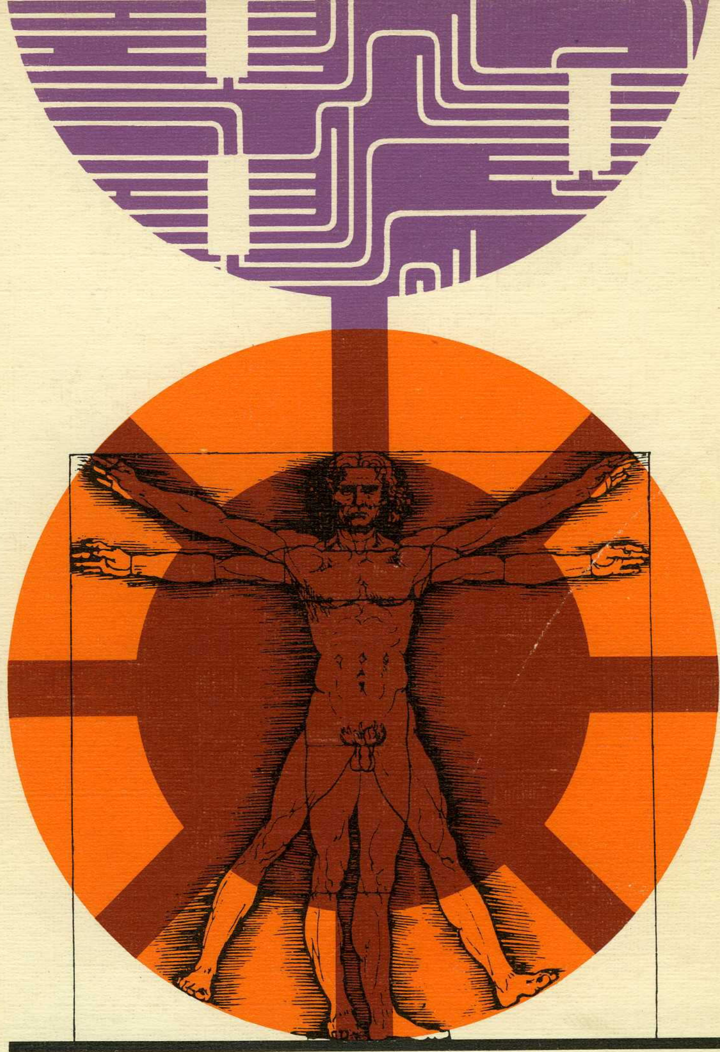


REFERENCE MANUAL

rc 4000®

DATAMATICS

RCSL No: 55-D1

RC 4000 COMPUTER

REFERENCE MANUAL

2. Edition

Edited by
Per Brinch Hansen

A/S REGNECENTRALEN
Copenhagen – June 1969

FOREWORD

The RC 4000 is a general-purpose digital computer, designed and manufactured by A/S Regnecentralen for real-time control, numerical computation, and administrative data processing. Its principal features are: direct addressing of a large internal store; integer and floating-point arithmetic; standardized connection of peripheral devices; program interruption; storage protection and privileged instructions.

This manual provides basic programming and operating information for programmers and users of the RC 4000 computer.

A summary of the RC 4000 is given in Chapter 1. Chapter 2 describes the considerations that guided the design of the computer. Chapters 3 to 9 contain specifications of word formats, storage addressing, arithmetic, multiprogramming features, and input/output control. Chapters 10 to 12 summarize the functions of the standard peripheral devices and the panels for operator control and engineering maintenance. Chapters 13 and 14 complete the picture with an exact definition of the instruction execution; the basic instruction cycle and all operations are described in an extended Algol language.

The main difference between the present second edition and the first edition of the reference manual (April 1967) is due to the extension of the RC 4000 computer with floating-point arithmetic, a new protection system, and a high-speed data channel. The speed of the machine has been increased by 25 percent.

Although the manual contains hints about the effective use of the instruction set no attempt has been made to teach programming techniques.

The function of peripheral devices is described in a separate manual.

CONTENTS

1. RC 4000 SPECIFICATIONS 9
2. DESIGN CONSIDERATIONS 11
   2.1. Word Length 11
   2.2. Register Structure 11
   2.3. Address Modification 11
   2.4. Monitor Control 12
   2.5. Input/Output Control 13
3. DATA AND INSTRUCTION FORMATS 15
   3.1. Data Formats 15
   3.2. Storage Addressing 15
   3.3. Working Registers 16
   3.4. Instruction Format 17
   3.5. Address Modify Instruction 18
4. INTEGER ARITHMETIC 19
   4.1. Number Representation 19
   4.2. Byte Arithmetic 19
   4.3. Multiplication and Division 20
   4.4. Overflow and Carry Indication 21
5. FLOATING-POINT ARITHMETIC 22
   5.1. Number Representation 22
   5.2. Arithmetic Operations 23
   5.3. Normalization and Rounding 23
   5.4. Precision Modes 24
   5.5. Underflow, Overflow, and Non-Normalized Operands 25
   5.6. Number Conversion 26
   5.7. Exact Arithmetic with Floating-Point Instructions 26
6. PROTECTION SYSTEM 28
   6.1. Storage Protection 28
   6.2. Privileged Instructions 29
   6.3. Summary of Protection System 29
   6.4. An Example of Protected Areas 30
7. INTERRUPTION SYSTEM 32
   7.1. Interruption Logic 32
   7.2. Interruption Conditions 33

8. LOW-SPEED DATA CHANNEL 35
8.1. Main Characteristics 35
8.2. Input/Output Instruction 35
8.3. Channel Operation 36
8.4. Disconnected and Busy Indication 37
8.5. Device Commands 37
8.6. Read 37
8.7. Sense 37
8.8. Write 38
8.9. Control 39
8.10. Summary of Low-Speed Channel 39
9. HIGH-SPEED DATA CHANNEL 41
10. STANDARD PERIPHERAL DEVICES 43
10.1. Console Devices 43
10.2. Interval Timer 43
11. OPERATOR CONTROL PANEL 44
11.1. Indicators and Control Keys 44
11.2. Machine Errors 44
11.3. Reset Control 44
11.4. Start Control 45
11.5. Autoload Control 45
11.6. Local/Remote Indication 46
12. TECHNICAL CONTROL PANEL 47
12.1. Operating Modes 47
12.2. Instruction Step Keys 47
12.3. Register Setting and Display 47
12.4. Microinstruction Selection and Display 48
12.5. Parity Control 48
13. INSTRUCTION SET 49
14. DEFINITION OF INSTRUCTIONS 51
14.1. Algol Notation 51
14.2. Register Structure 51
14.3. Elementary Operations 54
14.4. Control Panel Functions 55
14.5. Instruction Fetch Cycle 55
14.6. Protection Procedures 56
14.7. Arithmetic Procedures 57
14.8. Instruction Execution 58

APPENDIX 79
A.1. Reserved Storage Locations 79
A.2. Numeric Instruction Codes 80
A.3. Instruction Execution Times 81
INDEX 83

Chapter 1

RC 4000 SPECIFICATIONS

Implementation
Monolithic integrated circuits extensively used.

Internal Store
Magnetic core store with 1.5 usec cycle time.
Basic module of 16384 words. Direct addressing of up to 8 388 608 words. Each word contains 24 data bits, 3 protection bits, and 1 parity bit.

Backing Store
Magnetic drum or disk.

Working Registers
4 result registers of 24 bits each. Three of these also function as index registers. The registers are addressable as the first four words of the internal store.

Data Formats
12 bit bytes and 24 bit words for integer arithmetic.
48 bit double words for integer and floating-point arithmetic.

Instruction Format
24-bit single-address instruction. Address modification includes indexing, indirect addressing, and relative addressing.

Instruction Execution Times
1.5 to 4.5 usec typically (including access time).

Instruction Set
58 instructions.
Arithmetic includes add, subtract, multiply, and divide.
Data manipulation assisted by byte operations and word comparison.
Logical operations permit setting and testing of single bits.

Protection System
Privileged instructions and storage protection associated with a monitor mode ensure complete monitor control.

Interruption System
Program interruption system with 24 maskable priority levels.
Interrupt response time is 7.5 usec.

Input Output Control
Low-speed data channel for transfer of single words between character-oriented devices and working registers under program control.
High-speed data channel for transmission of blocks between block-oriented devices and the internal store simultaneous with program execution.

Chapter 2

DESIGN CONSIDERATIONS

This chapter describes some of the factors that influenced the design of the RC 4000.

2.1. Word Length

Arithmetic Operands. The basic arithmetic operand is a 24-bit word. This word length is sufficient for most integer arithmetic in process control applications. Double-length operands of 48 bits satisfy the requirements of engineering computation and administrative data processing.

Byte Handling. As a control computer, the RC 4000 must handle a large number of analog input data of from 10 to 12 bits each. Direct addressing of 12-bit bytes ensures efficient storage of these small integers. Byte handling is also a powerful tool in the manipulation of character strings encountered in file maintenance activities and program translation.

2.2. Register Structure

In earlier computers there is a sharp distinction between the accumulator (the register in which arithmetic operations are performed) and index registers (used solely to modify the address part of the instruction). This register structure often makes programming awkward. Since all operations destroy the previous contents of the accumulator the programmer is forced to make numerous storage operations in order to save and restore intermediate results. Empty transfers to the store are also required when an index register must be modified by the contents of the accumulator.

The RC 4000 eliminates this deficiency with four working registers, three of which also function as index registers. By extending the number of accumulators to four and removing the distinction between accumulators and index registers, the full instruction set becomes available for immediate address modification, while empty transfers of registers to the store are considerably reduced.

2.3. Address Modification

The efficiency of computer programs is closely connected to the handling of address fields within instructions. The two main problems here are program relocation and table look-up.

Program Relocation. The ability to relocate programs in the internal store is vital in a computer in which the library of programs is kept on a backing store and only brought to the internal store when active. Normally it is not possible to predict the combinations of programs and data in the store when a program must be loaded. The programs therefore cannot expect to be loaded into their previous storage areas, but must be relocated to new areas currently available. Efficient relocation requires that programs can be written in such a way that their execution is independent of their location. In the RC 4000 this is achieved by a bit in the instruction format specifying relative addressing. This implies that the address part of the instruction is interpreted relative to its current location in the store.

Table Look-Up. The purpose of data processing is to transform a set of data into a result according to certain rules. In a computer with an addressable store, one of the most general ways of specifying the rules of transformation is to use a set of tables. Each item of data to be transformed is converted to an address that is used to look up a table to extract a new data value or the address of an action to be performed. The requirement that addresses can be modified by the values of data being processed is met efficiently through the use of index registers. The RC 4000 instruction format permits the programmer to specify a modification of the address part with the contents of a working register.

2.4. Monitor Control

In the process control applications the computer usually has a number of concurrent tasks to perform. These tasks must be repeated at regular intervals if real-time control of the plant is to be maintained. In such a multiprogramming system it is vital that erroneous programs are prevented from interfering destructively with other programs. The different tasks must therefore be coordinated by a monitor program that has complete control of the system. In the RC 4000, monitor control is guaranteed by (1) storage protection, (2) privileged instructions, and (3) program interruption.

Storage Protection. An erroneous program may attempt to destroy data or instructions within other programs. The protection system for the RC 4000 permits mutual storage protection of up to 8 programs including a monitor program. The protection is achieved by providing each storage word with a protection key of 3 bits and by introducing a protection register to specify which protection keys are accessible within the current program. Although a subordinate program can read a protected word, only the monitor program can alter its contents. Any attempt to execute or destroy the contents of a protected location leads immediately to an interruption of the current program. The protection sphere of a program can be changed immediately by loading the protection register with another bit pattern. The system is thus well suited to multiprogramming applications in which the computer must switch rapidly among a number of independent programs.

Privileged Instructions. Further system protection is achieved by privileged instructions that can only be executed within the monitor program. These instructions include all input/output functions as well as control of the interruption system and storage protection.

Program Interruption. A computer used in process control must respond quickly to exceptional internal and external events. In the RC 4000 this is achieved through a program interruption system that can register up to 24 signals simultaneously. Any of these signals interrupts the current program immediately and starts the monitor program.

2.5. Input/Output Control

The design of the input/output control is based on the following principles:

(1) The computer makes no restrictions on the kinds of peripheral devices that can be connected to it.

(2) Program execution continues while input/output operations are in progress.

(3) Exceptional events occurring in input/output operations are completely under program control and will never cause a machine stop.

To permit future expansion of the array of peripheral equipment the connection of devices has been standardized in such a way that the computer is unaware of the types of devices attached to it. First, the RC 4000 handles all input/output operations by a single instruction that identifies devices by addresses only. Second, all devices are connected to a standard data channel, capable of transferring 24 bits of data.

In real-time applications it is unacceptable to halt computation while a data transfer is in progress. Accordingly, the peripheral devices release the RC 4000 as soon as an input/output operation has been initiated. The computer then continues the program, while the device in question completes its operation independently.

When the computer attempts to initiate an input/output operation, the peripheral device may answer by a rejection, indicating that it is occupied with another operation. This information is made available in an exception register, which can be sensed by the program to decide an alternative course.

