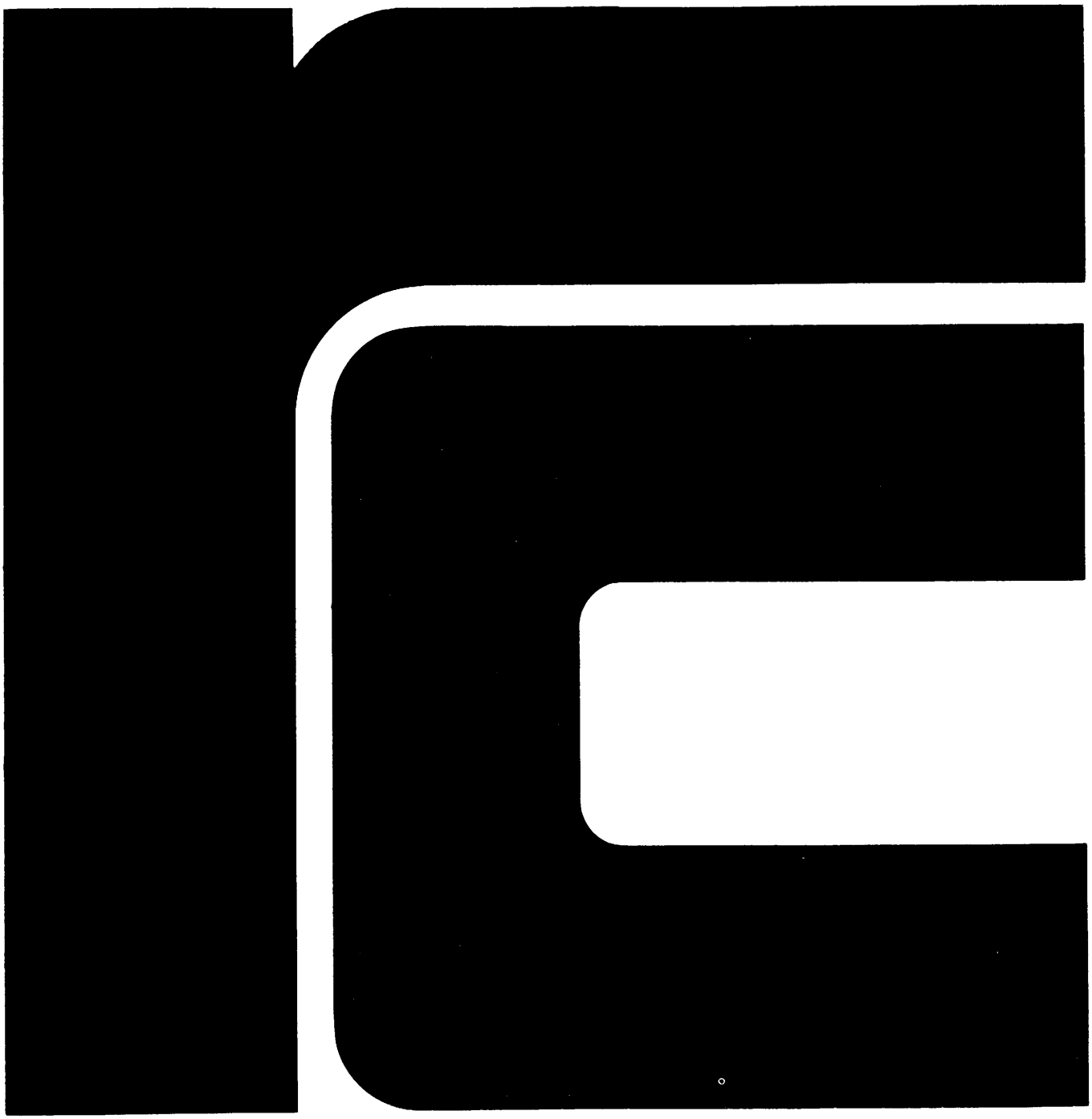
Reference Manual



8000

RCSL: 31-D383
Author: Finar Mossin
Editor: Pierce Hazelton
Edited: January 1976

RC 8000 COMPUTER REFERENCE MANUAL

The technical information in this document, whilst correct at the time of publication, is liable to change without notice.

Keywords: RC 8000, Computer, Reference Manual

Abstract: This manual provides basic programming and operating information for programmers and users of the RC 8000 computer.

CONTENTS 0

Implementation

Large-scale integrated circuits extensively used.
Compromise between hardwired logic and microprogramming, balancing flexibility and speed.

Bus structure

Asynchronous unified bus.
Parallel data lines (24 bits + 3 parity bits) and address lines (23 bits + 1 parity bit).

Primary memory

Magnetic core memory with 800 nanosecond cycle time. Other memory types as required.
Basic module of 32 768 words. Direct addressing of up to 8 388 608 words. Each word contains 24 data bits and 3 parity bits.

Peripherals

Complete range of input/output devices, interfaced through peripheral processors or programmable front end. Both processor types are connected to the unified bus.

Working registers

Four 24-bit working registers, three of which also function as index registers.
The registers are addressable as the first four words of the primary memory.

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.
Dynamic relocation through use of modified base register technique.

Instruction execution times

0.9 to 2.3 microseconds typically (including access and address modification time, and depending on primary memory cycle time).

Instructions

Comprise 64 function codes, each working on 4 registers, with 16 address calculation modifications and a 12-bit displacement.
Arithmetic includes add, subtract, multiply, and divide.
Data manipulation aided by byte operations and word comparison.
Logical operations permit setting and testing of single bits.
Escape facility permits programmed actions on any or all instructions.

Protection system

Privileged instructions and memory protection associated with a monitor mode ensure complete monitor control.

Interruption system

Program interruption system with 32 priority levels, expandable in modules of 8 up to 64 levels.
Assignment of levels and disabling of interrupts under program control.
Interrupt response time, including saving of registers, is 10 microseconds. Return from Interrupt instruction requires 10 microseconds to re-establish all registers.

Input/output system

All peripherals, except the primary memory, are connected to the unified bus by means of standardized block-oriented controllers, which perform all input and output functions under the control of their own channel programs.

Data is transferred directly between the controllers and the primary memory. An asynchronous, fully interlocked request/acknowledge communication technique is employed.

DESIGN FEATURES 2

Operand length 2.1

Arithmetic and logical operands. The basic arithmetic or logical operand is a 24-bit word. Double-length operands of 48 bits satisfy the requirements of engineering computation and administrative data processing.

Byte handling. Direct addressing of 12-bit bytes permits efficient packing of data.

NOTE: The expression "byte" is used in this manual to refer to 12-bit halfwords, whereas "character" refers to 8-bit bytes containing any information.

Working registers and addressing 2.2

Register structure 2.2.1

The RC 8000 has four working registers, three of which also function as index registers.

