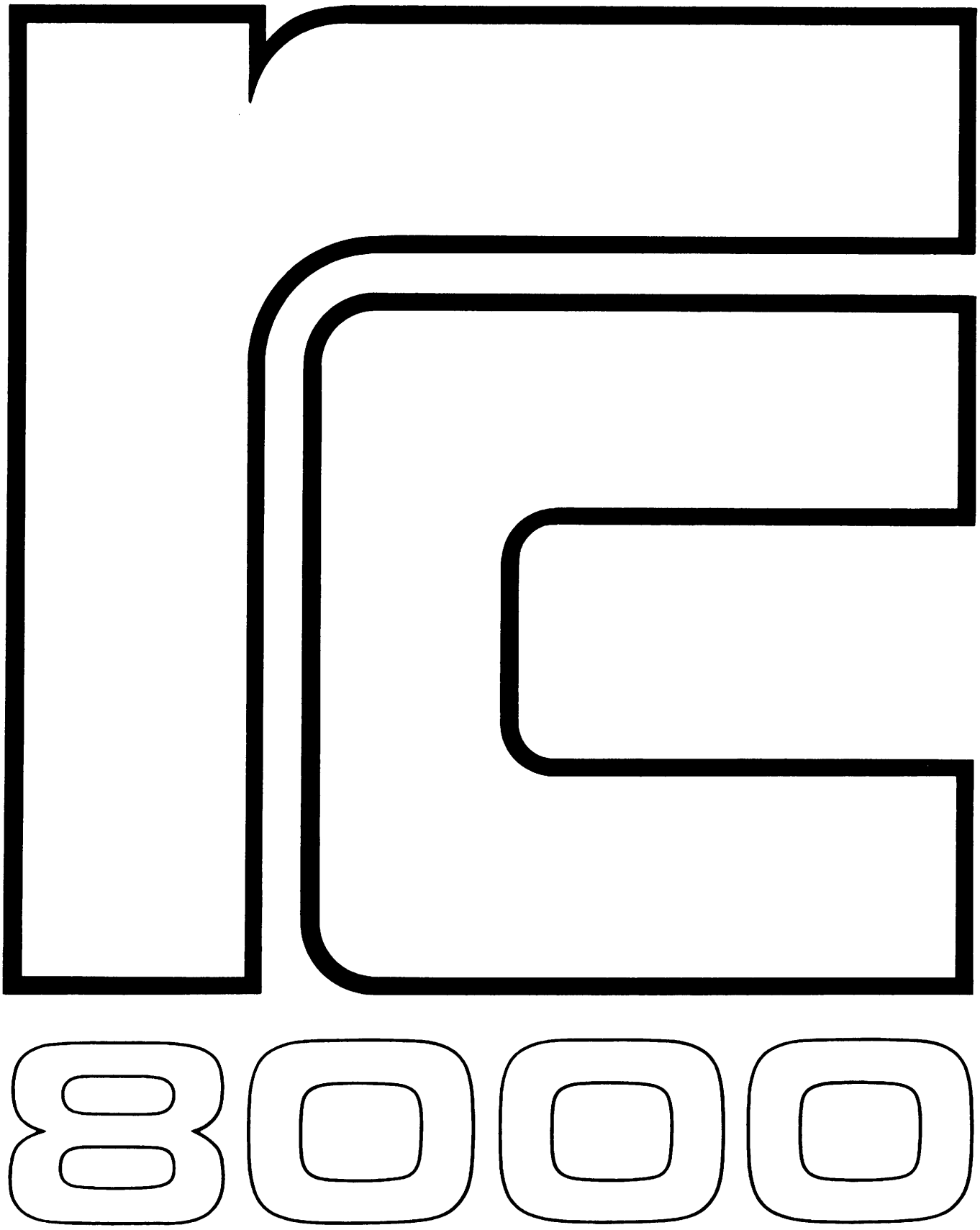
Computer Family Reference Manual



RC 8000 Computer Family
Reference Manual

A/S REGNECENTRALEN
Information Department

First Edition
June 1979
RCSL 42-i 1235

AUTHOR: Einar Mossin

KEYWORDS: RC 8000, Computer, Reference Manual

ABSTRACT: This manual provides basic programming information for programmers and users of the RC 8000 Computer Family.