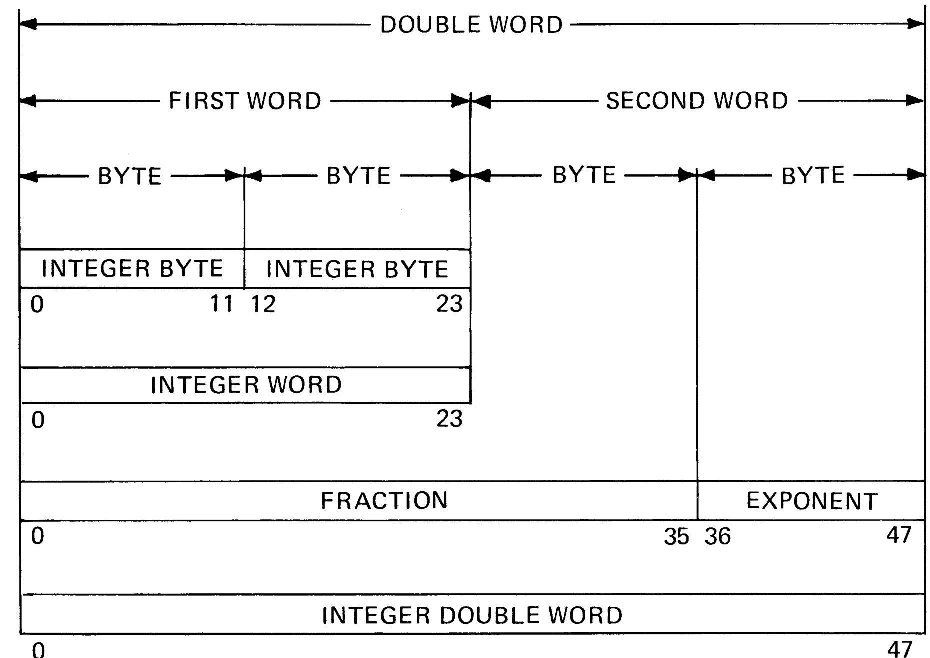
When a data transfer is completed, the program can request the device to deliver information about exceptional conditions that occurred during the transfer. This is necessary because a real-time system cannot rely on the operator to discover and react to such emergencies.

Chapter 3

DATA AND INSTRUCTION FORMATS

3.1. Data Formats

The data structure of the RC 4000 is shown in the following figure:



The basic arithmetic operand is an integer of 24 bits. Small integers are packed with two bytes per word. The 12-bit bytes are directly addressable. Double words are used to represent integers of 48 bits and floating-point numbers with 36-bit fractions and 12-bit exponents.

3.2. Storage Addressing

Storage addresses are always expressed as **byte addresses.** The byte locations are numbered consecutively starting with zero.

In **word operations,** the right-most bit in the effective address is ignored. Thus it is irrelevant whether a word operation refers to the left or right half of a word.

In **double-word operations,** the right-most bit in the effective address is ignored. The word thus specified is the second word of the operand.

3.3. Working Registers

The register structure of the RC 4000 consists of four working registers of 24 bits each. In each instruction, one of these registers is specified as the **result register.** Three of the registers also function as **index registers.** The current index register is selected by the instruction format.

The working registers are **addressable** as the first eight bytes (or four words) of the internal store. The programmer can therefore perform operations directly between two registers by specifying a storage address between 0 and 7. It is also possible to execute instructions stored in the working registers. Like the rest of the storage words each working register is supplied with its own **protection key** (see Section 6.1.).

BYTE ADDRESS		
0	24 BITS	WORKING REGISTER 0
2	24 BITS	WORKING REGISTER 1
4	24 BITS	WORKING REGISTER 2
6	24 BITS	WORKING REGISTER 3

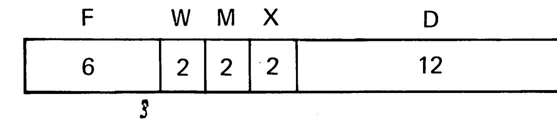
Two adjacent working registers can be used to hold a **double length operand** of 48 bits. In double-length operations, the four registers are connected cyclically as follows:

W3 **concatenate** W0
W0 **concatenate** W1
W1 **concatenate** W2
W2 **concatenate** W3

These connections are established by specifying the second register W0, W1, W2, and W3, respectively, in the instruction format.

3.4. Instruction Format

The **instruction** format is divided into an **operation byte** and an **address byte** of 12 bits each:



The operation byte specifies 64 basic operations in the **F field** of 6 bits. One of the four working registers is specified as the result register in the **W field** of 2 bits. The current index register is selected by the **X field** of 2 bits. Only working registers W1, W2, and W3 act as index registers (X field = 0 indicates no indexing).

A truncated address of 12 bits (the **D field**) specifies a displacement from −2048 to +2047 bytes within the program. This is adequate for the majority of addresses. It is, however, insufficient for direct addressing of the entire store. A full address of 24 bits is formed by means of the displacement D in connection with the contents of an index register X and the contents of the instruction counter IC. The generation of the effective address A is controlled by the address mode field M as follows:

M = 00	A = X + D
M = 01	A = word (X + D)
M = 10	A = X + IC + D
M = 11	A = word (X + IC + D)

In the address calculation, the displacement is treated as a 12-bit signed integer that is extended towards the left to 24 bits, before being added to the index register and the instruction counter. In the final addition of X, IC, and D, overflow is ignored.

The address modes 01 and 11 permit **indirect addressing** in one level. The indirect address fetched from the store is assumed to be a full address of 24 bits.

The address modes 10 and 11 modify the indexed displacement with the current load address of the instruction. This permits relocation of programs during loading.

In storage access operations, the effective address is treated as an unsigned integer of 24 bits. The upper limit to the expansion of the store is therefore 16 777 216 bytes. In an installation in which only a part of the maximum storage capacity is available, reference to a nonexistent storage location will cause a program interruption (see Section 7.2.).

(At this point it is suggested that the reader studies Chapter 1 in the manual of the **Slang Assembler** in order to become familiar with the notation used in the following programming examples).

3.5. Address Modify Instruction

The instruction **modify next address** deserves special mention in connection with the possibilities of address modification. This instruction modifies the displacement in the following instruction by its own effective address. We shall use the mnemonic operation codes defined in Chapter 13 to illustrate three uses of this instruction: (1) direct indexing with the contents of any storage location, (2) multiple indexing with the sum of two or more working registers, and (3) multi-level indirect addressing.

The possibility of using any storage location as an index register is illustrated by the following example:

```
AM (X1 + D1)
JL  D2
```

The effective address of the AM instruction is A1 = word (X1 + D1). This is used to modify the displacement D2 in the following JL instruction to produce an effective address A2 = word (X1 + D1) + D2.

A series of AM instructions can be used to modify an instruction with the sum of several index registers. The second example shows the actual instructions to the left and their effective addresses to the right:

```
AM X1 + 0    ;  A1 = X1
AM X2 + 0    ;  A2 = X1 + X2
JL  X3 + D3  ;  A3 = X1 + X2 + X3 + D3
```

The third example illustrates the use of the AM instruction to obtain multi-level indirect addressing:

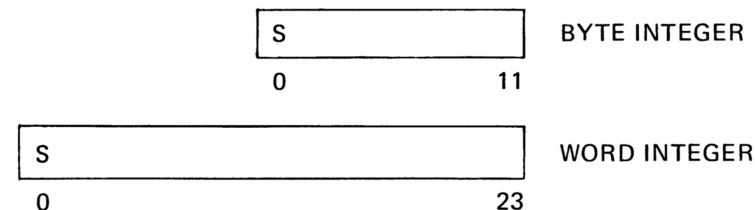
```
AM (X1 + D1);  A1 = word (X1 + D1)
AM (0)      ;  A2 = word (A1)
AM (0)      ;  A3 = word (A2)
etc.
```

Chapter 4

INTEGER ARITHMETIC

4.1. Number Representation

The standard arithmetic operands are signed integers of 12 and 24 bits:



Positive numbers are represented in true binary form with a zero in the sign bit. Negative numbers are represented in the **two's complement** notation with a one in the sign bit. The two's complement of a number may be obtained by inverting each bit in the number and adding 1 to the rightmost bit.

The main virtue of the complement notation is the simple handling of signed operands. An addition or subtraction of two data-words is simply performed as if they both were unsigned binary numbers of 24 bits.

The complement notation also facilitates the handling of small integers represented by 12-bit bytes. A small integer can be extended to the standard form of 24 bits simply by a duplication of the sign bit toward the left; conversely, when the high-order digits of a small integer are elided, the leading digit of the truncated integer still reflects the sign properly.

4.2. Byte Arithmetic

A signed integer represented by a 12-bit byte must be confined to the following **range**:

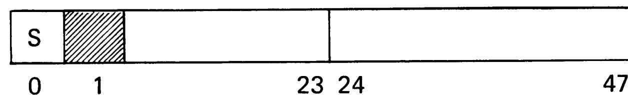
$$-2**11 = -2048 <= \text{integer byte} <= 2047 = 2**11-1$$

The instruction **load integer byte** serves to extend a signed 12-bit byte toward the left to 24 bits, as it is placed in a working register. The arithmetic instructions **add** and **subtract integer byte** perform addition to or subtraction from a working register with a byte fetched from the store and extended to 24 bits. The instruction **store half register** stores the right-most 12 bits of a working register in a byte.

The sign extension of byte operands makes it possible to perform integer arithmetic with mixed 12-bit and 24-bit operands.

4.3. Multiplication and Division

Integer **multiplication** of the contents of a working register with the contents of a storage word produces a double-length product that is placed in a double register of 48 bits with the sign bit at the extreme left:

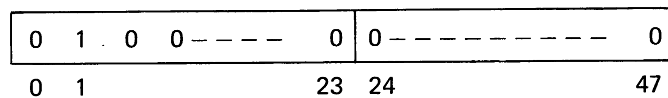


A double-length product will normally consist of a sign plus a most 46 digits. In this case, bit 1 in the double register will be identical with the sign bit.

The only exception to this occurs in the multiplication of two maximum negative numbers:

(–2**23)*(–2**23)=2**46

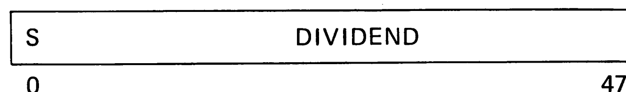
This result will be represented as shown below:

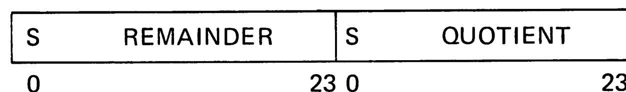


It should be noted that in this representation of double-length integers, bit 24 does not function as a sign bit, but contains a significant digit.

The contents of a double register can be **divided** by the contents of a storage word. The **dividend** is then replaced by a 24-bit **remainder** in the left-hand register and a 24-bit **quotient** in the right-hand register. A non-zero remainder satisfies the following requirements:

(1) dividend = divisor * quotient + remainder
(2) 0 < abs (remainder) < abs (divisor)
(3) sign (remainder) = sign (dividend)





4.4. Overflow and Carry Indication

Arithmetic operations indicate a normal or an exceptional result by setting the right-most 2 bits of a 3-bit register, called the **exception register.** This register can be tested by a single instruction, **skip if no exceptions.**

After a normal result, exception bits 22 and 23 are set to zero. An **integer overflow** will set exception bit 22 to one and provoke a **program interruption** as defined in Section 7.2. An overflow condition is recognized in the following situations.

(1) The result of an addition, subtraction, or division exceeds the range of a 24-bit signed integer:

–2**23=–8 388 608<= integer word< = 8 388 607 = 2**23–1

(Note that multiplication can never produce overflow).

(2) The instruction **load address complemented** specifies complementation of the maximum negative number: –(–2**23) = 2**23.

(3) One or more significant digits are lost during arithmetic shifts toward the left. (The shift instructions test overflow conditions after each single-bit shift).

If overflow occurs in division the dividend remains unchanged in the working registers. All other arithmetic operations deliver the result moduls 2**24 after an overflow.

Exception bit 23 is set, when addition or subtraction produces a **carry** from the sign position. This indicates that the result interpreted as an unsigned integer of 25 bits exceeds 2**24 – 1 = 16 777 215. The carry indication simplifies the programming of multiple-length addition and subtraction.

Thus, the exception register has the following meaning after an integer arithmetic operation:

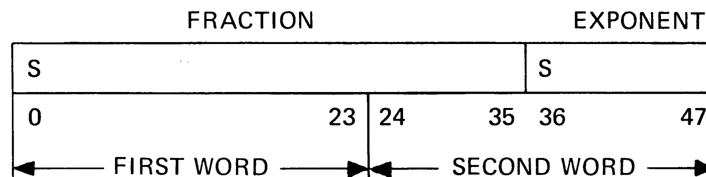
exception bit:
21 (unchanged)
22 integer overflow
23 integer carry

Chapter 5

FLOATING-POINT ARITHMETIC

5.1. Number Representation

A floating-point number F = fraction * 2 ** exponent is stored in a double word or a pair of working registers:



The **address** of a floating-point number refers to the second word of the storage operand. The **working register field** within a floating-point instruction refers to the second word of the register operand.

The left-most 36 bits of a floating-point number represent a signed, **normalized fraction** in the **two-s complement** notation. The right-most 12 bits is a signed **exponent** also in the two-s complement form.

The **range** of floating-point numbers is the following:

−1 * 2**2047 <= F <	−0.5 * 2**(−2048)	F negative
F =	0 * 2**(−2048)	F zero
0.5 * 2**(−2048) <= F <	1 * 2**2047	F positive

or approximately:

10**(−616) < abs(F) < 10**616

The relative **precision** of a floating-point number is 2**(−35) / abs(fraction) which lies between 2**(−35) = 3 * 10**(−11) and 2**(−34) = 6 * 10**(−11).

The left-most two bits of a normalized fraction are 01 and 10, respectively, for positive and negative numbers.

The floating-point **zero** is represented by the fraction 0 and the exponent −2048.

Accordingly, the **sign** or zero value of a floating-point number may be determined by examining its first word only. This can be done by means of the instructions **skip if register high, low, equal, or not equal** using the effective address zero as a comparison operand.

As an example, consider a floating-point number with the address F0. The following instructions will load the floating-point number in w0 and w1 and test whether it is negative:

```
DL W1 F0
SH W0 −1 ;  if F0 < 0
JL     G0 ;  then goto G0;
```

5.2. Arithmetic Operations

Before an arithmetic operation, the fractions are placed left-justified in anonymous 38 bit registers and extended to the right with two zeroes. The positions are numbered 0 through 37 in these registers.