In computers where there is a sharp distinction between the working register and index registers, programming often becomes awkward. Since all operations destroy the previous content of the working register, the programmer is forced to make numerous storage operations in order to save and restore intermediate results. Empty transfers to memory are also required when an index register must be modified by the content of the working register.

The register structure of the RC 8000 eliminates this deficiency. By extending the number of working registers to four and removing the distinction between working and index registers, the full instruction set is made available for immediate address modification, while the number of empty transfers of registers to memory is greatly reduced.

Program relocation 2.2.2

Efficient relocation requires that programs can be written in such a way that their execution is independent of their location. This is achieved in the RC 8000 in two ways.

First, the instruction format contains a bit that specifies relative addressing. It indicates that the address part of the instruction is to be interpreted relative to its current location in the primary memory. This permits relocation of programs during loading.

Second, the base register structure defines the current relation between the logical address as seen by the program and the actual physical address, thereby ensuring that programs also can use saved address information after restarting in new memory areas.

Escape facility 2.3

The RC 8000 is provided with an escape facility, implemented by means of an escape mode and an escape mask, which permits independent supervision of instruction execution as well as programmed emulation of virtual memory, instruction sets, and the like.

Monitor control 2.4

In a multiprogramming system, where many concurrent tasks are performed, it is vital that erroneous programs be prevented from interfering destructively with other programs. The various tasks are therefore co-ordinated by a monitor program that has complete control of the system. Monitor control in the RC 8000 is guaranteed by memory protection, privileged instructions, and program interruption.

Memory protection. An erroneous program may attempt to destroy data or instructions within other programs. Mutual memory protection is accomplished in the RC 8000 through limit registers, so that a program can only alter the contents of memory locations in its own area. The remainder of the memory is divided into a read-accessible lower part and a read-protected upper part. The modification of the base register structure ensures that the lower, read-accessible part is addressed independently of dynamic program relocation. Any attempt to violate the protection system leads immediately to a program interruption.

Privileged instructions. Further protection is achieved through 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, memory protection, and dynamic program relocation.

Program interruption. Multiprogrammed computers must respond quickly to exceptional events. In the RC 8000 this is made possible by a program interruption system that can register up to 64 signals simultaneously. Any of these signals interrupts the current program immediately and starts the monitor program.

Input/output system 2.5

The input/output system is based on a unified bus, i.e. a common bus for communication between all devices connected to it, none of which has a special status. Besides permitting the implementation of a wide range of systems, including multiprocessing systems, the unified bus facilitates communication with other systems and provides a basis for implementation of other bus structures.

The connection of peripheral devices is standardized in such a way that the central processor is unaware of the types of devices attached to it. All peripheral devices except the primary memory are connected to the bus via standardized device controllers (peripheral processors). Data transfers between the central processor and the peripheral processors are handled by a single input and a single output operation.

In order not to suspend program execution while an input or output operation is in progress, the direct transfer of data between processors is minimized; thus the peripheral processor, as soon as it is started, will fetch its commands from the channel program in the primary memory and execute them without engaging the central processor, which continues with its program.

When the central processor attempts to initiate an input or output operation, the peripheral device may reject it. Information about this as well as other exceptional events is made available in a status register, which can be used by the program in order to take appropriate action. Device status will be transferred to the central processor when a data transfer is completed.

Peripheral devices may also be connected to the bus in a more primitive manner, i.e. without the channel program concept. In this case, the device

controller will be regarded as a set of registers, to which the central processor transfers control information and from which it obtains device information by means of the input and output instructions.

Bus control 2.5.1

Since the bus is shared by numerous devices, only one may have control of it at a time. This device is called the bus master, and the device which it addresses is called the slave.

When a device that is capable of being a bus master (viz. a central unit, primary memory, or peripheral processor) desires to obtain control, it sends a request to the bus control unit.

The control unit responds with a select signal, which is daisy-chained through all the devices on the bus. The first device having sent a request breaks the chain and returns an acknowledgement, completing the selection procedure. If the control unit does not receive an acknowledge signal, it generates a bus timeout, after which the selection procedure may be repeated.

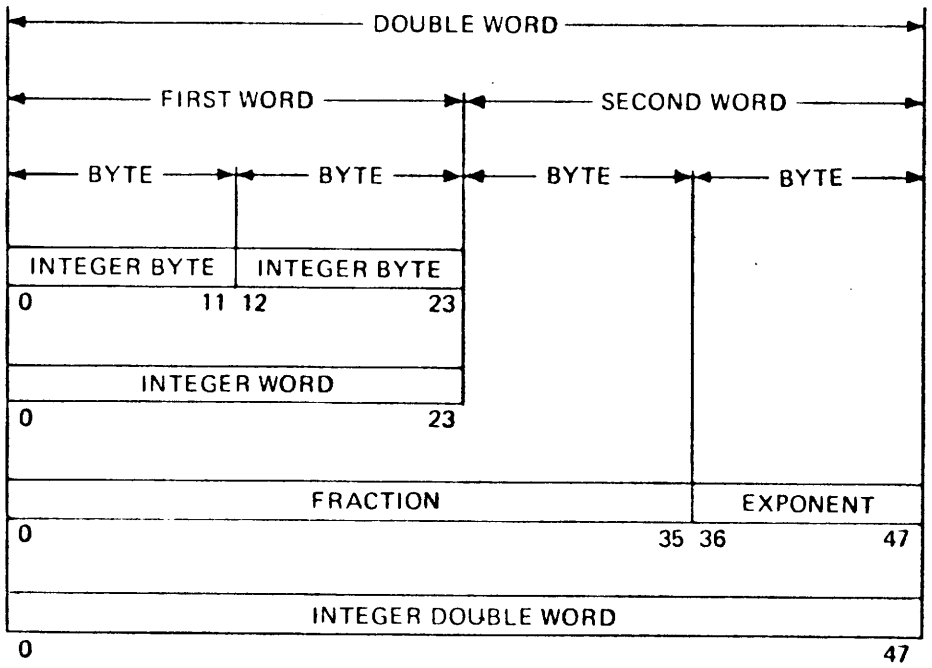
As soon as the current master completes its transfer and releases the bus, the selected device becomes the new master and sends a busy signal while using the bus. During this time the next bus master is selected.

Bus communication 2.5.2

To facilitate interfacing with other systems, an asynchronous fully interlocked technique is used for bus communication. This so-called handshake technique, in which each request that is sent by a master must be acknowledged by the slave to complete the transfer, permits operations between devices having different response times.