Printed by A/S Regnecentralen, Copenhagen

Table of Contents

Preface

The RC 8000 Computer Family is a family of medium scale general purpose computers designed and manufactured by A/S Regnecentralen. The models in the family share a common architecture with an asynchronous unified bus giving full compatibility with respect to peripheral devices and allowing multi CPU configuration. The common instruction set includes multiply/divide and floating point arithmetic and support for mutually protected multiprogramming. The models differ in CPU implementation and in physical specification resulting in different sets of instruction execution times and different configuration possibilities.

This manual provides the common basic programming information for programmers and users of the RC 8000 Computers:

- Chapter 1 lists the common specifications.
- Chapter 2 contains a survey of design features and gives a preliminary introduction to concepts used in chapters 3-9 in an attempt to reduce the need for forward references during the first sequential reading of these.
- Chapters 3-5 describe data formats, instruction format, memory addressing and arithmetic.
- Chapter 6 describes the multiprogrammming facilities.
- Chapter 7 describes the facilities for program error diagnostis and debugging.
- Chapter 8 describes the input/output control.
- Chapter 9 describes the power restart and autoload facilities.
- Chapter 10 completes the picture with a formal algorithmic description of the information in the preceeding chapters.

Although the manual contains hints and examples of efficient use of the instruction set, no attempt is made to teach programming techniques.

Instruction execution times and other model specific information are given in the General Information Manual for each model.

The function of peripheral devices is described in separate manuals.

1 RC 8000 Specifications

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
Direct addressing of up to 4 194 304 words. Each word contains 24 data bits. Parity check or error recovery are supplied in all models. Memory type and configuration depends on model.

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 halfwords 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.

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 halfword operations and word comparision.

Logical operations permit setting and testing af single bits.
Escape facility permits programmed actions on any or all instructions.
Exception concept permits programmed diagnostics on programming errors.

Protection System

Privileged instructions and memory protection associated with a monitor mode ensure complete monitor control.
Monitor mode is entered through monitor calls and interrupts.
Violation of protection system leads to exceptions.

Interruption System

Program interruption system with priority levels.
Assignment of levels and disabling of interrupts under program control.
Power failure interrupt and power restart are standard.

Input/output System

All peripherals are connected to the unified bus.
Blockoriented controllers are standardized and perform all input/output functions under the control of channel programs in primary memory.
Data is transferred directly between the controllers and the primary memory. An asynchronous, fully interlocked request/acknowledge communication technique is employed.
CPU's have no special status on the bus.
Interrupts are achieved by addressing a CPU.

2 Design Features

2.1 Operand Length

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.

Halfword Handling
Direct addressing of 12-bit halfwords aids efficient packing of data.

2.2 Registers and Addressing

2.2.1 Registers Structure
The 8 dynamic registers define the current state of a program. They are dumped when control is transferred to the monitor program or to the exception or escape service routines of the program and are reloaded when the program is resumed. The dynamic registers comprise:

The four working registers, "w0", "w1", "w2" and "w3", see 3.2. Each instruction specifies one of these, see 3.3. The working registers except for "w0" can function as index registers.

The status register, "status", contains the following information. Bits are numbered 0-23 with 0 indicating the most significant bit.

Bit no.	Contents
0	monitor mode: privileged instructions allowed, see 6.
1	escape mode: escape facility active, see ch.7.
2	after address modify: address calculation in next instruction starts from present contents of the address register, see 3.3.2.
3	after escape: address calculation in next instruction is skipped. The contents of the address register is used as resulting address. If the "after address modify" bit is set too, the contents of the word pointed out by the address register is used, see ch. 7.
4	integer exception active, see ch. 4.