Addition and **subtraction** require an alignment of radix points. This is done by shifting the fraction with the smaller exponent to the right a number of positions equal to the difference in exponents. Bits shifted out of the 38 bit register are thrown away. After alignment, the addition or subtraction of the fractions is performed, and the larger exponent is attached to the result. The resulting fraction is normalized and rounded as described in Section 5.3.

Multiplication is performed by addition of the exponents and multiplication of the fractions. The fraction product of 38 bits is formed by repetition of an add-and-shift cycle. Bits shifted out of the 38 bit register are thrown away. Normalization and rounding of the resulting fraction proceeds as for addition and subtraction, see Section 5.3.

Division is performed by subtraction of the exponents and division of the fractions. The fraction quotient of 36 bits is formed by the non-restoring division method. The shift-and-add(subtract) cycle is repeated until the quotient is normalized. The exponent is adjusted by adding 35 initially and subtracting 1 per cycle. Rounding of the quotient is performed as described in Section 5.3. The remainder is thrown away.

5.3. Normalization and Rounding

If the resulting 38 bit fraction is zero, a floating-point zero with exponent −2048 is delivered as final result.

A non-zero fraction is **normalized** either by left shifts to eliminate leading sign bits or by a single right shift to correct for overflow of the fraction. The exponent is decreased (increased) by the number of left (right) shifts performed.

A non-zero, normalized fraction is **rounded** by adding 1 in bit 36. After rounding, the fraction may require normalization once more before the high-order 36 bits and the exponent are delivered as the final result.

The maximum value of the **rounding error** is 0.5 in the least significant position of the 36 bit fraction of the result.

For **addition** and **subtraction** this may be seen as follows: Consider the 36 bit fractions f1 and f2 to be exact, f1 being the fraction of the larger operand. If the exponents differ less than three, f2 is shifted at most two positions and retains all significant bits in the 38 bit register. If the exponents differ more than two, f2 and the resulting fraction satisfies the following inequalities:

abs(f2 shifted) <= 1 * 2**(−3)

abs(f1 + or − f2 shifted) >= 0.5 − 1 * 2**(−3) = 3/8

Thus, at most one left shift is required to normalize the result. If fraction overflow occurs, normalization requires a single right shift. In both cases, the result contains at least 37 significant bits, and rounding to 36 bits can at most cause an error of 0.5. This is also true in the special cases requiring re-normalization.

After **multiplication**, the product of the fractions lies in the interval:

0.25 <= abs(f1 * f2) <= 1

and may thus require one left shift for normalization. Again, the result contains at least 37 significant bits before rounding takes place.

After **division,** rounding of the resulting fraction is performed by adding 1 in bit 35 if bit 36 is 1. Since rounding to 36 bits is performed on a normalized quotient of 37 bits, it follows that the maximum error is 0.5.

5.4. Precision Modes

The arithmetic operations may be performed in two modes called full and low precision. The **significance** of a numerical result may be estimated by performing the same computation in both modes and comparing the results.

In the **full precision mode,** the result is computed with a 36 bit fraction correctly rounded as described above.

In the **low precision mode,** the result is first computed and rounded as in the full precision mode, but in the final result the fraction bits 34 and 35 are set equal to bit 33.

The precision mode is selected by setting bit 21 in the **exception register**: 0 = full precision, 1 = low precision.

The effect of the low precision mode is that of working with floating-point numbers with a 33 bit fraction with practically the same error characteristics as the full precision arithmetic.

This can be shown as follows: the rounding of the eight possible values of bits 33, 34 and 35 to either 000 or 111 introduces errors of 0, −1/8, −2/8, −3/8, 3/8, 2/8, 1/8, and 0, respectively, in the least significant position of the 33 bit fraction. Assuming these eight values occur with equal probability, obviously the mean and maximum errors are 0 and 3/8, respectively. If we further assume an operand distribution such that the error from rounding in bit 36 has the mean value 0, then the mean and maximum total errors of the low precision fraction becomes 0 + 0 = 0 and (3 + 0.5)/8 = 0.44, respectively.

Now let F33 and F36 be the results of a given computation performed in the two modes. If the exact result is denoted by F we have:

F = F33 + error33 = F36 + error36

and hence:

F36 − F33 = error33 − error36

Assuming that errors in the full and low precision modes are uniformly distributed with mean values 0 and maximum values 0.5 and 3.5, respectively, there is a high probability that abs(error33) >> abs (error36), which justifies the approximation:

F36 − F33 = error33

This means that the identical digits in the two results in most cases can be regarded as the significant digits of F33. This has been verified in a limited number of numerical experiments.

The operation times are the same in the full and low precision modes.

5.5. Underflow, Overflow, and Non-Normalized Operands

Underflow and overflow occur when the exponent of the final result (after normalization, rounding, and re-normalization) is less than −2048 or greater than 2047, respectively. This will set bit 22 in the **exception register** to one and provoke a **program interruption** as defined in Section 7.2.

After **underflow** or **overflow,** the fraction is correct while the exponent is taken modulo 4096. Thus, if the sign of the resulting exponent is negative, the interrupt was caused by overflow, otherwise by underflow.

Division by zero leaves the register operand unchanged and gives an interrupt; this is also true if zero is divided by zero.

Considering the enormous range of floating-point numbers, both underflow and overflow will usually be an indication of a programming mistake.

It is not checked whether operands are correctly normalized floating-point numbers. If a floating-point operation is carried out on **non-normalized numbers** it will in some cases give a non-normalized result.

The exception register has the following meaning after a floating-point arithmetic operation.

exception bit 21: low precision (unchanged)
22: floating-point underflow or overflow
23: 0

5.6. Number Conversion

The instruction **convert integer to floating** converts a 24 bit integer stored in a working register to a 48 bit floating-point number stored in a pair of working registers consisting of the register specified in the instruction and the preceding one. The effective address A of the instruction is used as a signed **scaling factor.** Thus, the value of the floating-point number becomes:

integer * 2**A

Program interruption with bit 22 of the **exception register** set to one occurs if the resulting exponent exceeds the 12 bit range.

The instruction **convert floating to integer** converts a 48 bit floating-point number stored in a pair of working registers to a 24 bit rounded integer stored in the register specified in the instruction. The effective address A of the instruction is used as a signed **scaling factor.** Thus, the value of the integer becomes:

round(floating-point number * 2**A)

Program interruption with bit 22 of the **exception register** set to one occurs if the resulting integer exceeds the 24 bit range.

The conversion instructions do not distinguish between low and full precision because they always convert to and from a much lower precision of 24 bits.

If the **real** F0 and the **integer** I0 are two Algol variables, the assignment statement I0:= F0 can be performed by the instructions:

```
DL W1 F0 ; W0W1: = F0
CF W1 0  ; W1: = round(W0W1 * 2**0)
RS W1 I0 ; I0 : = W1
```

The assignment F0:= I0 may be performed in a similar way.

Since the CF instruction rounds off the result the Algol function **entier**(F0) may be performed by subtracting 0.5 before the conversion:

```
DL W1 F0 ;W0W1:= F0
FS W1 F1 ;W0W1:= W0W1 – 0.5
CF W1 0  ;W1:= round(W0W1 * 2**0)
```

5.7. Exact Arithmetic with Floating-Point Instructions

The floating-point arithmetic may be used to simulate exact arithmetic with 35 bits integers in the following sense: as long as operands and results only assume integer values in the **range**

–2**35 <= F < 2**35

any floating-point operation gives the **exact integer result.** All integers in this

range can be represented exactly as floating-point numbers, and since the error in each operation cannot exceed 0.5 in the 36th fraction bit the error must be zero.

While **addition, subtraction,** and **multiplication** of the integer values automatically give integer results, it is often necessary to modify a floating-point **quotient** to obtain an integer value. If the absolute value of the quotient does not exceed 2**34 the correctly rounded integer quotient may be obtained by adding and subtracting the floating-point number 2**34. In Algol the real quotient F0 may be rounded by the statement:

F0:= F0 + 2**34 – 2**34

This works because the addition will shift the fraction of F0 to the right until the last retained bit corresponds to the integer unit position of F0.

The integer division in Algol defined by:

F0//F1 = sign(F0/F1) * entier(abs(F0/F1))

may be simulated in floating-point arithmetic by the following statements:

```
Q:= F0/F1;
Q:= if Q>=0 then (Q – 0.5) else (Q + 0.5);
Q:= Q + 2**34 – 2**34;
```

Chapter 6

PROTECTION SYSTEM

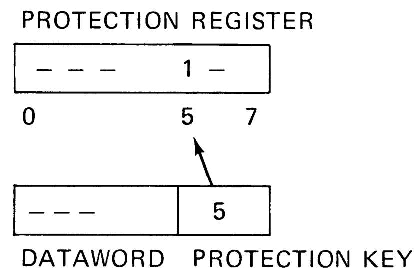
6.1. Storage Protection

The RC 4000 is designed to operate as a **multiprogramming** system under the control of a **monitor program.** Monitor control of the RC 4000 is guaranteed by **storage protection** and **privileged instructions.**

Each storage word is provided with a **protection key** of 3 bits. It is thus possible to identify eight different storage areas which can be prevented from destroying one another. The setting of the protection keys is controlled by the monitor program. The protection keys do not enter the working registers in the data operations. They are used only by the control unit to test whether the current program is allowed to **alter** or **jump** to a given storage word. Attempts to violate storage protection cause **program interruption.**

A **protection register** of eight bits specifies the storage areas which may be altered or entered by the current program. Each bit in this register corresponds to one of the eight different protection keys. In store and jump operations, the protection key of the addressed word is used as an index to select a bit within the protection register. This bit defines whether the storage word is protected against the current program.

As an example, a program may attempt to store the contents of a working register into a storage word in which the protection key has the value 5. In this case, the control unit tests bit number 5 in the protection register. If the selected bit is zero the storage operation will be executed, otherwise the current program will be interrupted.



In this system, the monitor can change the protection sphere of a subordinate program immediately by loading the protection register with another bit pattern.

The **protection key zero** is reserved for the monitor program, and bit 0 in the protection register is permanently equal to one. Thus the monitor is always protected from being destroyed by subordinate programs. It is also protected against erroneous calls i.e. jumps that enter the monitor at arbitrary points.

A call of the monitor program within a subordinate program is possible now only by provoking a program interruption (e.g. by attempting a direct jump to the monitor). Prior to this, information about the desired entry can be loaded into a working register. The program interruption transfers control to a fixed point in the monitor, which then decides whether the entry is correct or not.

6.2. Privileged Instructions

It is highly desirable that the protection status of the internal store can be controlled by the monitor. This has been achieved by two instructions, **load protection register** and **store protection key,** which can only be executed within the monitor, i.e. the interrupt response program. This concept of **privileged instructions** has been further extended to prevent subordinate programs from accidently seizing control from the monitor. First, subordinate programs cannot change the status of the interrupt system (e.g. by disabling it permanently). Second, it is impossible for a subordinate program to monopolize an input/output device needed by other programs. The following classes of instructions are therefore executable within the monitor only:

Storage Protection Control:
- LOAD PROTECTION REGISTER
- STORE PROTECTION KEY

Program Interruption Control:
- LOAD MASK REGISTER
- CLEAR INTERRUPT BITS
- JUMP WITH INTERRUPT ENABLED
- JUMP WITH INTERRUPT DISABLED

Input/Output Control:
- INPUT/OUTPUT
- AUTOLOAD WORD

6.3. Summary of Protection System

The protection system can be summarized as follows:

A **program interruption** sets the control unit in the **monitor mode** and starts a monitor program at a well-defined point. In the monitor mode, all instructions

can be executed as long as they are fetched from protected storage words. It is also allowed within the monitor to alter protected storage words.

The control unit returns to the **task mode,** when the first unprotected instruction is executed. In the task mode, program interruption results if the following is attempted:

(1) Executing a privileged instruction.

(2) Storing into a protected location.

(3) Jumping to a protected location (by explicit branching or by sequential program execution).

The protection status of a storage word is defined by the value of its protection key in connection with the corresponding bit in the protection register.

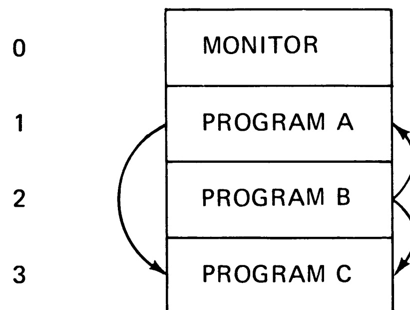
6.4. An Example of Protected Areas

As an example, consider a multiprogramming system in which the internal store is divided between a monitor program and three subordinate programs A, B, and C. The protection keys of the four program areas are 0, 1, 2, and 3, respectively.

Let the protection spheres of these programs be defined as follows:

1) Neither A, B, nor C have access to the monitor area.
2) A, B, and C of course have access to their own areas.
3) Furthermore, A can access C, while B can access both A and C.

PROTECTION KEY



Each time the monitor transfers control from one program to another, it loads the protection register with one of the following bit patterns:

	PROTECTION REGISTER
PROGRAM A	1 0 1 0 – – – –
PROGRAM B	1 0 0 0 – – – –
PROGRAM C	1 1 1 0 – – – –
	0 1 2 3 – – – 7

Chapter 7

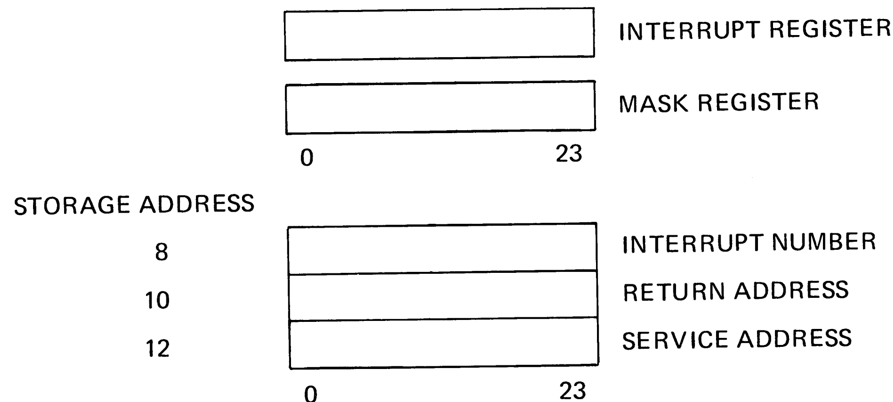
INTERRUPTION SYSTEM

7.1. Interruption Logic

The **program interruption** system permits an automatic switching from the current sequence of instructions to another sequence in immediate response to specific internal or external events.