Data formats 3.1

The data structure of the RC 8000 is shown in the following figure:



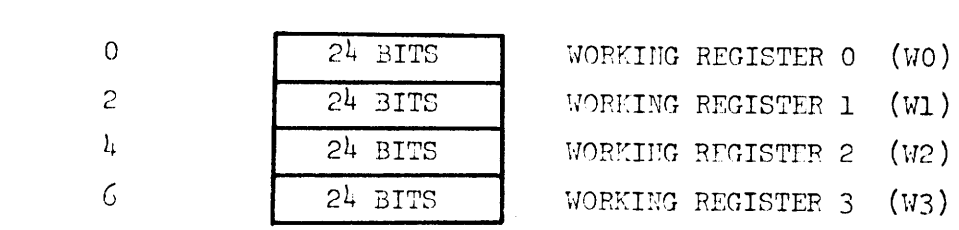
The basic arithmetic or logical operand is an integer of 24 bits. Data is packed with two halfwords per word. The 12-bit halfwords, which are called bytes, are directly addressable. Double words are used to represent integers of 48 bits and floating-point numbers with 36-bit fractions and 12-bit exponents.

Working registers 3.2

The register structure includes four 24-bit working registers, one of which is specified as the result register in each instruction. Three of the working 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 primary memory. The programmer can therefore per-

form operations directly between two registers by specifying a memory address between 0 and 7. It is also possible to execute instructions stored in the working registers.



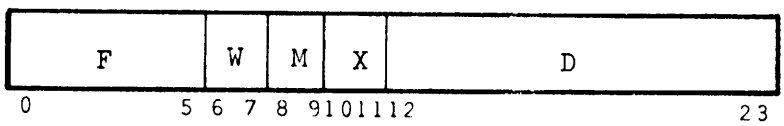
Two adjacent working registers can be used to hold a double-length operand of 48 bits. In double-length operations, the four registers are connected cyclically as follows:

W3 concatenate with W0
W0 concatenate with W1
W1 concatenate with W2
W2 concatenate with W3

These connections are established by specifying the second register of the concatenation in the instruction format.

Instruction format 3.3

The instruction format is divided into an operation byte and an address byte, each containing 12 bits:



Bits 0:5 F field. Contains the function code, specifying one of sixty-four basic operations.

Bits 6:7 W field. Specifies one of the four working registers as the result register.

Bits 8:9 M field. Specifies one of four address modes, used to control generation of the effective address (see below).

Bits 10:11 X field. Selects the current index register. Only working registers W1, W2, and W3 act as index registers (X field = 0 indicates no indexing).

Bits 12:23 D field. Contains a truncated address, specifying a displacement from -2048 to +2047 bytes within the program. This is adequate for the majority of addresses, but not sufficient for direct addressing of the entire memory.

A full address of 24 bits is formed by means of the displacement, D, in conjunction with the content of an index register, X, and the content 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 on one level. The indirect address fetched from the memory 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.

Note

In the notation used in this section as well as Chapters 4 and 5, indirect addressing is indicated by parentheses, and comments are preceded by semicolons. The mnemonic operation codes are defined in Chapter 10, which contains a complete list of the instructions and their function codes.

Use of the effective address as an operand 3.3.1

For some function codes, the effective address is used directly as an operand. This is done in three different ways.

The effective address or its two's complement can be assigned to the addressed register.

The content of the working register can be compared with the effective address (word comparison) in several ways, the result of the comparison determining whether the following instruction is to be executed or skipped.

The effective address can define a number of shifts to be performed on the addressed register.

Use of the effective address to refer to memory locations 3.3.2

For other function codes, the effective address is used to refer to memory locations.

Memory 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 the right half of a word. In double-word operations, the right-most bit in the effective address is also ignored; the word thus specified is the second word of the operand.

The effective address is used to refer to memory locations in register transfer instructions as well as instructions for arithmetic and logical operations. Here, the effective address is treated as a logical address.

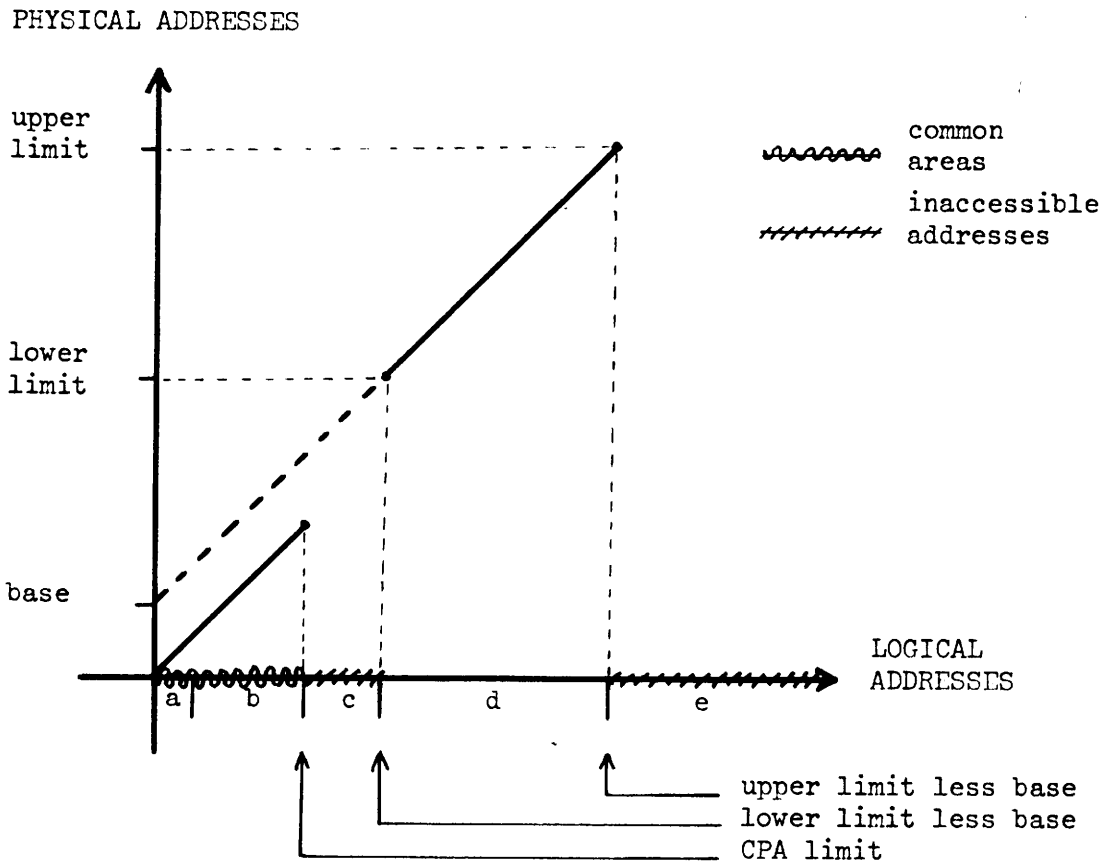
If the effective address lies within the current values of the limit registers less the base, the content of the base register will be added to it before the memory is addressed.

