binary locks)
CS (1 word)
CDS (1 doubleword)

These instructions ensure that a second processor cannot access a memory location already being accessed in an update. The condition code can be queried for the original value of the corresponding operands in each of these instructions.

All other memory access instructions and those listed below are **not suitable**, because they do not exclude the possibility of simultaneous access by a second processor:

MVI, NI, OI, XI, MVC, NC, OC, XC, ST, STM, STC.

7.6.2 Resetting locks

The following instructions are suitable for resetting locks on 7.590 systems and higher:

- MV1 instruction for a byte
- ST and MVC instructions for a word
- STM, STD and MVC instructions for a doubleword.

As mentioned above, you should use the instructions CS, CDS, and TS on other systems. Since these commands ensure that write is a safe operation for the lengths listed in the table, their use in conjunction with TS / CS / CDS ensures unique memory assignments.

The preconditions for using the MVC instruction are:
the two operands must not overlap and both must be aligned on word or doubleword boundary (according to their length).

Setting a count lock is equivalent to decrementing the counter, so CD/CDS are the only valid instructions.

The instructions listed below are **not suitable** for resetting binary locks and lockwords. They require two memory access operations (read followed by write) and are not protected against intervening access by other processors:

NI, NC, OI, OC, XI, XC.

7.6.3 Querying locks

The following instructions are suitable for querying locks on 7.590 systems and higher:

- CLI and IC instructions for a byte
- C, CL, CLC, L and MVC instructions for a word
- LM, LD, CLC and MVC instructions for a doubleword.

As mentioned above, you should repeat the instructions (except CLC) on other systems. Conditions for using the CLC instruction are:
the two operands must not overlap and both must be aligned on word or doubleword boundary (according to their length).

The instructions listed below are **not suitable** for querying locks. They require two memory access operations (read followed by write) and are not protected against intervening access operations by other processors:

NI, NC, OI, OC, XI, XC, ZAP.

7.6.4 Examples

Counter locks

Example 1: Increment

```
      L     Rold,LOCK
   @CYCLE
     LR    Rnew,Rold
     AH    Rnew,=H'1'
   @WHEN EQ
     CS    Rold,Rnew,LOCK
     @BREAK
   @BEND
```

Example 2: Increment conditionally

```
   @CYCLE
     L     Rold,LOCK
   @WHEN GZ
     LTR   Rnew,Rold
   @AND  EQ
     AH    Rnew,=H'1'
     CS    Rold,Rnew,LOCK
   @BREAK
     @IF   LE
       LTR    Rold,Rold
     @THEN
       @PASS NAME=wait
     @BEND
   @BEND
```

It would be wrong to access the lock value a second time (i.e. `L Rnew,LOCK` instead of `LTR Rnew,Rold`), because there would no longer be any guarantee that the modified value and the reference value differ exactly by the intended difference - the lock could have been changed by a second program (task/contingency) in the time between the two access instructions.

In the second example, assignment with the `L Rold,LOCK` instructions must take place within the loop, because the CS instruction is not executed unless the first part of the query (Rold > 0) is true.

Resource management

Resources, e.g. memory elements (or entries) of the same size, need managing. Elements of this nature can be assigned dynamically to a table and released when necessary. Consequently, each element can be either 'free' or 'reserved'. The elements can be managed with the aid of a bit vector in which each bit is assigned to an element and acts as a status flag for the element in question:
Let us assume that Bit(N) = 0 means 'element(N) is free'

Bit(N) = 1 means 'element(N) is reserved'

For the sake of simplicity, let us assume that a maximum of 64 elements, i.e. 64 bits, is adequate for our purposes.

Example 3: Reserving an element

```
LOCK    DC      D'0'
        ...
OLD     DS      D
NEW     DS      D


        ...
        SLR     R4,R4
        LA      R5,1
        SLDL    R4,64-bitnum (bitnum 1...64)
        LM      R6,R7,LOCK
        @CYCLE
          LR    R8,R6
          LR    R9,R7
        @IF   NZ
          NR    R8,R4
        @OR   NZ
          NR    R9,R5
        @THEN
          LA    R15,reserved
        @EXIT

        @BEND
        LR   R8,R6
        LR   R9,R7
        OR   R8,R4
        OR   R9,R5
      @WHEN EQ
        CDS  R6,R8,LOCK

        @BREAK
        @BEND
```

Register pair (R4,R5) contains a bit set to 1 at the position that represents the element to be reserved; all other bits are 0. Register pair (R6,R7) contains the original contents of the bit vector 'LOCK'. Register pair (R8,R9) is the result of ORing (R6,R7) and (R4,R5), so its set bits are those set in (R6,R7), plus the bit set in (R4,R5). The OR result is protected and written into the 'LOCK' bit vector by the CDS instruction.