The interruption system performs the following functions: (1) Collection of interrupt signals, (2) Interrogation of interrupt signals, (3) Selection among competing interrupt requests, (4) Saving of return information, and (5) Branching to the interruption program.

The RC 4000 can collect up to 24 interrupt signals in an **interrupt register.** The monitor has selective control over these interrupt lines by means of a **mask register.** For each of the 24 interrupt lines a mask register bit defines whether an interrupt request will be honoured (mask bit =1) or ignored (mask bit =0). The interrupt register is interrogated once in every instruction cycle. If any of the masked interrupt bits are set, the contents of the **instruction counter** will be stored in storage word 10, before **branching** to an address kept in storage word 12. The problem of simultaneous interrupt signals is solved by selecting the left-most signal first. This is done by turning the interrupt bit off and storing its register position as an integer (0 – 23) in storage word 8. The interruption program uses this **interrupt number** to branch to a specific service routine. To facilitate the branching, the interrupt number is stored as a word address, with the unit position in bit 22 and with bit 23 equal to zero.



Only the instruction counter is stored as return information about the interrupted program. The interruption program is responsible for saving and restoring the contents of the **working registers** and the **exception register.**

The entire interruption system can be **disabled** for short intervals when an interruption would be awkward (e.g. while a previous interrupt is being processed). When the system is disabled, interrupt signals are still collected, but not interrogated. The system is automatically disabled, when the interruption program is entered. It can be **enabled** again (or disabled) within the monitor using the privileged instructions **jump with interrupt enabled** (or **disabled**).

7.2. Interruption Conditions

The **interrupt signals** can be classified according to **priority** as follows:

First:	Instruction Exception	(bit 0)
Second:	Integer Overflow	(bit 1)
Third:	Floating-Point Overflow	(bit 2)
Fourth:	External Interruption	(bits 3 - 23)

Instruction exceptions are internal interrupts generated by instructions that attempt to violate the protection system. This interrupt has the highest priority and is the only one that can neither be masked off nor disabled. The interrupt number is set to zero before branching to the interruption program. Instruction exceptions are recognized in the following situations:

(1) Execution of an unassigned operation code.

(2) The effective address of an instruction refers to a non-existent storage location.

These two kinds of instruction exceptions will cause interruptions even when detected within the monitor program.

In the **task mode,** instruction exceptions are also caused by:

(3) Execution of a privileged instruction, and

(4) Attempts to alter or jump to a protected location.

An **integer interruption** with interrupt number 2 is caused by overflow occurring in integer arithmetic. This interruption can be masked off.

A **floating-point interruption** with interrupt number 4 is caused by exponent overflow or underflow in floating-point arithmetic. The interruption is maskable.

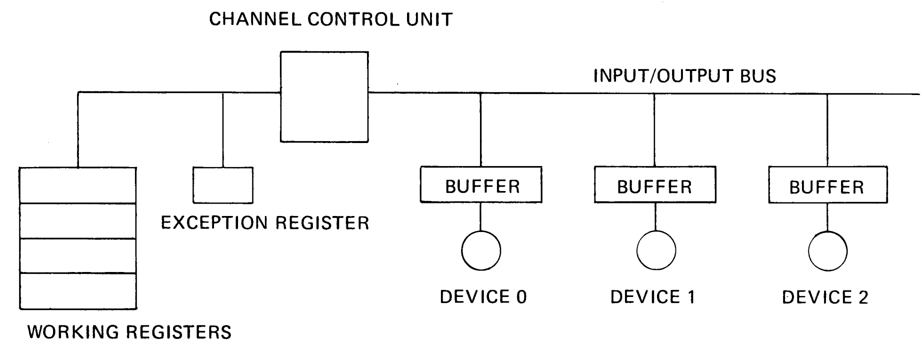
The remaining 21 bits in the interrupt register can be assigned to **external signals.** They have interrupt numbers from 6 to 46 and are all maskable. Through these interruptions the computer can respond to attention signals from real-time clocks, alarm contacts, operator interruption keys, and input/output devices. The system permits the connection of several interrupt signals to a single interrupt bit. This is done by connecting the interrupt bit to an external register of 24 bits. Each device connected to the same interrupt level has a bit position in this register. The external register is treated as a peripheral device by the computer. It can be input and cleared when the individual signals must be identified and served.

Chapter 8

LOW-SPEED DATA CHANNEL

8.1. Main Characteristics

Slow character-oriented devices such as typewriters, paper tape readers, and paper tape punches are connected to a single **low-speed data channel** that communicates directly with the internal **working registers.** Each device has a separate **buffer register** of up to 24 bits, which transmits or receives one character at a time to or from the external data medium.

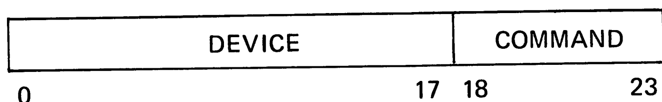


The data channel consists of a channel control unit and an input/output bus, with 24 bits for transfer of data to or from the device buffers and 6 bits for channel control signals. The transfers of data between the working registers and the device buffers take place one at a time under program control. Transfers between buffer registers and the external data media are, however, controlled independently by the devices; thus several of these transfers can take place at one time. The operation of this low-speed data channel will now be described in detail.

8.2. Input/Output Instruction

Initial program loading is controlled by the input instruction, **autoload word** (see Section 11.5.). All other input/output operations are handled by a single instruction, **input/output,** which has the standard instruction format (defined in Section 3.4.). Here, the **W field** selects the internal working register that will be connected to the data channel.

The **effective address** of the instruction is interpreted as follows:



The device is identified by a **device number** of 18 bits. Installations in which the channel does not have the capacity to address the maximum number of devices will interpret the device address modulo 2**N, where $N < 18$.

The input/output operation desired is defined by a **command code** of 6 bits.

8.3. Channel Operation

An input/output operation is initiated by the computer in two stages:

Selection Phase. The channel control unit attempts to establish a connection with the device by sending the device number on the bus simultaneously through all device control units. The success of this selection depends on whether the device in question is:

(1) available,
(2) busy, or
(3) disconnected.

The device is considered **disconnected,** if no device responds to the selection. This is either because the device number designates a non-existent device or because the power to the device is switched off, or because the device has been removed from its control unit.

The device responds to the selection with a **busy** signal, if it is in the process of executing a previous input/output operation. In the busy and disconnected states, the input/output operation is rejected and the computer proceeds immediately to the next instruction. The cause of rejection is made available to the program in the exception register.

Data Transfer Phase. If the device is available, it will accept the command code from the busline. Some of the commands will now cause the channel to perform an immediate data transfer from a working register to the external buffer register or vice versa. Finally, when the device operation is initiated, the computer proceeds to the next instruction.

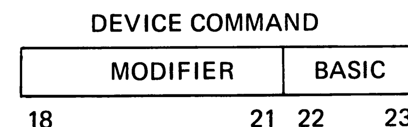
8.4. Disconnected and Busy Indication

The **exception register** has the following meaning after an input/output operation:

exception bit:
21 (unchanged)
22 disconnected
23 busy

8.5. Device Commands

The command code of 6 bits is divided into a **basic command** field of 2 bits and a **modifier** field of 4 bits:



The 16 possible modifications are specific for each type of device. The channel control unit recognizes only the 4 basic commands which are:

00 sense
01 control
10 read
11 write

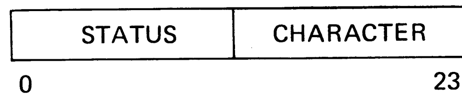
8.6. Read

The read command directs the device to start a transfer of the next character from the external data medium into its buffer register. The computer is released as soon as the operation is initiated (or rejected). The device is busy until the read operation has been completed.

8.7. Sense

The sense command is a request to the device to transfer the contents of its buffer register to a working register. This is done immediately under program control. The device is available after a sense command.

The right-most bits of the data-word transferred contain the last character received in the buffer register, whereas the left-most bits are **status bits** set during the input/output operation. The number of bits reserved for status information and for the character depends on the type of device. Unused bits in the data-word are permanently equal to zero.



Non-zero status bits indicate an exceptional result of the last input/output operation. The following are examples of status bits:

intervention
transmission error
timer
end of medium

Intervention is indicated when the operator has interfered manually with the device, e.g. by opening the door of a tape station.

Transmission error signifies a parity error during input from a paper tape reader; an analog input signal that is out of range, and so forth.

Timer indication is set when the device does not complete an operation within a certain time interval, for example, if the operator fails to respond to a typewriter input request within a certain time interval.

End of medium indicates that the operator should insert a roll of paper tape, a printer form, etc.

The status word can only be sensed if the device is **available.** If the device is **busy** or **disconnected,** the command is rejected as described above. The sense command can therefore be used in connection with the exception register to test whether a device is busy.

8.8. Write

The write command causes an immediate transfer of the contents of a working register to a device buffer, followed by an output operation to the external data medium. The computer is released as soon as the output operation has been initiated (or rejected). The device is busy until the write operation has been completed.

8.9. Control

The control command causes an immediate transfer of the contents of a working register to a device control unit, followed by a control operation of the device. The data-word transferred is interpreted either as a **selection address** (of an analog input terminal, or a track on a drum) or as a **control code** (specifying rewinding of a magnetic tape, etc.). The computer proceeds to the next instruction as soon as the control operation has been initiated (or rejected). The device is busy until the control operation has been completed.

8.10. Summary of Low-speed Channel

The operation of the low-speed data channel can be summarized as follows. The execution of an input/output instruction will always result in the exception register being set to indicate whether the operation specified by the command was initiated or rejected by the device. The computer will in any case immediately proceed to the next instruction.

A successful input operation from a device is performed by two input/output commands, as shown by the following simplified example:

```
A0:  IO       5<6 + 2    ; read (device 5);
     ----
A1:  IO W1    5<6 + 0    ; wait:
     SX       2.01       ; w1:= sense (device 5);
     JL.      A1.        ; if busy then goto wait;
```

The first command (read) directs the device to start reading the next character into its buffer register. As soon as this operation has been initiated, program execution continues. The second command (sense) is used in connection with an SX instruction to wait for the completion of the input. When the data value is available in the buffer register, the sense command will immediately transfer it to the working register specified.

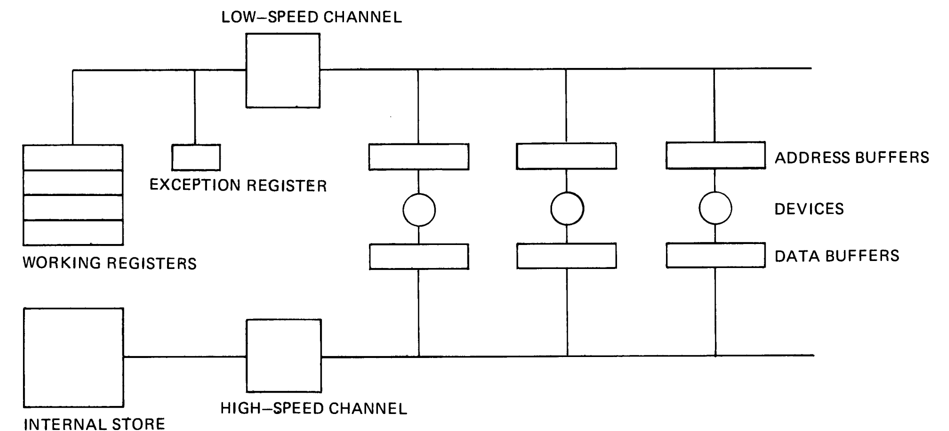
An output operation is initiated by a write command, which transfers the contents of a working register to an external buffer register and directs the device to start writing it out. The computer is released immediately, and is not concerned with the device in question until the operation has been completed. The outcome of the output operation is tested by a sense command, which transfers a status word to a working register.

Finally, it should be mentioned that it is possible to connect a device to the interrupt register so that the end of an input/output operation can be signalled directly by the device as a **program interruption.**

Chapter 9

HIGH–SPEED DATA CHANNEL

Input/output devices such as disk files, drums, and magnetic tape stations, which transmit large volumes of data at high speeds are connected to a single high-speed data channel. This channel performs buffered input/output directly to or from the internal store on a cycle-stealing basis. Program execution proceeds simultaneously with input/output operations.



Block transfers can take place on several devices at one time. A multiplexer switches rapidly among the devices connecting them to the high-speed channel, whenever they are ready to transfer a complete data-word to or from the store.

The capacity of the high-speed channel is only limited by the cycle time of the internal store. With a cycle time of 1.5 usec the maximum channel speed is:

input: 2.0 usec/word or 500 000 words/sec
output: 1.5 usec/word or 667 000 words/sec

Storage protection keys are not changed by high-speed input/output.

The devices use the low-speed channel to transfer commands and addresses of buffer areas in the store. This is done by means of the **control** command. The **sense** command is used to transfer a status word at the end of an operation. All

input/output operations are thus handled by the standard **input/output** instruction, and the high-speed devices are simply numbered in continuation of the low-speed devices.

If one or more data words are lost during input/output due to overloading of the data channel, the device in question delivers a status bit, **data overrun,** at the end of the operation. The program can then repeat the operation.

For further details, the reader should consult the reference manuals for specific peripheral devices.