If, on the other hand, the effective address lies within the common protected area, CPA, i.e. is less than the current value of the CPA limit register, then the effective address will be used directly to address the memory, but only read-access will be allowed.

Any attempt to write-access the CPA as well as any reference to areas outside the ones just mentioned, which are the only logically existent areas, will result in a program interruption.

The effective addresses 0:7 will, as previously mentioned, always refer to the working registers.

The following figure shows the physical address as a function of the logical address for a normal job process:



The normal division of the logical address area is as follows:

a. 0 <= logical address < 8
   Common area for references to the four working registers using absolute addresses (physical address = logical address).
   Normal access.

b. 8 <= logical address < CPA limit
   Common protected area (CPA) for references to system parameters using absolute addresses (physical address = logical address).
   Read-access only.

c. CPA limit <= logical address < lower limit - base
   Non-accessible area.

d. lower limit - base <= logical address < upper limit - base
Program's own job area.
Relocatable (physical address = logical address + base).
Normal access.

e. upper limit - base <= logical address
Non-accessible area.

Jump instructions 3.3.3

The jump instructions represent a special kind of memory reference, as they transfer program control to the instruction pointed out by the effective address.

The effective address is treated as a logical address as described above, and program execution is regarded as a read-access.

Subroutine jumps are implemented as follows: If the W field is different from zero, the logical address of the return point, i.e. the instruction following the subroutine jump, is placed in the addressed working register. A jump is then made to the effective address.

At the end of the subroutine, a return jump is made as a simple unconditional jump.

Modify Next Address instruction 3.3.4

The AM instruction, which modifies the displacement in the following instruction by its own effective address, provides various possibilities of address modification.

One use of this instruction is for direct indexing with the content of any memory location, for example:

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

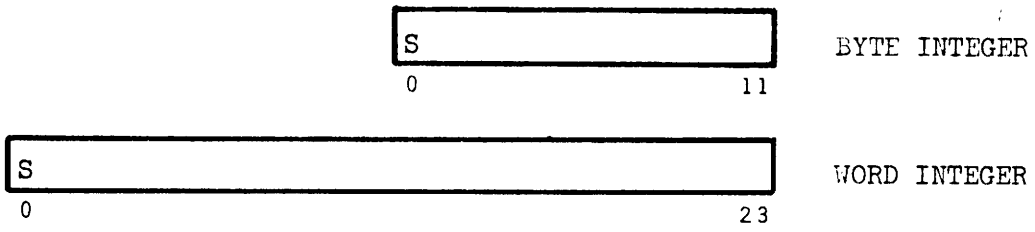
The effective address of the AM instruction is A1 = word (X1 + D1). This is used to modify the displacement D2 in the following JL instruction to produce an effective address A2 = word (X1 + D1) + D2.

Another use of the AM instruction is for multi-level indirect addressing, for example:

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

Number representation 4.1

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 right-most bit.

Byte arithmetic 4.2

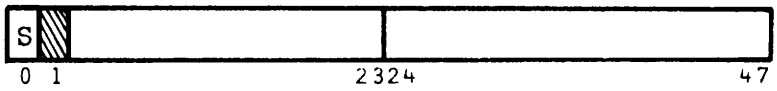
A signed integer represented by a 12-bit byte must be confined to the following range:

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

The instruction Load Integer Byte (BL) 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 (BA and BS) perform addition to or subtraction from a working register with a byte fetched from the memory and extended to 24 bits. The instruction Store Half Register (HS) 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.

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

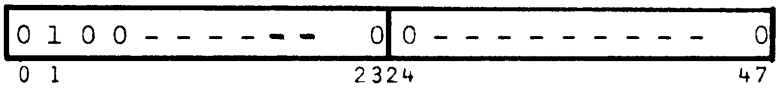


A double-length product will normally consist of a sign bit plus at most 46 digits. In this case, bit 1 in the double register will be identical with the sign bit.

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 here:



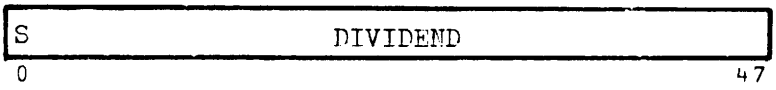
It should be noted that in this representation of double-length integers, bit 24 does not function as a sign bit, but contains a significant digit.

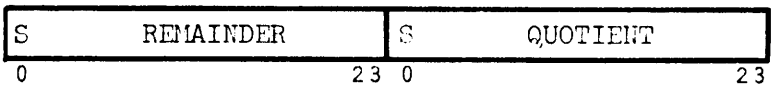
The content of a double register can be divided by the content of a memory word. The dividend is then replaced by a 24-bit remainder in the left-hand register and a 24-bit quotient in the right-hand register. A non-zero remainder satisfies the following requirements:

(1) dividend = divisor * quotient + remainder

(2) 0 < abs (remainder) < abs (divisor)

(3) sign (remainder) = sign (dividend)





Overflow and carry indication

Arithmetic operations indicate a normal or an exceptional result by setting the right-most two bits of a 3-bit register called the exception register. Physically, the exception bits are the last three bits in the CPU status register, but they are treated by special instructions as a logically independent register. Thus the exception register can be tested by a single instruction, Skip if No Exceptions (SX).

After a normal result, exception bits 22 and 23 are set to zero. An integer overflow will set exception bit 22 to one, and may provoke a program interruption as defined in Chapter 7.

Here it should be noted that a program interruption presupposes that the mask for arithmetic interrupts contains the bit that enables interrupts for integer arithmetic.

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, viz.

$-2^{**}23=-8\ 388\ 608<=$integer word$<=8\ 388\ 607=2^{**}23-1$

Note that multiplication can never produce overflow.

(2) The instruction Load Address Complemented (AC) 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 modulo $2^{**}24$ 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.

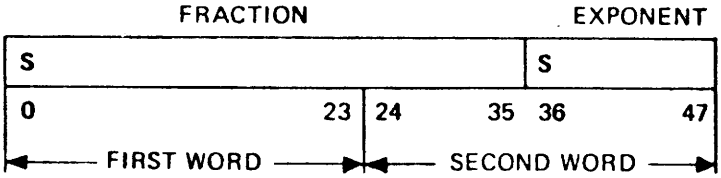
The exception register, then, has the following meaning after an in-

teger arithmetic operation:

Bit	Meaning
21	(unchanged)
22	integer overflow
23	integer carry

Number representation 5.1

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 memory 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 are a signed exponent, also in the two's complement form.