Example 4: Releasing an element (contrasting example)

```
LOCK   DC     D'0'
       ...
OLD    DS     D
NEW    DS     D


       ...
       SLR    R4,R4
       LA     R5,1
       SLDL   R4,64-bitnum  (bitnum 1..64)
       X      R4,=A(X'FFFFFFFF')
       X      R5,=A(X'FFFFFFFF')
       LM     R8,R9,LOCK          *
 /\/→                              *
       LM     R6,R7,LOCK          *
       ┌─── @CYCLE                *
       │      NR    R8,R4         *
       │      NR    R9,R5         *
       ├─── @WHEN EQ
       │      CDS   R6,R8,LOCK
       │
       ├──← @BREAK
       │      LR    R8,R6         @
       │      LR    R9,R7         @
       └─── @BEND
```

Register pair (R4,R5) contains a bit set to 0 at the position that represents the element to be released; all other bits are 1. Register pair (R6,R7) contains the original contents of the bit vector 'LOCK'. Register pair (R8,R9) is the result of ANDing (R6,R7) with (R4,R5), so its set bits are those set in (R6,R7) without the bit set in (R4,R5). The AND result is protected and written into the bit vector by the CDS instruction.

The error is due to the fact that the result of the comparison (R6,R7) and the base content for the modification (R8,R9) are fetched from memory in two steps. If a program interrupt occurs at the point marked /\/→, and if the doubleword 'LOCK' is modified thereby or if it is modified at this point by another processor, this change is revoked. An element just released by another program (process/task) is flagged as reserved or, even worse, an element just reserved by another program is released without good reason. Switching the instructions before and after the position marked by /\/→ would merely make an error more unlikely, but would not reliably exclude the possibility of error.

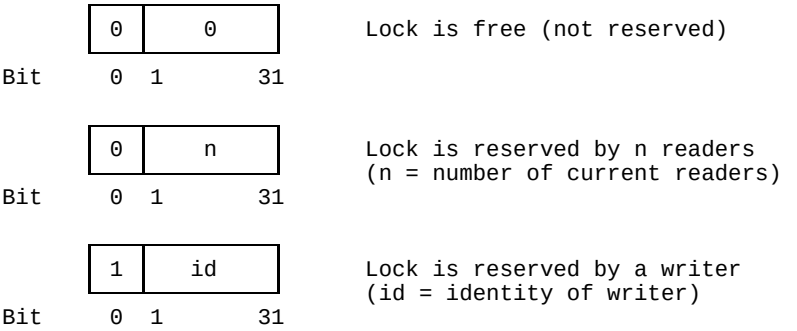
The way to solve the problem is to dispense with the second access to LOCK and to initialize (R8,R9) at the start of the loop from (R6,R7), as in example 3 above.

```
LM    R6,R7,LOCK
┌─ @CYCLE
│    LR   R8,R6
│    LR   R9,R7
│    NR   R8,R4
│    NR   R9,R5
```

Insert this code instead of the code string marked * above. The second code string marked @ is then unnecessary.

Reader-writer synchronization

In the simplest case, this requires a lockword capable of assuming the following statuses:



A reader must observe the following protocol:

```
Request read lock (GET_READ_LOCK)
Read
Release read lock (REL_READ_LOCK)
```

A writer must observe the following protocol:

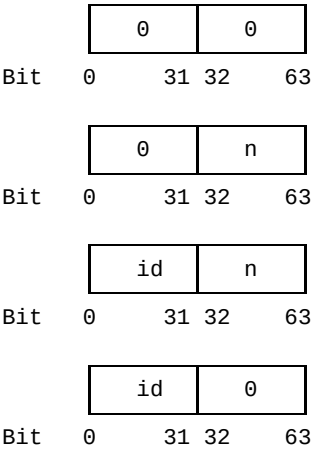
```
Request write lock (GET_WRITE_LOCK)
Write
Release write lock (REL_WRITE_LOCK)
```

The table below lists the status transitions implemented for the lockword by the various lock functions, with an indication as to whether a CS instruction is necessary (yes/no).

Function	Status transition	CS
GET_READ_LOCK	(0,n) -> (0,n+1)	yes
REL_READ_LOCK	(0,n) -> (0,n-1) condition: n>0, otherwise error	yes
GET_WRITE_LOCK	(0,0) -> (1,id)	yes
REL_WRITE_LOCK	(1,id) -> (0,0)	no

This mode of synchronization is not efficient because a writer may never be granted a write lock and therefore may never have an opportunity to write. This mode should be used only for readers who seldom request a read lock and do not reserve the lock for any length of time.

One way of achieving efficient synchronization is to use a lock doubleword that can assume the following statuses:



A reader must observe the following protocol:

```
Request read lock (GET_READ_LOCK)
Read
Release read lock (REL_READ_LOCK)
```

A writer must observe the following protocol:

```
Request write lock and inhibit read lock (GET_WRITE_LOCK_INHIBIT_READ)
Wait until read lock is released
Write
Release write lock and cancel read lock (REL_WRITE_LOCK_ADMIT_READ)
```

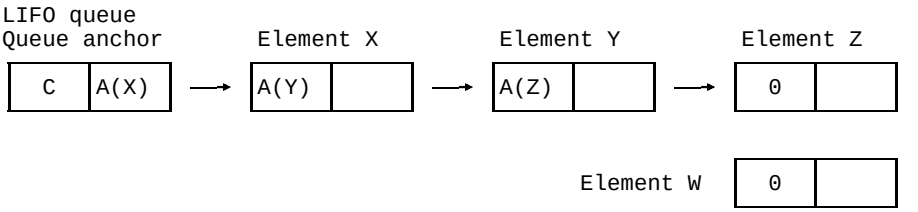
The table below lists the status transitions implemented for the lockword by the various lock functions, with an indication as to whether a CDS or CS instruction is necessary (yes/no).