Chapter 10

STANDARD PERIPHERAL DEVICES

10.1. Console Devices

The standard input/output equipment mounted on the console consists of a **paper tape reader,** a **paper tape punch** and a **typewriter** operating on the low-speed data channel.

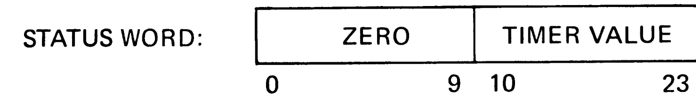
The specifications of these are defined in the reference manuals for peripheral devices.

10.2. Interval Timer

The interval timer is connected to the low-speed data channel by means of a buffer register of 24 bits.

The interval timer consists of a binary counter of 14 bits (denoted bits 10 to 23). The counter is increased by one in bit 23 every 100 microseconds. It counts modulo 2**14. The frequency stability is 1 part in 10**6. The value of the timer is random when the RC 4000 is started.

The value of the timer can be input to a working register by means of a sense command. This operation does not change the value of the timer.



The timer produces an interrupt with regular intervals. The value of the timer is not changed by an interrupt.

The interrupt interval can be set manually to one of the following values:

1.6 msec
3.2 -
6.4 -
12.8 -
25.6 -
51.2 -
102.4 -
204.8 -
409.6 -
819.2 -
1638.4 -

Chapter 11

OPERATOR CONTROL PANEL

11.1. Indicators and Control Keys

Operator communication with the RC 4000 is performed mainly via typewriters. The operator control panel has therefore been minimized to contain only 3 control keys: **reset, start** and **autoload.** Two indicator lamps show whether the machine is operating(**CPU running**) or stopped(**CPU error**).

A keylock switch prevents unauthorized or accidental operation of any of the console keys. When the control panel is locked, the keys have no effect.

11.2. Machine Errors

The RC 4000 is controlled by a microprogram in a read-only store. The control unit performs parity checking of each microinstruction fetched. Parity checking is also performed during each read operation from the internal store. Detection of a parity error will cause the microprogram to be stopped and the **CPU error** lamp to be lit.

11.3. Reset Control

The **reset** key is a special interruption key, which is interrogated once in every instruction cycle. Pressing the reset key has the following effect:

(1) The program in progress is interrupted with the return address stored in storage word 10.

(2) The control unit is set in the monitor mode with the interrupt system disabled

The computer remains in the reset state, until the operator presses either the **start** key or the **autoload** key.

A **power failure** detected at the end of an instruction will automatically cause a reset action. In this connection, it should be noted that while the internal store is non-volatile on power shut-downs, the contents of all registers are lost.

Turning the **power on** also sets the computer in the reset state (without the instruction counter being stored).

11.4. Start Control

The **start** key is used in the reset state to start a monitor program. When this key is pressed, the computer jumps to an address kept in storage word 14.
The start key has no effect, unless the machine is in the reset state.

11.5. Autoload Control

The **autoload** key is used in the reset state to initiate a bootstrapping routine that loads a program into the internal store from device number 0 (normally the paper tape reader). The **autoload** key has no effect, unless the machine is in the reset state.

When the autoload key is pressed, the computer reads four 6-bit characters into working register 0. Following this, it sets the protection key of the register to zero and proceeds to execute its contents as an instruction.

The bootstrapping program continues the loading by means of a privileged instruction, **autoload word.** This instruction reads four 6-bit characters from device number 0 into the storage word designated by the effective address. The loading is completed by setting the protection key of the word to zero.

The use of the autoload instruction is illustrated by the following example. Let us assume that a program represented on binary paper tape must be loaded into the store from location 16 onwards. The sequence of instructions on the paper tape should be as follows:

```
AW   2
AW   4
JL   0
AW  16
First Program Instruction
AW  18
Second Program Instruction
-----
JL  16
```

When the **reset** and **autoload** keys are pressed, the computer reads the first autoload instruction (AW 2) into working register 0 and executes it. This instruction in turn reads the second autoload instruction (AW 4) into working register 1, and proceeds to execute it after the instruction counter has been increased sequentially. This has the effect of placing a jump instruction (JL 0) into working register 2.

We have now established the following programming loop in the working registers:

address:	instruction:
0	AW 2
2	AW 4
4	JL 0

The return jump to working register 0 causes the third autoload instruction (AW 16) to be placed in working register 1. Its execution loads the first instruction of the actual program into storage word 16. By repetition of this loop, the following program instructions are loaded into words 18, 20, 22, etc. The loading is terminated by reading a jump instruction (JL 16) into working register 1. Its subsequent execution transfers control to the start of the program loaded.

The autoload instruction should be used only to load an initial program into an empty store. Using it within a real-time monitor is indeed hazardous: if the loading device is disconnected or if input exceptions, such as parity error are detected, the autoload instruction will immediately set the computer in the reset state, in which it is beyond program control. In the reset state, the autoloading can be repeated by pressing the **autoload** key.

11.6. Local/Remote Indication

Each peripheral device is supplied with a **local/remote** switch, which can be set manually. In the **remote** state, the device is program controlled. In the **local** state, the operator can insert paper tapes, adjust printing sheets, and so on. In this state, a device can accept an input/output command, but the actual initiation of the operation is delayed until the operator sets the device remote again.

The operator control panel shows the status of **local/remote** switches.

Chapter 12

TECHNICAL CONTROL PANEL

12.1. Operating Modes

The technical control panel is intended mainly for manual control of the RC 4000 during maintenance periods. It will therefore only be considered briefly in this manual.

A keylock switch turns the technical panel on (**technical mode**) and off (**normal mode**).

In the **normal mode**, the operator is prevented from using the maintenance panel, i.e. the control keys have no effect.

In the **technical mode**, the operator can perform the following control functions:

(1) Start and stop the computer.

(2) Execute instructions step by step.

(3) Alter and display registers.

(4) Select, display, and execute single microinstructions.

The operator control panel is inoperative in the **technical mode.**

12.2. Instruction Step Keys

The **single instruction** key stops the program at the end of the instruction cycle. If this key is pressed repeatedly, the computer steps through the program, stopping after each instruction.

The **single microinstruction** key permits the operator to stop the computer after each microinstruction.

The computer returns to continuous program execution, when the **continue** key is pressed (or when the maintenance panel is set in the **normal mode**).

12.3. Register Setting and Display

When the computer is stopped, the registers of the central processor can be displayed and set manually. The indicator lamps display a single register in binary form. Each indicator has two push-buttons for setting the corresponding register

bit to 0 and 1 respectively. Each register has a selection key that connects it to the display when pressed.

The registers that can be displayed and set are: W0-W3, FR, IC, PR, SB (including PK), SE, AR, AE, BR, BE, SC, and EX. IR and IM can only be displayed. (The meaning of these register names is defined in Section 14.2).

12.4. Microinstruction Selection and Display

A display is also provided for the microprogram store. This shows the address and the format of the current microinstruction on binary form.

The microaddress register (MAR) can be set manually by means of push-buttons. It is thus possible to start program execution from any address in the microprogram.

The control key **MAR manually controlled** causes the computer to repeat the present microinstruction continuously, when the **continue key** is pressed. Only in this state can the microaddress register be set and the correponding microinstruction be displayed.

A microinstruction consists of a **micro command** part and a **jump selector** part.

The control key **MAR computer controlled** returns the microprogram to its normal sequential execution.

12.5. Parity Control

In the **technical mode,** the parity control of the core store and the microprogram store can be turned on or off by push buttons called **core store control** and **MPS control.**

In the **normal mode**, the parity control is always on.

When a parity error is detected while the parity control is on the machine goes into the reset state and one of the indicators **core store error** or **MPS error** is lit.

Chapter 13

INSTRUCTION SET

Address Handling
AM Modify Next Address
AL Load Address
AC Load Address Complemented

Register Transfer
HL Load Half Register
HS Store Half Register
RL Load Register
RS Store Register
RX Exchange Register and Store
DL Load Double Register
DS Store Double Register

Integer Byte Arithmetic
BZ Load Byte with Zeroes
BL Load Integer Byte
BA Add Integer Byte
BS Subtract Integer Byte

Integer Word Arithmetic
WA Add integer Word
WS Subtract Integer Word
WM Multiply Integer Word
WD Divide Integer Word

Integer Double Word Arithmetic
AA Add Integer Double Word
SS Subtract Integer Double Word

Arithmetic Conversion
CI Convert Integer to Floating
CF Convert Floating to Integer

Floating-Point Arithmetic
FA Add Floating
FS Subtract Floating
FM Multiply Floating
FD Divide Floating

Logical Operations
LA Logical And
LO Logical Or
LX Logical Exclusive Or

Shift Operations
AS Shift Single Arithmetically
AD Shift Double Arithmetically
LS Shift Single Logically
LD Shift Double Logically
NS Normalize Single
ND Normalize Double

Sequencing
JL Jump with Register Link
SH Skip if Register High
SL Skip if Register Low
SE Skip if Register Equal
SN Skip if Register Not Equal
SO Skip if Register Bits One
SZ Skip if Register Bits Zero
SX Skip if No Exceptions
SP Skip if No Protection

Monitor Control

JE Jump with Interrupt Enabled
JD Jump with Interrupt Disabled
IC Clear Interrupt Bits
IS Store Interrupts Register
ML Load Mask Register
MS Store Mask Register
XL Load Exception Register
XS Store Exception Register
PL Load Protection Register
PS Store Protection Register
KL Load Protection Key
KS Store Protection Key
IO Input/Output
AW Autoload Word

Chapter 14

DEFINITION OF INSTRUCTIONS

14.1. Algol Notation

This chapter gives a formal definition of the instruction logic. The basic instruction cycle and all operations are described in the Algol 60 language, extended with the following concepts:

Declarations. A register declaration consists of an identifier, followed by a specification of the bit size in parentheses. As an example:

register SB(0:23)

is a declaration of a storage buffer register SB of 24 bits, numbered 0 to 23 from the left. Similar declarations are introduced for register arrays and the internal store:

register array W(0:3)(0:23)
storage array word(0:word limit)(0:23)

Algorithms. Reference to a sub-field within a register is defined in the following way: Bit number i in the register SB is denoted SB(i). The register field from bit i to bit j is described as SB(i:j). Storage references to bytes, words, and protection keys are denoted byte(SB), word(SB), and protection key(SB), respectively.

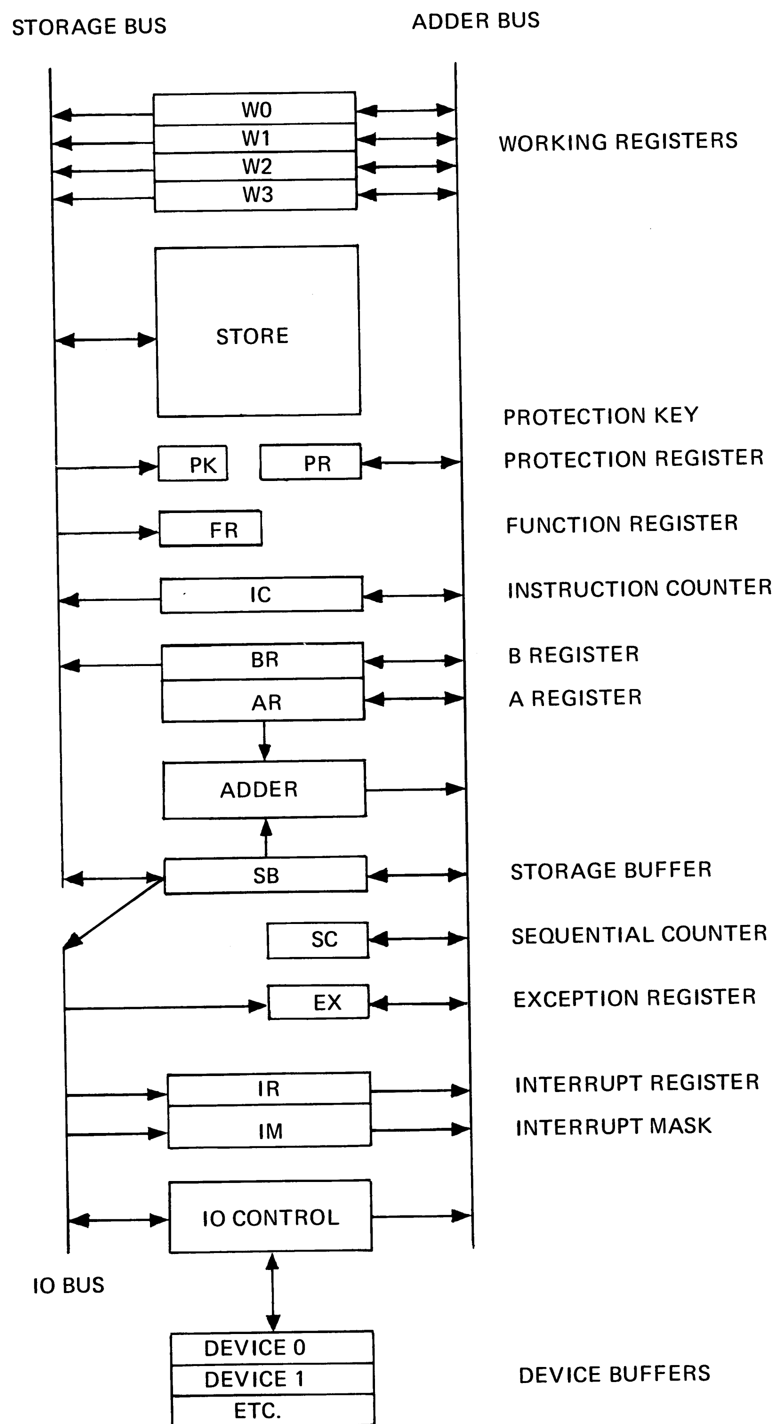
14.2. Register Structure

The instruction logic is defined within the frame of the register structure shown in the following figure. It corresponds very closely to the actual structure of the central processor.