The range of floating-point numbers is the following:

$$-1*2^{**}2047 <= F <= -0.5*2^{**}(-2048) \quad \text{F negative}$$

$$F = 0*2^{**}(-2048) \quad \text{F zero}$$

$$0.5*2^{**}(-2048) <= F <= 1*2^{**}2047 \quad \text{F positive}$$

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 (SH,

SL, SE, or SN) 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;
```

Arithmetic operations 5.2

Before an arithmetic operation, the fractions are placed left-justified in anonymous 38-bit registers and extended to the right with two zeros. 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 below.

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 below).

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 below. The remainder is thrown away.

Normalization and rounding 5.3

If the resulting 38-bit fraction is zero, a floating-point zero with ex-

ponent -2048 is delivered as the 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 satisfy the following inequalities:

$$\text{abs}(f2 \text{ shifted}) <= 1 * 2 ** (-3)$$

$$\text{abs}(f1 + \text{or} - f2 \text{ shifted}) >= 0.5 - 1 * 2 ** (-3) = 3/8$$

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 fraction lies in the interval

$$0.25 <= \text{abs}(f1 * f2) <= 1$$

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.

Underflow, overflow, and non-normalized operands 5.4

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 (see Section 4.4) to one, and may provoke a program interruption as defined in Chapter 7.

Here it should be noted that a program interruption presupposes that the mask for arithmetic interrupts contains the bit that enables interrupts for floating-point arithmetic.

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 may provoke an interrupt as defined in Chapter 7. This is also true, if zero is divided by zero.

Considering the enormous range of floating-point numbers, both underflow and overflow will usually indicate a programming error.

No check is made of 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:

Bit	Meaning
21	(unchanged)
22	floating-point underflow or overflow
23	0

Number conversion 5.5

The instruction Convert Integer to Floating (CI) 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 may occur, if the resulting exponent exceeds the 12-bit range.

The instruction Convert Floating to Integer (CF) 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 may occur, if the resulting integer exceeds the 24-bit range.

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)
```