Function	Status transition	CS/CDS
GET_READ_LOCK	(0,n) -> (0,n+1)	CDS
REL_READ_LOCK	(x,n) -> (x,n-1) x = 0 | id condition: n>0, otherwise error	CS lock.right_word
GET_WRITE_LOCK_...	(0,n) -> (id,n)	CS lock.left_word
REL_WRITE_LOCK_...	(id,0) -> (0,0)	no

Multiprocessor capability of queueing mechanisms

Multiprocessor capability is illustrated by reference to the input/output queuing mechanisms for a LIFO (last in first out) queue.

Let us assume that the queue has an anchor Q and contains the queued elements X, Y and Z. The free element W is to be placed in the queue at an opportune time.



If you want to dispense with lock protection while processing the queue (locking would impair performance to an unacceptable degree), you must use the CS instruction to update the queue.

When you place an element in the queue, the CS instruction ensures serialization of those processes that might want to place elements in the queue at the same time. This is because only the queue anchor need be evaluated to transfer the pointer to what was previously the first element into the link field of the elements to be queued, and to update the anchor itself.

When you unqueue an element, however, the CS instruction cannot exclude simultaneous access by a number of unqueueing operations. Note the following sequence of operations on the queue configuration outlined above:

Unqueue request by task/processor 1	Other tasks/ processors	Status of the queue
Initial status	:	Q → X → Y → Z
:	:	
L Ra,A(X) 1st element	:	
L Rc,A(Y) queued element	:	
:	:	
:	Unqueue element X	Q → Y → Z
:	Unqueue element Y	Q → Z
:	Queue element W	Q → W → Z
:	Queue element X	Q → X → W → Z
:	:	
CS Ra,Rc,queue anchor	:	Q → Y → ?
:	:	
Element X unqueued	Unqueue element Y	Q → ?

Task/processor 1 starts unqueueing element X. Once the address of the element to be unqueued and the link to the next element have been loaded, task/ processor 1 is delayed (e.g. due to a task change, processor is preempted, intervention by VM2000). Before the CS instruction can complete queue updating, other actions are performed on the queue by other tasks/processors.

If the CS instruction of task/processor 1 now completes, the constellation above would indicate a successful return, because the contents of the queue anchor have not changed since the previous load instruction. The value supplied to the anchor is element Y which is wrongly considered to be still queued. Element W, placed in the queue during the interruption, is lost along with the succeeding queue elements.

If this sequence of operations on the queue is followed by another unqueueing operation, element Y is assigned a second time and is thus used by two instances in parallel, but without mutual coordination. Errors that can result from this situation are overwrites and a linking loop (for example, if element Y is placed two times into the queue shown above).

Diagnosing these errors is extremely difficult, if not impossible, as they usually do not become apparent until much later. The problem can be circumvented by adding a counter to the queue anchor. The counter must count in one direction only (e.g. incrementally). The CDS instruction is used instead of CS, because the former can modify the queue anchor over a doubleword length in a way that is suitable for multiprocessors. The algorithm shown above must be modified as follows:

```
L    Ra,C                  Load current counter from queue anchor
L    Rb,A(X)               Localize address of foremost element
LA   Rc,1(,Ra)             Increment counter (wrap around)
L    Rd,A(Y)               Localize address of queued element

CDS Ra,Rc,queue anchor     Unqueue element
```

A counter that always shows the number of elements in the queue is not suitable for diagnostics, because in the queuing example above it would continue to appear unchanged like the queue element in the anchor.

References

[1] **ASSEMBH** (BS2000)
Reference Manual

[2] **Introductory Guide to XS Programming (for Assembler Programmers)** (BS2000)
User's Guide

[3] **BS2000/OSD-BC**
Executive Macros
User Guide

Index

C

Contents

User Guide - English

FUJITSU

Assembler Instructions (BS2000/OSD)

Reference Manual

Valid for
ASSEMBH V1.2

Edition May 1993 - new edition October 2016. Contains no technical changes.

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@ts.fujitsu.com

Certified documentation according to DIN EN ISO 9001:2008

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

All hardware and software names used are trademarks of their respective manufacturers.

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.
This document is an new edition of an earlier manual for a product version which was released a considerable time ago in which no changes have been made to the subject matter.
Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.
Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format …@ts.fujitsu.com.
The Internet pages of Fujitsu Technology Solutions are available at http://ts.fujitsu.com/...