A data-word transferred to or from the store is held in the combined register SB and PK.

When an instruction is fetched, the operation byte is assigned to the FR register, while the displacement byte is placed in SB and extended to 24 bits.

SB and AR act as input registers to the adder. AR performs single-length shift operations and, combined with BR, double-length shift operations.



SE, AE, and BE denote extensions of the registers SB, AR, and BR used in floating-point operations.

The sequential counter SC is used to determine the number of iterations in arithmetic operations.

The functional units are **declared** as follows:

```
register PK(0:2), PR(0:7), FR(0:11), IC(0:23),
         BR(0:23), AR(-1:23), SB(0:23),
         BE(24:35), AE(24:37), SE(24:37),
         SC(11:23), EX(21:23), IR(0:23), IM(0:23);

register array W(0:3)(0:23),
         device buffer(0:device limit)(0:23);

storage array word(0:word limit) (0:23),
         protection key(0:word limit) (0:2);

boolean monitor mode, interrupt disabled,
         reset key, start key, autoload key;
```

In references to registers and the store, the following **abbreviations** are used:

Abbreviation:	Used instead of:
W	W(FR(6:7))
Wpre	W(if FR(6:7) = 0 then 3 else FR(6:7)–1)
Wfrac	Wpre concat W(0:11)
Wexp	W(12:23)
ARBR	AR concat BR
BF	BR concat BE
AF	AR concat AE
SF	SB concat SE
SBexp	Various 12-bit registers holding the exponent of a storage operand.
byte(SB)	if SB(23) = 0 then word(SB)(0:11) else word(SB)(12:23)
fraction(SB)	word(if SB = 0 then 6 else SB–2) concat word(SB)(0:11)
exponent(SB)	word(SB)(12:23)

14.3. Elementary Operations

Instructions operating on registers and storage operands will be defined in terms of the following elementary operations:

positive signed not + – shiftleft shiftright

The monadic operator **positive** extends an operand to the left with zeroes until it has the same number of bits as the operand to which the result is assigned.

The monadic operator **signed** extends the signbit of an operand to the left until it has the same number of bits as the operand to which the result is assigned.

The monadic operators **shiftleft** and **shiftright** shift an operand one position to the left and right with zero extension.

The monadic operator **not** negates all bits of an operand, i.e. ones become zeroes and zeroes become ones.

The dyadic operators + and – perform addition and subtraction of two operands in the binary two's complement representation.

Consider four registers:

register R(0:j), A(0:j), B(0:j), C(i:j)

where 0 < i < j. The operators can now be defined by the following statements:

Statement:	Definition:
R: = positive C;	R(i:j):= C; R(0:i–1):= 0;
R:= signed C;	R(i:j):= C; for bit:= 0 step 1 until i–1 do R(bit):= R(i);
R:= shiftleft R;	R(0:j–1):= R(1:j); R(j):= 0;
R:= shiftright R;	R(1:j):= R(0:j–1); R(0):= 0;
R:= not R;	for bit:= 0 step 1 until j do R(bit):= if R(bit) = 0 then 1 else 0;

R:= A +B;	next carry:= 0;
	for bit:=j step – 1 until 0 do
	begin carry:= next carry;
	if A(bit)<>B(bit) then R(bit):= not carry
	else begin R(bit):= carry; next carry:= A(bit) end;
	end;

R:= A – B;	R:= A + ((not B) + 1);

14.4. Control Panel Functions

```
Reset System:
   word(10):= IC;
   comment: the system is reset when the operator unlocks the control panel and
   depresses the reset key, when the power is switched off, or when input errors
   are detected during autoloading;
Power On:
   monitor mode:= interrupt disabled:= true;
   reset key:= start key:= autoload key:= false;
   comment: these booleans are set to true, when the operator depresses the corre-
   sponding keys on the control panel;
After Reset:
   if autoload key then
   begin IC:= SB:= 0;
      goto Autoload Word;
   end;
   if start key then
   begin IC:= word(14); IC(23):= 0;
      goto Fetch Instruction;
   end;
   goto After Reset;
```

14.5. Instruction Fetch Cycle

```
Next Instruction:
   if interrupt disabled then goto Fetch Instruction;
Interruption Service:
   for bit:= 0 step 1 until 23 do
   begin if IR(bit) = 1 and IM(bit) = 1 then
      begin IR(bit):= 0;
         word(8):= shiftleft bit;
         word(10):= IC;
         IC:= word(12); IC(23):= 0;
         monitor mode:= interrupt disabled:= true;
         goto Fetch Instruction;
      end;
   end;
Fetch Instruction:
   AR:= positive IC; comment: save relative address;
   if IC > word limit then goto Instruction Exception;
```

```
   if reset key then goto Reset System;
   FR:= byte(IC); SB:= signed byte(IC +1);
   PK:= protection key(IC);
Decode Instruction:
   IC:= IC + 2;
   if not monitor mode and PR(PK) = 1
   then goto Instruction Exception;
   monitor mode:= PR(PK) = 1;
   if FR(8) <>0 then SB:= SB + AR(0:23); comment: relative address;
   if FR(10:11) <> 0 then SB:= SB + W(FR(10:11)); comment: indexing;
   if FR(9) <> 0 then
   begin Test Address; SB:= word(SB); comment: indirect address;
   end;
   comment: SB contains the effective address and IC points to
   the next instruction;
   goto operation (FR(0:5));
```

14.6. Protection Procedures

```
Instruction Exception:
   IR(0):= 1; comment: IM(0) is always 1;
   goto Interruption Service;

procedure Test Address;
begin if SB(0:22) > word limit then goto Instruction Exception,
end;

procedure Test Mode;
begin if not monitor mode then goto Instruction Exception;
end;

Procedure Test Protection;
begin PK:= protection key(SB);
   if not monitor mode and PR(PK) = 1
   then goto Instruction Exception;
end;
```


14.7. Arithmetic Procedures

```
procedure Test Integer,
begin if AR(-1) <> AR(0) then IR(1):= EX(22):= 1;
   EX(23):= carry;
end;

procedure Fetch Floating Operands;
comment: The procedure moves the register and storage operand to the
following registers:
   AF      register fraction
   SF      storage fraction
   Wexp    register exponent
   SBexp   storage exponent;
begin
   AF(-1:35):= signed Wfrac; AF(36:37):= 0;
   SF(0:35):= fraction(SB); SF(36:37):= 0;
   SBexp:= exponent(SB);
end;

procedure Test Precision Mode and Store;
comment: Depending on exception bit 21 the procedure leaves the fraction in
AF unchanged, or sets the last two bits of it equal to the last but two.
Finally, the fraction in AF and the exponent in SC are stored in double
working registers;
begin
   if EX(21) = 1 then AF(35):= AF(34):= AF(33);
   Wfrac:= AF(0:35); Wexp:= SC(12:23);
end;

procedure Normalize and Round Floating;
comment: The procedure normalizes the fraction in AF, rounds it, and re-normal-
izes it if necessary. The exponent in SC is adjusted correspondingly. Finally, the
exponent is tested for overflow and underflow, i.e. the interrupt and exception
bits are set if the exponent exceeds 2047 or -2048;
begin
```

```
Again: if AF(-1)  <> AF(0) then
       begin comment: Right shift;
             AF(0:37):= AF(-1:36); SC:= SC + 1;
       end else
       if AF = 0 then SC:= -2048 else
       begin
             for SC:= SC, SC - 1 while AF(0) = AF(1) do
             AF:= shiftleft AF;
             comment: Left shifts until normalized;
       end;
       if AF(36) = 1 then
       begin comment: Rounding;
             AF:= AF + 4; AF(36:37):= 0; goto Again;
       end;
       if SC(11) <> SC(12) then EX(22):= IR(2):= 1;
   end;
```

14.8. Instruction Execution

For each instruction the normal execution is defined. Also specified are the setting of the exception register and the conditions that will cause a program interruption. The algorithms follow the actual micro-program closely with the omission of irrelevant intermediate steps.

Modify Next Address

Use the effective address as an increment to the displacement in the next instruction. The operation changes only the effective address of the next instruction whose D field remains unchanged.

```
comment: the modifier address is saved in AR, and the next
instruction is fetched and modified;
AR:= signed SB;
if IC > word limit then goto Instruction Exception;
FR:= byte(IC); SB:= signed byte(IC + 1);
PK:= protection key(IC);
SB:= SB + AR(0:23);
AR:= positive IC; comment: save relative address;
goto Decode Instruction;
```

Exception: unchanged.

Interruption: disabled until the next instruction has been executed.

Load Address

Load the W register with the effective address.

```
W:= SB; goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

Note: When the same register is specified by the W and X fields, the operation increments the register by the value of the D field.

Load Address Complemented

Load the W register with the two's complement of the effective address. Complementation of the maximum negative number -2**23 gives the result -2**23 and produces an overflow.

```
EX(22:23):= 0;
AR:= 0 - signed SB;
W:= AR(0:23); Test Integer;
goto Next Instruction;
```

Exception: (22) overflow, (23) carry.

Interruption: (1) integer overflow.

Note: When the same register is specified by the W and X fields and the D field is zero, the operation is a sign reversal of the register.

Load Half Register

Insert the storage byte addressed in the right-most 12 bits of the W register without changing the left-most 12 bits. The storage byte remains unchanged.

```
Test Address; W(12:23):= byte(SB);
goto Next Instruction;
```

Exception: unchanged.

Interruption: (0) undefined address.

Note: When SB < 8, the operation moves 12 bits from the left or right side of one register to the right side of another register.

Store Half Register

Store the right-most 12 bits of the W register in the storage byte addressed. The register remains unchanged.

```
Test Address; Test Protection;
byte(SB) := W(12:23);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.
Note: When SB < 8, the operation moves 12 bits from the right side of one register to the left or right side of another register.

Load Register
Load the W register with the storage word addressed. The storage word remains unchanged.

```
Test Address; W:= word(SB);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation is a register to register transfer.

Store Register
Store the W register in the storage word addressed. The register remains unchanged.

```
Test Address; Test Protection;
word(SB):= W;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.
Note: When SB < 8, the operation is a register to register transfer.

Exchange Register and Store
The W register is stored in the storage word addressed and the previous contents of the storage word is loaded into the register.

```
AR:= signed W;
Test Address; W:= word(SB);
Test Protection; word(SB):= AR(0:23);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.
Note: When SB < 8, the operation exchanges the contents of two registers.

Interruption: (0) undefined address or protection violation.
Note: When SB < 8, the operation exchanges the contents of two registers.

Load Double Register
Load the register pair Wpre and W with the storage double word addressed. The storage word remains unchanged.

```
BR:= if SB = 0 then 6 else SB – 2;
Test Address;
W:= word(SB); Wpre:= word(BR);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation is a double register to double register transfer except when the W register is also the first word of the storage operand.

Store Double Register
Store the register pair Wpre and W in the storage double word addressed. The register pair remains unchanged.

```
Test Address; Test Protection; word(SB):= W;
SB:= if SB = 0 then 6 else SB – 2;
Test Protection; word(SB):= Wpre;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.
Note: When SB < 8, the operation is a double register to double register transfer except when the Wpre register is also the last word of the storage operand.

Load Byte with Zeroes
Insert the storage byte addressed in the right-most 12 bits of the W register and extend it towards the extreme left with zeroes. The storage byte remains unchanged.

```
Test Address; W:= positive byte(SB);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation moves 12 bits from the left or right side of one register to the right side of another register, followed by a zero extension to 24 bits.

Load Integer Byte

Insert the storage byte addressed in the right-most 12 bits of the W register and extend the sign bit towards the extreme left. The storage byte remains unchanged.

```
Test Address; W:= signed byte(SB);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation moves 12 bits from the left or right side of one register to the right side of another register, followed by a sign extension to 24 bits.

Add Integer Byte, Subtract Integer Byte

The storage byte addressed is extended towards the left to 24 bits and added to (or subtracted from) the W register, and the result is placed in the register. The storage byte remains unchanged.

```
EX(22:23):= 0; Test Address;
if add then AR:= signed W + signed byte(SB)
       else AR:= signed W – signed byte(SB);
W:= AR(0:23); Test Integer;
goto Next Instruction;
```

Exception: (22) overflow, (23) carry.
Interruption: (0) undefined address, (1) integer overflow.
Note: When SB < 8, the operations adds (or subtracts) 12 bits from the left or right side of one register to (or from) 24 bits in another register.

Add Integer Word, Subtract Integer Word

The storage word addressed is added to (or subtracted from) the W register, and the result is placed in the register. The storage word remains unchanged.

```
EX(22:23):= 0; Test Address;
if add then AR:= signed W + signed word(SB)
       else AR:= signed W – signed word(SB);
W:= AR(0:23); Test Integer;
goto Next Instruction;
```

Exception: (22) overflow, (23) carry.
Interruption: (0) undefined address, (1) integer overflow.
Note: When SB < 8, the operation adds (or subtracts) one register to (or from) another register.

Multiply Integer Word

The W register is multiplied by the storage word addressed. The 48-bit signed product is placed in the register pair Wpre and W. Overflow cannot occur.

```
comment: The multiplicand and the multiplier are placed in SB and
BR. SC determines the number of iterations. After multiplication,
ARBR contains the product;
Test Address; AR:= 0; BR:= W; SB:= word(SB);
for SC:= 22 step –1 until 0 do
begin if BR(23) = 1 then AR:= AR + signed SB;
   ARBR(0:47):= ARBR(–1:46);
end;
if BR(23) = 1 then AR:= AR – signed SB;
ARBR(0:47):= ARBR(–1:46);
Wpre:= AR(0:23); W:= BR;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation is a register by register multiplication.