Exact arithmetic with floating-point instructions 5.6

The floating-point arithmetic may be used to simulate exact arithmetic with 35-bit 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;
```

When a program is started, the escape function is normally inactive, as the CPU status bit "escape mode" is logical 0. In the escape mode, i.e. when this status bit is set, the escape function is controlled by a mask, which comprises bits 6:11 of the status register. Each of these bits represents an instruction characteristic as follows:

Bit	Characteristic
6	privileged instruction
7	modifies the instruction counter
8	involves more than one memory word
9	modifies a memory word
10	refers to a memory word
11	auxiliary function

Each instruction is described by a bit pattern that indicates its characteristics (see Chapter 10). The six bits of this pattern correspond to bits 6:11 of the status register, i.e. the escape mask. If a mask and an instruction contain one or more bits in corresponding positions, an escape routine will be activated to perform some desired action.

Example

The bit pattern of the Load Double Register instruction (DL) is 001010. Bit 10 = 1 indicates that DL refers to a memory location. Bit 8 = 1 indicates that the reference is to a double word operation.

If, for example, the escape facility is being used to emulate virtual memory, the mask will contain bits 9 and 10, and an attempt to execute DL will result in an escape.

If, on the other hand, the escape facility is being used for tracing, the mask will contain bit 7, and the DL instruction will be executed in the normal way.

An escape normally takes place after generation of the effective address. An exception to this is indirect addressing, where the instruction is treated as an instruction that refers to a memory location. Instructions

with indirect addressing can thus occasion two escapes.

When an escape occurs, return information is saved starting from a pre-defined escape address. This information contains the working registers, CPU status, instruction counter, and effective address.

The working registers are then initialized with information concerning the escape situation, the escape mode status bit is reset, so that the escape function is once more inactive, and program execution continues starting from the word immediately following the saved return information.

Example

When a programming error occurs, it is often helpful to obtain information about the program sequences that led up to it. This can be done using the escape facility in the following manner.

The program is first augmented with an escape routine, and then run in the escape mode with a mask containing bit 7, i.e. for instructions that modify the instruction counter.

For every escape, the routine will save the desired sequencing information cyclically in a buffer. When the programming error occurs, the buffer will contain information about the events leading up to it.

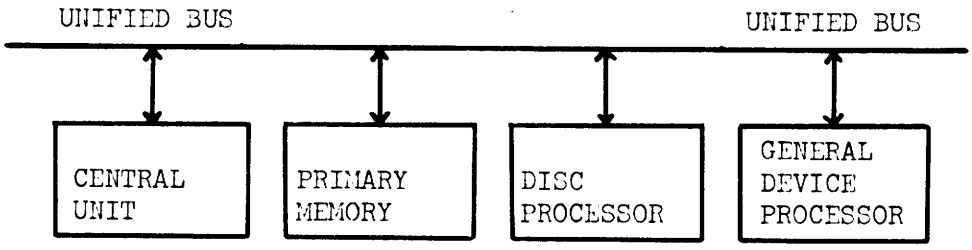
When the desired action has been performed, a Return from Escape instruction (RE) is executed in order to return to resumption of the interrupted instruction. The effective address of the RE instruction points to the saved return information.

By modification of this return information, or by pointing to other, similar information, the escape return can be used to modify the escape mode and/or escape mask, mask for arithmetic interrupts (as defined in Chapter 7), working registers, and instructior counter.

In course of preparation.

Main characteristics 8.1

The input/output system is based on a common bus for communication between all central units, primary memories, and peripheral device controllers, none of which has a special bus status.



Input/output devices on the bus as regarded as sets of registers. The only way to communicate with a device is to transfer data to and from these registers.

Device control functions are performed by addressing a device register and transferring the appropriate bit pattern to it. Device status is checked by addressing the status register of the device and transferring the content to the central processor. The current bus master (see Chapter 2) interrupts a central processor on the bus by addressing and transferring its interrupt number to a specific register in that processor.

For further information, the reader should consult the device controller handbook (RCSL 30-M4).

Input and output operations 8.2

All input and output operations are handled by two instructions, Data In and Data Out, which have the standard instruction format (see Chapter 3). Here, the W field selects the working register to be connected to the bus, while the effective address of the instruction is used to address the device register (see below). The basic bus communication technique used in these operations is described in Chapter 2.

Data In instruction 8.2.1

This instruction is used for input operations, i.e. whenever data is to be received from a device address on the bus. The content of the addressed device register is transferred to the specified working register.

The master addresses the slave and sends a ready signal. If the address is correct, the slave places the data on the bus and sends an acknowledge signal. The master receives the data and this signal and checks the data for parity, completing the transfer.

If the received data contains a parity error, the master sets the parity error status bit and generates an interrupt.

If the slave is busy, it sends a not-acknowledge signal, rejecting the operation. The master, receiving this signal, sets the communication error status bit and generates an interrupt.

If no signal whatsoever is received, because the address contains a parity error or simply does not exist, the master sets the timeout status bit and generates an interrupt.

Data Out instruction 8.2.2

This instruction is used for output operations, i.e. whenever data is to be sent to a device address on the bus. The content of the specified working register is transferred to the addressed device register.

The master addresses the slave, places the data on the bus, and sends a ready signal. If the address is correct, the slave checks the received data for parity and sends an acknowledge signal, completing the transfer.

If the data contains a parity error, the slave sends a not-acknowledge signal, rejecting the operation, and sets its own parity error status bit.

If the slave is busy, it also sends a not-acknowledge signal.

The master, receiving this signal, sets the communication error status bit and generates an interrupt.

If no signal whatsoever is received, because the address contains a parity error or simply does not exist, the master sets the timeout status bit and generates an interrupt.

Exception indication 8.2.3

The so-called exception bits of the status register have the following meaning after an operation:

Bit	Input	Output
21	bus parity error	(unchanged)
22	bus timeout	bus timeout
23	bus communication error (device dependent)	bus communication error (device dependent)

Standardized block-oriented device controllers 8.3

Standardized block-oriented controllers, such as the disc processor and general device processor, are started by means of an output operation, which addresses the controller as described below. Here, the content of the working register is irrelevant.

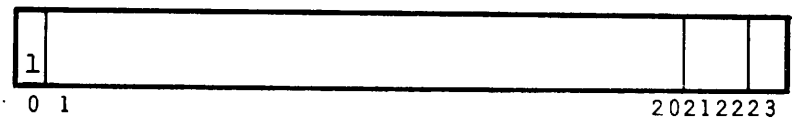
Once started, the controller fetches its commands from the channel program in the primary memory and executes them without engaging the central processor.

Data to be read from or written to a device is transferred directly between the device controller and the primary memory.

The channel program is normally terminated by a STOP command, which transfers the standard status information to the primary memory and interrupts the controlling central processor.

Device address 8.3.1

Device addresses have the following format:



Bit 0 Logical 1, indicating I/O address. This bit is set by the Data Out instruction.

Bits 1:20 Device address. Bits 1:20 are also used to calculate the device description address (see below).

In the case of multi-device controllers, the address is divided into a main device field and a sub-device field. In the disc storage system, for example, bits 1:18 contain the binary number of the addressed disc processor, preceded by zeros, and bits 19:20 the logical number of one of the four disc drives.

The function of direct controller commands is defined only by the effective address; the data transferred by the instruction is irrelevant.

Bits 21:22 Device function. Bits 21:22 have the following meaning:

00	START CHANNEL PROGRAM
01	RESET DEVICE
10	(unassigned)
11	(unassigned)

START CHANNEL PROGRAM causes the addressed controller to start its channel program by fetching the first word of the device description using the device description address (see below). During program execution the controller will not accept further START CHANNEL PROGRAM commands.

RESET DEVICE causes the addressed controller to enter an idle and unassigned state, in which it awaits addressing and can generate no interrupts.

Bit 23 Irrelevant.

Device description 8.3.2

The address of the device description is calculated using the device address (bits 1:20) as follows: device base + device address x 8

The device base, which is common to all devices connected to the controller, has an absolute address in the primary memory.

The device description contains the following:

1st word: Start of channel program. Address of the first channel program command.

2nd word: Status address. First address of the area in the primary memory to which the standard status information is to be

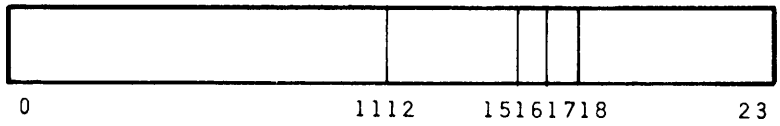
transferred at the end of the program.

3rd word: Interrupt destination. I/O address of the central processor to be interrupted at the end of the program.

4th word: Interrupt level. Current interrupt number to be transferred to the central processor.

Channel program 8.3.3

Channel programs consist of sequences of three-word commands, each of which contains a channel command and two parameters. The command proper (the first word) has the following format:



Bits 0:11 Irrelevant.

Bits 12:15 Command field. Contains the function code.

Bit 16 D field. Indicates data chaining.

Bit 17 S field. Indicates skipping.

Bits 18:23 Modifier field. Used to change the effect of the basic command.

Command field

The basic commands can be divided into three groups according to parameter structure: some require two parameters, others only one, still others none whatsoever.

The parameter FIRST CHAR ADDRESS specifies the start address of the memory area to or from which characters are to be transferred or fetched.

The parameter CHAR COUNT specifies the maximum number of characters to be transferred or fetched.

The parameters DATA 1 and DATA 2, which are device dependent, specify data areas.

For some device controllers, only three bits of the command field are interpreted, in which case the bit pattern x111 indicates STOP.

Bits 12:15	Basic Command	Parameter 1	Parameter 2
0000	SENSE	FIRST CHAR ADDRESS	CHAR COUNT
0001	READ	"	"
0010	CONTROL	"	"
0011	WRITE	"	"
0100	WAIT	(irrelevant)	(irrelevant)
0101	(unassigned)	"	"
0110	CONTROL NO PARAM	"	"
0111	(unassigned)	"	"
1000	(unassigned)	DATA 1	(irrelevant)
1001	"	"	"
1010	"	"	"
1011	"	"	"
1100	(unassigned)	DATA 1	DATA 2
1101	"	"	"
1110	"	"	"
1111	STOP	(irrelevant)	(irrelevant)

SENSE transfers data from the internal registers or memory of the controller.

READ transfers data from the external data medium.

CONTROL transfers data to the internal registers or memory of the controller.

WRITE transfers data to the external data medium.

WAIT permits the controller to generate an interrupt on certain events, such as power low or intervention. The controller enters a semi-idle state, in which it can accept a new START CHANNEL PROGRAM command (see Section 8.3.1).

For devices used for autoloading, CONTROL NO PARAM with modification 0 performs either an initializing function or no function at all.

STOP terminates the channel program.

Other fields

Other fields in the command word are not necessarily interpreted; if they are, they have the following meanings:

The D field indicates data chaining and is used to link the current command to the next command, so that a connected data transfer may take place to or from a non-connected memory area, indicated by a sequence of FIRST CHAR ADDRESS and CHAR COUNT parameters.

The S field means "skip data transfer" and is used only in conjunction with data chaining to transfer portions of connected data.

The meaning of the modifier field is device dependent, but modification 0 always indicates normal use of the device.

Standard status information 8.3.4

Standard status information is transferred to the primary memory starting from the status address, contained in the second word of the device description, either on normal termination of the channel program by the STOP command or on abnormal termination by a device error.

The standard status information includes the following:

1st word: Channel program address. Indicates the command following the current command.

2nd word: Remaining character count. Refers to the latest read or write command or chain of such commands; in the latter case, the count will be the total count for the chain.

3rd word: Current status. Reflects the status of the device at the termination of the program.

4th word: Event status. Contains information about events that have occurred since the last sensing of the event status register.

Peripheral processors 8.4

All peripheral devices, with the exception of the primary memory, are connected to the bus by means of peripheral processors (see Section 8.3).

Disc processor 8.4.1

A disc storage system consists of up to four disc drives connected through a single adapter to a disc processor. The drives are programmed as separate devices, and their channel programs can be run simultaneous-

ly. Up to four disc processors can be connected to the bus.

For further information, the reader should consult the disc processor reference manual (RCSL 30-M3).

General device processor 8.4.2

The general device processor, up to eight of which can be connected to the bus, includes the processor proper, a parallel channel interface, and a programmable front end.

The processor contains two independent sub-devices, called the transmitter and the receiver. These are programmed as separate devices, and can run their own channel programs simultaneously, though transmission in both directions cannot take place at the same time.

The interface consists of a set of symmetrical input and output lines, with a maximum transfer rate of 600 000 characters a second. An asynchronous, fully interlocked request/acknowledge communication technique is used.