Divide Integer Word

The register pair Wpre and W is divided by the storage word addressed. The 24-bit signed quotient is placed in the W register, while the 24-bit signed remainder is placed in the preceeding register Wpre. The absolute value of the remainder is less than the absolute value of the divisor, and a non-zero remainder has the same sign as the dividend. An overflow is registered, if the divisor is zero or if the quotient exceeds 24 bits. In this case the dividend remains unchanged in the working registers.

```
  EX(22:23):= 0; Test Address;
  AR:= Wpre; BR:= W; SB:= word(SB);
comment: The dividend and the divisor are placed in ARBR and SB. After
division by the non-restoring method, AR(–1:22) contains a remainder and
ARBR(23:47) contains a 25 bit quotient in which the left-most bit is
represented by its complemented value and the right-most bit by a one;
  next digit:= if AR(–1) = SB(0) then 1 else 0;
  for SC:= 23 step –1 until 0 do
  begin ARBR:= shiftleft ARBR; ARBR(47):= next digit;
        AR:= if next digit = 1 then AR – signed SB
                                 else AR + signed SB;
        next digit:= if AR(–1) = SB(0) then 1 else 0;
```

```
end;
ARBR:= shiftleft ARBR; ARBR(47):= 1;
if AR(23) = BR(0) then
Quotient Overflow:
begin IR(1):= EX(22):= 1; goto Next Instruction end;
comment: the following ensures that abs(remainder) <abs(divisor);
if abs(AR(-1:22)) = abs(SB) then
begin if SB <0 then BR:= BR - 1 else
      begin BR:= BR + 1;
      if AR(23) = BR(0) then goto Quotient Overflow;
      end;
      AR:=0;
end;
AR(0:23):= AR(-1:22);

comment: the following ensures that sign(remainder) = sign(dividend)
for a non-zero remainder;
if AR <> 0 and AR(-1) <> Wpre(0) then
begin if AR(-1) = SB(0) then
      begin AR:= AR - signed SB; BR:= BR + 1 end
      else
      begin AR:= AR + signed SB; BR:= BR - 1 end;,
end;
Wpre:= AR(0:23); W:= BR;
goto Next Instruction;
```

Exception: (22) overflow, (23) zero.
Interruption: (0) undefined address, (1) integer overflow.
Note: When SB < 8, the operation is a double register by register division.

Add Integer Double Word, Subtract Integer Double Word
The storage double word addressed is added to (or subtracted from) the register pair Wpre and W, and the result is placed in the register pair. The storage double word remains unchanged.

```
EX(22:23):= 0; Test Address;
if add then AR:= signed W + signed word(SB)
       else AR:= signed W - signed word(SB);
W:= AR(0:23);
```



```
SB:= if SB = 0 then 6 else SB - 2;
if add then
begin if carry then AR:= signed Wpre + signed word(SB) + 1
               else AR:= signed Wpre + signed word(SB);
end else
begin if carry then AR:= signed Wpre - signed word(SB)
               else AR:= signed Wpre - signed word(SB) - 1;
end;
Wpre:= AR(0:23); Test Integer;
```

Exception: (22) overflow, (23) carry.
Interruption: (0) undefined address, (1) integer overflow.
Note: When SB < 8, the operation adds or subtracts two register pairs except when the W register is also the first word of the storage operand.

Convert Integer to Floating
Convert the W register, interpreted as an integer multiplied by 2**effective address, to a floating-point number and place it in the register pair Wpre and W. An overflow is registered if the exponent exceeds the 12 bit range.

```
EX(22:23):= 0;
AF(-1:23):= signed W; AF(24:47):= 0;
if AF = 0 then
begin Wfrac:= 0; Wexp:= -2048;
      goto Next Instruction;
end;
SC:= 23;
Normalize and Round Floating;
SC:= SC + SB(11:23);
if SC(11) <> SC(12) then EX(22):= IR(2):= 1;
Wfrac:= AF(0:35); Wexp:= SC(12:23);
goto Next Instruction;
```

Exception: (22) overflow, (23) zero.
Interruption: (2) floating-point overflow.

Convert Floating to Integer
Convert the register pair Wpre and W, interpreted as a floating-point number multiplied by 2**effective address, to an integer and place it in the W register. Wpre remains unchanged. An overflow is registered if the integer exceeds the 24 bit range.

```
EX(22:23):= 0;
AF(–1:35):= signed Wfrac; AF(36:37):= 0;
SB:= 23 – SB – signed Wexp; SC:= SB(11:23);
if SB < 0 and AF <> 0 then goto Integer Overflow;
if SB >= 64 then AF:= 0;
if 0 < SB and SB< 64 then
for SC:= SC – 1 step –1 until 0 do AF(0:37):= AF(–1:36);
if AF(24) = 1 then AR:= AR + 1; comment: Rounding;
if AF(–1) <> AF(0) then
Integer Overflow: EX(22):= IR(1):= 1
else W:= AR(0:23);
goto Next Instruction;
```

Exception: (22) overflow, (23) zero.
Interruption: (1) integer overflow.

Add Floating, Subtract Floating
The storage double word addressed is added to (or subtracted from) the register pair Wpre and W as a floating-point number, and the result is placed in the register pair. The storage double word remains unchanged.

```
EX(22:23):= 0; Test Address; Fetch Floating Operands;
SC:= signed Wexp – signed SBexp;
if SC >= 38 then
begin SC:= signed Wexp; Test Precision Mode and Store;
      goto Next Instruction;
end else
if SC <= –38 then
begin if add then
      begin AF:= signed SF; SC:= signed SBexp;
            Test Precision and Store;
            goto Next Instruction;
      end;
      Wexp:= SBexp; AF:= 0;
end else
if SC > 0 then
for SC:= SC –1 step –1 until 0 do SF(1:37):= SF(0:36) else
if SC < 0 then
begin Wexp:= SBexp;
      for SC:= SC + 1 step 1 until 0 do AF(0:37):= AF(–1:36);
end;
if add then AF:= AF + signed SF
      else AF:= AF – signed SF;
SC:= signed Wexp;
Normalize and Round Floating;
Test Precision Mode and Store;
goto Next Instruction;
```


Exception: (22) overflow, (23) zero.
Interruption: (0) undefined address, (2) floating-point overflow.
Note: When SB < 8, the operation is a floating-point addition or subtraction of two register pairs.

Multiply Floating
The register pair Wpre and W is multiplied by the storage double word addressed as a floating-point number, and the product is placed in the register pair. The storage double word remains unchanged.

```
EX(22:23):= 0; Test Address; Fetch Floating Operands;
BF:= AF(0:35); AF:= 0;
for SC:= 34 step –1 until 0 do
begin if BF(35) = 1 then AF:= AF + signed SF;
      AF(0:37):= AF(–1:36); BF:= shiftright BF;
end,
if BF(35) = 1 then AF:= AF – signed SF;
SC:= signed Wexp + signed SBexp;
Normalize and Round Floating;
Test Precision Mode and Store;
goto Next Instruction;
```

Exception: (22) overflow, (23) zero.
Interruption: (0) undefined address, (2) floating-point overflow.
Note: When SB < 8, the operation is a floating-point multiplication of two register pairs.

Divide Floating
The register pair Wpre and W is divided by the storage double word addressed as a floating-point number, and the quotient is placed in the register pair. The storage double word remains unchanged.

```
EX(22:23):= 0; Test Address; Fetch Floating Operands;
SC:= signed Wexp – signed SBexp + 35;
if AR = 0 then
```

```
begin comment: Zero result or overflow for 0/0;
        if SB = 0 then EX(22):= IR(2):= 1
                  else Wexp:= -2048;
        goto Next Instruction;
end else
if SB = 0 then
begin comment: Overflow for X/0;
        EX(22):= IR(2):= 1;
        goto Next Instruction;
end else
if AF(-1) = SF(0) then
begin comment: First quotient digit = 0;
   BF:= 0; AF:= shiftleft (AF - signed SF);
end else
begin comment: First quotient digit = 1;
   BF:= -1; AF:= shiftleft (AF + signed SF);
end;
next digit:= if AF(-1) =SF(0) then 1 else 0;
for SC:= SC, SC - 1 while BF(0) = BF(1) do
begin comment: The iteration proceeds until the quotient is
   normalized. SC then contains the exponent;
   BF:= shiftleft BF; BF(35):= next digit;
   AF:= if next digit = 1 then AF - signed SF
                             else AF + signed SF;
   next digit:= if AF(-1) =SF(0) then 1 else 0;
   AF:= shiftleft AF;
end;
if AF(-1) =SF(0) then AF(36:37):= 2 else AF(36:37):= 0;
AF(-1:35):= signed BF;
Normalize and Round Floating;
Test Precision Mode and Store;
goto Next Instruction;
```

Exception: (22) overflow, (23) zero.
Interruption: (0) undefined address, (2) floating-point overflow.
Note: When SB < 8, the operation is a floating-point division of two register pairs.

Logical And
The W register is combined with the storage word addressed by a logical And operation. The result is placed in the register. The storage word remains unchanged.

```
Test Address; SB:= word(SB);
for bit:= 0 step 1 until 23 do
W(bit):= if W(bit) = 1 and SB(bit) = 1 then 1 else 0;
goto Next Instruction;
```

Exception: unchanged,
Interruption: (0) undefined address.
Note: When SB < 8, the operation is an And combination of two registers bit by bit.

Logical Or
The W register is combined with the storage word addressed by a logical Or operation. The result is placed in the register. The storage word remains unchanged.

```
Test Address; SB:= word(SB);
for bit:= 0 step 1 until 23 do
W(bit):= if W(bit) = 1 or SB(bit) = 1 then 1 else 0;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation is an Or combination of two registers bit by bit.

Logical Exclusive Or
The W register is combined with the storage word addressed by a logical Exclusive Or operation. The result is placed in the register. The storage word remains unchanged.

```
Test Address; SB:= word(SB);
for bit:= 0 step 1 until 23 do
W(bit):= if W(bit) <> SB(bit) then 1 else 0;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.
Note: When SB < 8, the operation is an Exclusive Or combination of two registers bit by bit. When all bits in the word addressed are ones, the operation is a logical Negation of the register, bit by bit.

Shift Single Arithmetically

Shift the contents of the W register the number of places specified by the effective address in SB. If SB is negative, then shift right with sign extension in the upper bits, otherwise shift left with zero extension in the lower bits. Overflow is tested for each single shift.

```
EX(22:23):= 0;
if SB = 0 then goto Next Instruction;
if abs(SB) >=64 then SB:= sign(SB)*48;
if SB < 0 then
begin for SC:= 1 step 1 until –SB do W(1:23):= W(0:22);
end else
begin for SC:= 1 step 1 until SB do
        begin if W(0) < > W(1) then EX(22):= IR(1):= 1;
              W:= shiftleft W;
        end;
end;
goto Next Instruction;
```

Exception: (22) overflow, (23) zero.
Interruption: (1) integer overflow.

Shift Double Arithmetically

Same as Shift Single Arithmetically performed with the register pair Wpre and W.

Shift Single Logically

Shift the contents of the W register the number of places specified by the effective address in SB. If SB is negative, then shift right with zero extension in the upper bits, otherwise shift left with zero extension in the lower bits. Overflow is not indicated.

```
if SB = 0 then goto Next Instruction;
if SB <= 64 then SB:= sign(SB)*48;
if SB < 0 then
begin for SC:= 1 step 1 until –SB do W:= rightshift W;
end else
begin for SC:= 1 step 1 until SB do W:= leftshift W;
end;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.

Shift Double Logically

Same as Shift Single Logically performed with the register pair Wpre and W.

Normalize Single

Shift the contents of the W register left with zero extension until bit 0 is different from bit 1. The number of shifts performed is stored as a negative integer in the storage byte addressed. If W = 0 the number of shifts is set to –2048.

```
if W = 0 then SC:= –2048 else
for SC:= 0, SC–1 while W(0) = W(1) do W:= shiftleft W;
Test Address; Test Protection;
byte(SB):= SC(12:23);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.

Normalize Double

Same as Normalize Single performed with the register pair Wpre and W.

Jump with Register Link

If the W field <> 0, the instruction counter is stored in the W register. Following this, a jump is made to the effective address.

```
Test Address; Test Protection;
if FR(6:7) < > 0 then W:= IC;
IC:= SB; IC(23):= 0;
goto Next Instruction;
```

Exception unchanged.
Interruption: (0) undefined address or protection violation.
Note: When the W field = 0 the operation is a simple unconditional jump that leaves all registers unchanged. When the W field <> 0, the operation is a subroutine jump that places the return address in the W register. A return jump is performed as a simple jump, with the same register specified in the X field.

Skip if Register High

Compare the W register and the effective address as signed integers. If the register is greater than the address, then skip the following instruction. The register remains unchanged.

```
AR:= signed W – signed SB;
if AR > 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.

Skip if Register Low

Compare the W register and the effective address as signed integers. If the register is less than the address, then skip the following instruction. The register remains unchanged.

```
AR:= signed W – signed SB;
if AR < 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.

Skip if Register Equal

Compare the W register and the effective address as signed integers. If the register equals the address, then skip the following instruction. The register remains unchanged.

```
AR:= signed W – signed SB;
if AR = 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.

Skip if Register Not Equal

Compare the W register and the effective address as signed integers. If the register is unequal to the address, then skip the following instruction. The register remains unchanged.

```
AR:= signed W – signed SB;
if AR <> 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.

Skip if Register Bits One

Use the effective address as a mask to test selected bits in the W register. If all the selected bits are one, then skip the following instruction. The register remains unchanged.


```
for bit:= 0 step 1 until 23 do
AR(bit):= if SB(bit) = 1 then not W(bit) else 0;
if AR(0:23) = 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.
Note: When the effective address is zero, the operation skips the following instruction unconditionally.

Skip if Register Bits Zero

Use the effective address as a mask to test selected bits in the W register. If all the selected bits are zero, then skip the following instruction. The register remains unchanged.

```
for bit:= 0 step 1 until 23 do
AR(bit):= if SB(bit) = 1 then W(bit) else 0;
if AR(0:23) = 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.
Note: When the effective address is zero, the operation skips the following instruction unconditionally.