The front end is the RC 3600, a computer system with an unusually wide range of peripheral and communications equipment. As a general device controller for the RC 8000 computer, the RC 3600 handles magnetic tape units, printers, paper tape and punched card equipment, and interactive terminals.

The RC 3600 is also a basic component in the RCNET computer network, where it serves as a network node, local device controller for the RC 8000, communications front end for the RC 8000 and computers of other manufacture, remote device controller for visual display unit terminals, and, finally, as a device controller performing the functions of a terminal controller, cluster controller, remote job entry terminal, or data entry system.

For further information, the reader should consult the reference manual for the general device processor (RCSL 30-M5).

Operator control panel 9.1

Operator control of the RC 8000 is exercised from a panel containing a power key, three indicator lamps, and an autoload button.

The power key is used to switch power to and remove power from the central unit, primary memory, and device controllers.

The indicators are marked POWER OK, RUN, and AUTOLOAD.

When POWER OK is lit, the power to the above-mentioned units is as it should be.

When RUN is lit, the RC 8000 is running.

AUTOLOAD lights when the autoload function is initiated, and remains lit until the autoloading is completed.

The autoload button, when pressed, stops the RC 8000; its release causes a reset, and the autoload function is initiated.

Operation 9.2

Power is removed from the RC 8000 by turning the power key to the off position. This generates a power interrupt, which permits the saving of restart information before the power disappears.

Power is switched to the RC 8000 by turning the power key to the on position. This causes a reset, and the power restart function is initiated.

If a power restart is undesirable, e.g. in a new installation or a system change, the autoload button is held depressed while the power key is turned to the on position; this places the RC 8000 in the stopped state, and when the autoload button is released, the autoload function will be initiated without power restart.

The autoload function is initiated by pressing the autoload button and releasing it.

Autoload 9.3

The autoload function places a device description, channel program, and autoload program loop in the primary memory and activates the loop. This causes a block to be loaded into the memory in a permanent area from a device with a permanent device number. The loading overwrites the autoload loop and must therefore, in the latter's place, consist of jump instructions to the jump address in the loaded block. The permanent device number is assigned to the desired autoload device when the system is configured, and the autoload channel program is designed so that any block-oriented input device can be used.

The AUTOLOAD indicator is extinguished by the loaded program after a self-check.

Options 9.4

The RC 8000 can be supplied with a special autoload device and/or a more sophisticated control panel as options.

Power restart 9.5

A power failure will cause the generation of a power interrupt so that restart information can be saved. The return of power will cause a reset, and the power restart function will be initiated.

Power restart may be regarded as an interrupt without the saving of information, since it is assumed that the necessary information was saved on the power interrupt.

A more detailed description of the power restart function will depend on the system in question.

NOTE

This chapter will be expanded at a later date.

Mnemonic Code		Numeric Code	Descr. Bit Pattern[1] (Status Reg. Bits) (6 7 8 9 10 11)
	Address handling		
AM	Next Address, Modify	9	0 0 0 0 0 1
AL	Address, Load	11	0 0 0 0 0 1
AC	Address Complemented, Load	33	0 0 0 0 0 1
	Register transfer		
HL	Half Register, Load	3	0 0 0 0 1 0
HS	Half Register, Store	26	0 0 0 1 0 0
RL	Register, Load	20	0 0 0 0 1 0
RS	Register, Store	23	0 0 0 1 0 0
RX	Register and Memory Word, Exchange	25	0 0 0 1 1 0
DL	Double Register, Load	54	0 0 1 0 1 0
DS	Double Register, Store	55	0 0 1 1 0 0
XL	Exception Register, Load	16	0 0 0 0 1 0
XS	Exception Register, Store	27	0 0 0 1 0 0
	Integer byte (halfword) arithmetic		
BZ	Integer Byte, Load (Zero Extension)	19	0 0 0 0 1 0
BL	Integer Byte, Load (Sign Extension)	2	0 0 0 0 1 0
BA	Integer Byte, Add	18	0 0 0 0 1 0
BS	Integer Byte, Subtract	17	0 0 0 0 1 0

[1]

Bit	Instruction Characteristic
6	privileged
7	modifies instruction counter
8	involves more than one memory word
9	modifies memory word
10	refers to memory word
11	auxiliary function

	Integer word arithmetic							
WA	Integer Word, Add	7	0	0	0	0	1	0
WS	Integer Word, Subtract	8	0	0	0	0	1	0
WM	Integer Word, Multiply	10	0	0	0	0	1	0
WD	Integer Word, Divide	24	0	0	0	0	1	0
	Integer double word arithmetic							
AA	Integer Double Word, Add	56	0	0	1	0	1	0
SS	Integer Double Word, Subtract	57	0	0	1	0	1	0
	Arithmetic conversion							
CI	Convert Integer to Floating	32	0	0	0	0	0	1
CF	Convert Floating to Integer	53	0	0	0	0	0	1
	Floating-point arithmetic							
FA	Floating, Add	48	0	0	1	0	1	0
FS	Floating, Subtract	49	0	0	1	0	1	0
FM	Floating, Multiply	50	0	0	1	0	1	0
FD	Floating, Divide	52	0	0	1	0	1	0
	Logical operations							
LA	Logical And	4	0	0	0	0	1	0
LO	Logical Or	5	0	0	0	0	1	0
LX	Logical Exclusive Or	6	0	0	0	0	1	0
	Shift operations							
AS	Arithmetically Shift Single	36	0	0	0	0	0	1
AD	Arithmetically Shift Double	37	0	0	0	0	0	1
LS	Logically Shift Single	38	0	0	0	0	0	1
LD	Logically Shift Double	39	0	0	0	0	0	1
NS	Normalize Single	34	0	0	0	1	0	0
ND	Normalize Double	35	0	0	0	1	0	0

Sequencing

	JL	Jump with Register Link	13	0	1	0	0	0	0
	SH	Skip if Register High	40	0	1	0	0	0	1
	SL	Skip if Register Low	41	0	1	0	0	0	1
	SE	Skip if Register Equal	42	0	1	0	0	0	1
	SN	Skip if Register Not Equal	43	0	1	0	0	0	1
	SO	Skip if Register Bits One	44	0	1	0	0	0	1
	SZ	Skip if Register Bits Zero	45	0	1	0	0	0	1
	SX	Skip if No Exceptions	46	0	1	0	0	0	1
	SP	Skip if No Write Protection	21	0	1	0	0	0	1
KL	RE	Return from Escape	22	0	1	1	0	1	0

Monitor control

	JE	Jump with Interrupt Enabled	15	1	1	0	0	0	0
	JD	Jump with Interrupt Disabled	14	1	1	0	0	0	0
IC	GP	General Register, Put	47	1	0	0	0	0	1
	IS	(unassigned)	31	1	0	0	0	0	0
ML	RI	Return from Interrupt	12	1	1	1	0	1	0
MS		(unassigned)	30	1	0	0	0	0	0
PL	GG	General Register, Get	28	1	0	0	0	0	1
PS		(unassigned)	29	1	0	0	0	0	0
KS		(unassigned)	51	1	0	0	0	0	0
IO	DO	Data Out	1	1	0	0	0	0	1
AEW	DI	Data In	0	1	0	0	0	0	1
		(unassigned)	58	1	0	0	0	0	0
		(unassigned)	59	1	0	0	0	0	0
		(unassigned)	60	1	0	0	0	0	0
		(unassigned)	61	1	0	0	0	0	0
		(unassigned)	62	1	0	0	0	0	0
		(unassigned)	63	1	0	0	0	0	0

In course of preparation.

NOTE

As many of the instructions found in the RC 4000 computer recur in the RC 8000 (cf. Chapter 10), the chapter "Defintion of Instructions" from the RC 4000 Computer Reference Manual is included here for your information.

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.

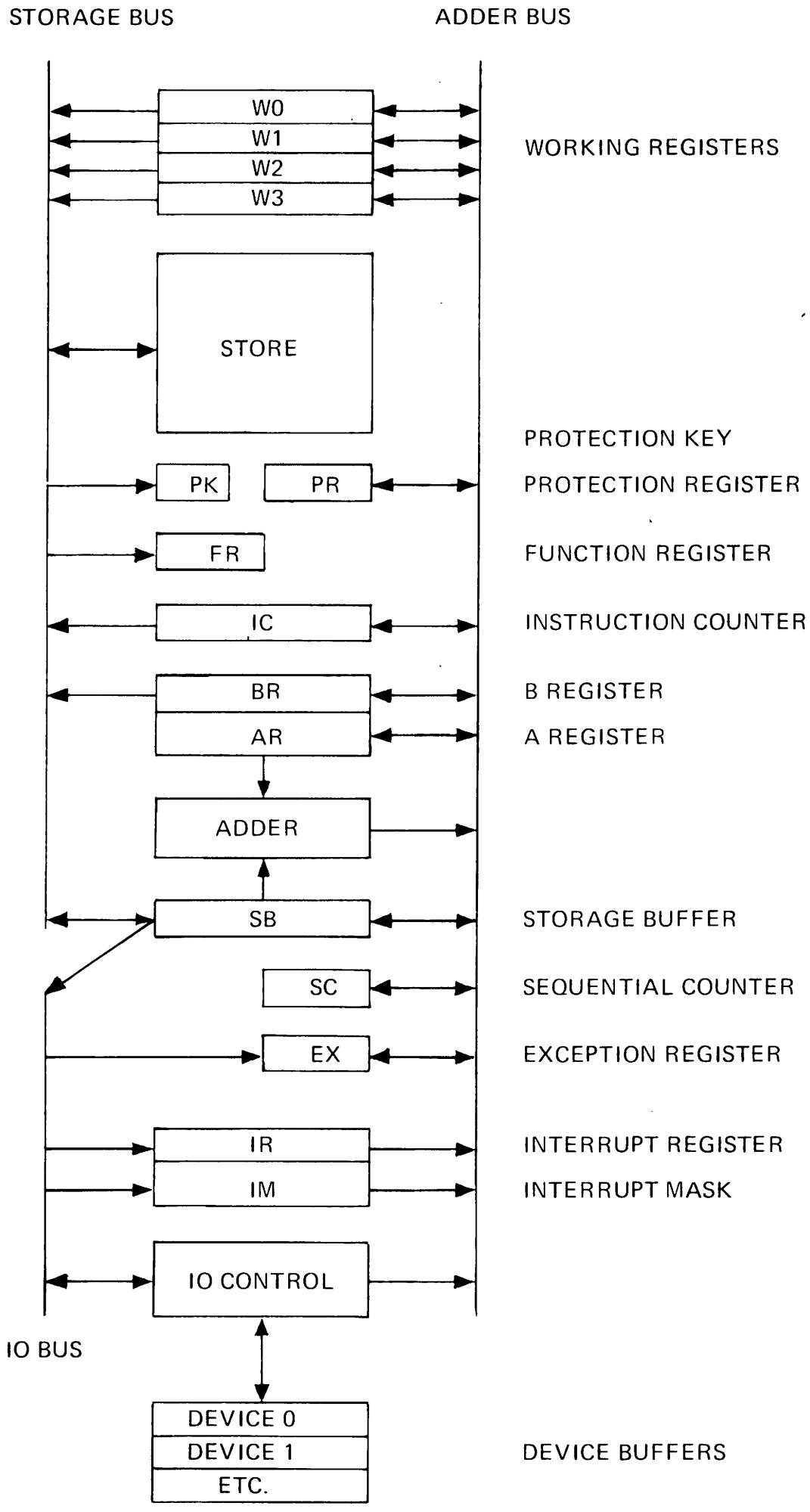
14.2. Register Structure

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.


STORAGE BUS
ADDER BUS
W0
W1
W2
W3
WORKING REGISTERS
STORE
PROTECTION KEY
PK
PR
PROTECTION REGISTER
FR
FUNCTION REGISTER
IC
INSTRUCTION COUNTER
BR
B REGISTER
AR
A REGISTER
ADDER
SB
STORAGE BUFFER
SC
SEQUENTIAL COUNTER
EX
EXCEPTION REGISTER
IR
INTERRUPT REGISTER
IM
INTERRUPT MASK
IO CONTROL
IO BUS
DEVICE 0
DEVICE 1
ETC.
DEVICE BUFFERS

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

Reserved memory locations A1

In course of preparation.

Instruction execution times

A2

In course of preparation.