Skip if No Exceptions

Use the right-most three bits of the effective address as a mask to test selected bits in the exception register. If all the selected bits are zero, then skip the following instructions. The exception register remains unchanged.

```
AR:= positive EX;
for bit:= 0 step 1 until 23 do
AR(bit):= if SB(bit) = 1 then AR(bit) else 0;
if AR(0:23) = 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: none.
Note: When the effective address is zero, the operation skips the following instruction unconditionally.

Skip if No Protection

Use the protection key of the storage word addressed as an index to select a bit in the protection register. If the selected bit is zero, then skip the following instruction.

```
Test Address; PK:= protection key(SB);
if PR(PK) = 0 then IC:= IC + 2;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.

Jump with Interrupt Enabled
Same as Jump with Register Link, except that the interruption system is enabled first. This is a privileged instruction.

```
Test Mode; Test Address; interrupt disabled:= false;
if FR(6:7) < > 0 then W:= IC;
IC:= SB; IC(23):= 0;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) not monitor mode or undefined address.

Jump with Interrupt Disabled
Same as Jump with Register Link, except that the interruption system is disabled first. This is a privileged instruction.

```
Test Mode; Test Address; interrupt disabled:= true;
if FR(6:7) < > 0 then W:= IC;
IC:= SB; IC(23):= 0;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) not monitor mode or undefined address.

Clear Interrupt Bits
Use the effective address as a mask to clear selected interruption signals. This is a privileged instruction.

```
Test Mode;
for bit:= 0 step 1 until 23 do
if SB(bit) = 1 then IR(bit):= 0;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) not monitor mode.

Store Interrupt Register
Store the interrupt register in the storage word addressed. The interrupt register remains unchanged.

```
Test Address; Test Protection;
word(SB):= IR;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.

Load Mask Register
Insert the storage word addressed in the interrupt mask register. Bit 0 of the mask register is permanently equal to one. This is a privileged instruction.

```
Test Mode; Test Address;
IM:= word(SB); IM(0):= 1;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) not monitor mode or undefined address.

Store Mask Register
Store the interrupt mask register in the storage word addressed. The mask register remains unchanged.

```
Test Address; Test Protection;
word(SB):= IM;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.

Load Exception Register
Insert the right-most three bits of the storage byte addressed into the exception register. The storage byte remains unchanged.

```
Test Address;
SB(12:23):= byte(SB); EX:= SB(21:23);
goto Next Instruction;
```

Exception: set as defined above.
Interruption: (0) undefined address.

Store Exception Register

Extend the exception register towards the left with zeroes and store it in the storage byte addressed. The exception register remains unchanged.

```
Test Address; Test Protection;
byte(SB):= positive EX;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.

Load Protection Register

Insert the right-most seven bits of the storage byte addressed into the protection register. Bit 0 of the protection register is permanently equal to one. The storage byte remains unchanged. This is a privileged instruction.

```
Test Mode; Test Address;
SB(12:23):= byte(SB); PR:= SB(16:23); PR(0):= 1;
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) not monitor mode or undefined address.

Store Protection Register

Store the protection register in the right-most eight bits of the storage byte addressed. The left-most four bits of the storage byte are set to zero. The protection register remains unchanged.

```
Test Address; Test Protection;
AR:= positive PR; byte(SB):= AR(12:23);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address or protection violation.

Load Protection Key

Load the right-most three bits of the W register with the protection key of the storage word addressed. The left-most twenty-one bits of the W register are set to zero. The protection key of the storage word remains unchanged.

```
Test Address;
W:= positive protection key(SB);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) undefined address.

Store Protection Key

Store the right-most three bits of the W register into the protection key of the storage word addressed. The register remains unchanged. This is a privileged instruction.

```
Test Mode; Test Address;
protection key(SB):= W(21:23);
goto Next Instruction;
```

Exception: unchanged.
Interruption: (0) not monitor mode or undefined address.

Input/Output

An input/output operation is initiated, if the selected device is available. If the device is busy or disconnected, the operation is rejected. This is indicated in the exception register. The detailed execution of the four basic commands: read, write, sense, and control is defined in Chapter 8.

```
Test Mode;
device:= SB(0:17);
EX(22):= if disconnected (device) then 1 else 0;
EX(23):= if busy(device) then 1 else 0;
if EX(22:23) <> 0 then goto Next Instruction;
if SB(22:23) = 0 then W:= device buffer(device);
comment: sense command;
if SB(23) = 1 then device buffer(device):= W;
comment: write or control command;
goto Next Instruction;
```

Exception: (22) disconnected, (23) busy.
Interruption: none.

Autoload Word

Four 6-bit characters with odd parity from device number 0 are loaded into the storage word addressed, and the protection key of the storage word is set to zero. This is a privileged instruction. It repeats input, if the status bit 0 (end of buffer) is set. The computer is, however, set in the reset state, if the loading device is disconnected or if any other status bits are set during input.

```
Test Mode;
comment: save the load address in SF and BF and read 4 characters
into AR from device 0;
SF(24:35):= SB(0:11); BF(24:35):= SB(12:23);
AR:= 0;
```

```
for SC:= 1 step 1 until 4 do
begin
Start Input:
        SB:= 2; comment: read command to device 0;
        if disconnected(0) then goto Reset System;
        if busy(0) then goto Start Input;
        EX(22:23):= 0;
Wait Input:
        SB:= 0; comment: sense command to device 0;
        if disconnected(0) then goto Reset System;
        if busy(0) then goto Wait Input;
        EX(22:23):= 0;
        SB:= device buffer(0);
        if SB(0) = 1 then goto Start Input;
        comment: end of buffer status;
        AR(0:17):= AR(6:23); AR(18:23):= 0;
        for bit:= 0 step 1 until 23 do
        AR(bit):= if AR(bit) = 1 or SB(bit) = 1
                  then 1 else 0;
        if SB(0:11) <> 0 then goto Reset System;
        comment: other status bits;
end;
        SB(0:11):= SF(24:35); SB(12:23):= BF(24:35);
        Test Address;
        word(SB):= AR(0:23); protection key(SB):= 0;
        goto Next Instruction;
```

Exception: (22) zero, (23) zero.
Interruption: none.

APPENDIX

A.1. Reserved Storage Locations

Storage Location:	Use:
0	working register 0
2	working register 1
4	working register 2
6	working register 3
8	interrupt number
10	address of interrupted program
12	address of interrupt response program
14	address of start key program

A.2. Numeric Instruction Codes

Numeric Code:	Mnemonic Code:	Numeric Code:	Mnemonic Code:
0	AW	32	CI
1	IO	33	AC
2	BL	34	NS
3	HL	35	ND
4	LA	36	AS
5	LO	37	AD
6	LX	38	LS
7	WA	39	LD
8	WS	40	SH
9	AM	41	SL
10	WM	42	SE
11	AL	43	SN
12	ML	44	SO
13	JL	45	SZ
14	JD	46	SX
15	JE	47	IC
16	XL	48	FA
17	BS	49	FS
18	BA	50	FM
19	BZ	51	KS
20	RL	52	FD
21	SP	53	CF
22	KL	54	DL
23	RS	55	DS
24	WD	56	AA
25	RX	57	SS
26	HS	58	(not used)
27	XS	59	(not used)
28	PL	60	(not used)
29	PS	61	(not used)
30	MS	62	(not used)
31	IS	63	(not used)

A.3. Instruction Execution Times

The execution times listed apply to direct addressing. For address modification, add the following:

relative addressing:	0.5 usec
indexing:	0.5 usec
relative addressing and indexing:	1.5 usec
indirect addressing (codes 0-31, 48-63):	1.5 usec
(codes 32-47):	1.0 usec

AM	2.0	CI	4.0 + 0.5*shifts	SO	3.5
AL	1.5	CF	5.0 + 0.5*shifts	SZ	3.0
AC	2.5			SX	3.0
		FA	11.0	SP	3.0
HL	3.0	FS	11.0		
HS	4.0	FM	26.0	JE	2.5 or 3.0 (with link)
RL	3.0	FD	27.0	JD	2.5 or 3.0 (with link)
RS	4.0			IC	2.5
RX	4.5	LA	3.0	IS	4.0
DL	5.0	LO	3.0	ML	3.0
DS	7.0	LX	4.0	MS	4.0
				XL	3.0
BZ	3.0	AS	3.0 + 0.5*shifts	XS	4.0
BL	3.0	AD	3.5 + 0.5*shifts	PL	3.0
BA	3.5	LS	3.0 + 0.5*shifts	PS	4.0
BS	3.5	LD	3.5 + 0.5*shifts	KL	3.0
		NS	5.0 + 0.5*shifts	KS	4.0
WA	3.0	ND	5.5 + 0.5*shifts	IO	4.0 (rejected)
WS	3.0				5.0 (read)
WM	15.5	JL	2.5 or 3.0 (with link)		7.0 (sense)
WD	17.0	SH	3.0		6.5 (write)
		SL	3.0		6.5 (control)
AA	5.0	SE	3.0		
SS	5.0	SN	3.0		

INDEX

add floating 66
add integer byte 62
add integer double word . . 64
add integer word 62
addition 19,23,27,54
addressing of device 36
addressing of storage 15ff
addressing of register 16
address modify instruction . 18
AE register 53
AF register 53
ARBR register 53
AR register 51ff
autoload key 45,55
autoload word 45,77

BE register 53
BF register 53
BR register 51ff
busy device 36
byte addressing 15
byte arithmetic 19

carry 21
clear interrupt bits 74
continue key 47
control command 39,41
control panel 44ff,55
convert floating to integer . 65
convert integer to floating . 65
core store control 48
core store error 44
CPU error 44
CPU running 44

data channel 35ff
data formats 15
data overrun 42
device buffer 35
device command 36ff
device number 36
disabling of interrupts 33
disconnected device 36
displacement 17
display of registers 47ff
divide floating 67
divide integer word 63
division 20,23,27
double-length registers 16
double word 15

effective address 17,56
enabling of interrupts 33
exact arithmetic 26ff
exception register 21,25,37
exchange register and store . 60
execution times 81
exponent 22
EX register 52ff
external interruption 34,40

floating-point arithmetic . . 22ff
fraction 22
FR register 51ff
full precision 24ff

high-speed data channel . . . 41ff

IC register 52ff
IM register 52ff
index registers 16ff
indirect addressing 17

input/output 35ff,77
instruction exception 33
instruction execution 58ff
instruction execution times . 81
instruction fetch cycle 55
instruction format 17
instruction set 49
interruption 32ff,55
interrupt number 32
interrupt register 32
interval timer 43
IR register 52ff

jump selector 48
jump with interrupt disabled 74
jump with interrupt enabled . 74
jump with register link 71

load address 59
load address complemented . 59
load byte with zeroes 61
load double register 61
load exception register 75
load half register 59
load integer byte 62
load mask register 75
load protection key 76
load protection register 76
load register 60
local/remote control 46
logical and 69
logical exclusive or 69
logical or 69
low precision 24ff
low-speed data channel 35ff

machine error 44
MAR computer controlled . . 48
MAR manually controlled . . 48
mask register 32
microcommand 48
microprogram store 48
mnemonic operation codes . . 49ff
modify next address 58
monitor control 28ff
monitor mode 29ff
MPS control 48
MPS error 48
multiplication 20,23,27
multiply floating 67
multiply integer word 63

normalize double 71
normalized number 22ff
normalize single 71
normal mode 47
not operator 54
number conversion 26
number representation 19,22
numeric operation codes . . . 80

operation byte 17
operation codes 80
overflow 21,25,33

parity error 44
precision modes 24ff
PK register 51ff
positive operator 54
power on 44,55
privileged instructions 28ff
program interruption 32ff
protection 28ff,56
protection key 28,41
protection register 28
PR register 52ff

quotient 20,23

range of numbers 19,21,22
relative addressing 17
register structure 51ff
read command 37
remainder 20,23
reset key 44,55
result register 16ff
rounding 23ff,57

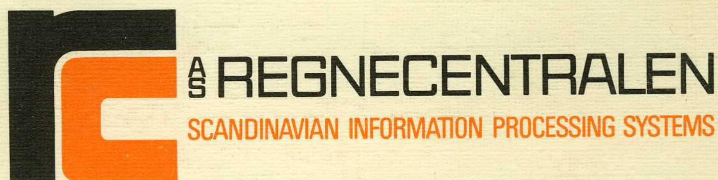
SBexp register 53
SB register 51ff
SC register 52ff
SE register 53
SF register 53
sense command 37,41
shift double arithmetically . . 70
shift double logically 71
shiftleft operator 54
shiftright operator 54
shift single arithmetically . . . 70
shift single logically 70
signed operator 54
sign of numbers 19ff,22
single instruction key 47
single microinstruction key . 47
skip if no exceptions 73
skip if no protection 73
skip if register bits one 72
skip if register bits zero 73
skip if register equal 72
skip if register high 71
skip if register low 72
skip if register not equal . . . 72
start key 45,55
status bits 38
storage addressing 15ff
storage protection 28ff
store double register 61
store exception register 76
store half register 59
store interrupt register 75
store mask register 75
store protection key 77
store protection register . . . 76
store register 60
subtract floating 66
subtract integer byte 62
subtract integer double word 64
subtract integer word 62
subtraction 19,23,27,54

task mode 29ff
technical mode 47

unassigned operation codes 33,80
undefined address 33
underflow 25,33

Wexp register 53
Wfrac register 53
Wpre register 53
W register 52ff
word 15
working registers 16ff
write command 38

zero representation 22



HEADQUARTERS: FALKONER ALLÉ 1 · DK-2000 COPENHAGEN F · DENMARK
PHONE: (01) 10 53 66 · TELEX: 6282 RCHQ DK · CABLES: REGNECENTRALEN

AUSTRIA
BENELUX
DENMARK
GERMANY
NORWAY
SWEDEN