5	floating point exception active, see ch. 5
6-11	escape mask, see ch. 7.
12-15	dump error count: used in connection with memory errors in registerdump at interrupt or monitor calls, see 10.7.4.
16-19	not used.
20	disable: disable level of interrupt limit register active, see 6.6.
21-23	exception: contains status information after arithmetic and input/output operations. Accessible as the exception register, ex, through special instructions, see ch. 4, 5 and 8.

The instruction counter, "ic", is a 24-bit register. Bit 23 is always zero. Bit 0 is ignored in instruction fetch, but the full register is used as a 24 two's complement integer in relative address calculation.

The cause register, "cause", indicates the cause of deviations from normal program flow, i.e. interruptnumber, monitor call number, exception cause and escape cause, see ch. 6 and 7.

The address register, "addr", is used internally as a working register for address calculations. Dumped values of addr are used in return from interrupt and escape, and are very useful in connection with the exception and escape facilities.

The process definition registers define the relocation, memory protection and interrupt disabling for the currently active program. The contents of the registers is changed under monitor control by interrupts and monitor calls and at return from these, see ch. 6.

The process definition registers comprise:

register	explanation
cpa	the common protected area limit register defines the upper limit of the nonrelocated, read-only lower part of memory, see 3.5.
base	the base register defines the displacement of physical addresses compared to logical addresses, see 3.5.

lowlim — the lower limit register defines the physical lower limit of the write accessible part of memory, see 3.5.

uplim — the upper limit register defines the physical upper limit of the write accessible part of memory, se 3.5.

intlim — the interrupt limit register defines two levels of interrupt disabling. The program shifts between the levels, which may be equal and describe no disabling, by execution of the special jump instructions: "jump disable" and "jump enable", that respectively sets and clears the disable bit in "status", see 6.6.

The monitor registers (ch. 6) are used in connection with interrupts and monitorcalls.

register	explanation.
inf	the system information register points to a system table of 6 words. The table contains register dump and entry addresses for interrupt, monitor call, exception and escape. At interrupt and monitor call inf is decreased by 12. This allows specification of special actions for the mentioned events during execution of the monitor program. At return from interrupt, inf is increased by 12 again.
size	defines upper limit of memory. Size is used for initialization of the upper limit register when monitormode is entered at interrupt or monitor calls.
montop	defines the number of monitor calls. Monitor calls are provoked by execution of the "jump disable" instruction with a resulting address fullfilling $-2048 \leq addr < -montop$ The monitor call cause is given as addr + 2048 and is used for switching to the specified monitor function.

rtc The real time clock register is a 16 bit read only register. Rtc is incremented by 1 every 0.1 milliseconds and counts modulo 65536. A timer interrupt is generated every timer interrupt period, 25.6 milliseconds in most models.

2.2.2 Program Relocation

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 process definition registers define the current relation between the logical address as seen by the program and the actual physical address, thereby ensuring that programs can use saved address information after restarting in new memory areas.

2.2.3 Address Calculation

Besides relative addressing RC 8000 supplies indexing and indirect addressing. The addressing modes can be used in any combination. The possibilities are enhanced by the fact that the 3 index registers are also working registers and through the address modify instruction. The address modify instruction makes indirect addressing in multiple levels, use of the contents of any memory-word as index, multiple indexing and conditional address calculation possible.

2.3 Escape and Exception

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. The exception concept allows specification of a diagnostic routine, which will be called when programming errors such as illegal instructions or addressing occurs. The concept may also be used for trapping of arithmetic overflow and underflow and for definition of breakpoints.

2.4 Monitor Control

In a multiprogramming system, where many concurrent tasks are performed, it is vital that erroneous programs can 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, program interruption and monitor calls.

2.4.1 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 and a read-protected part. The read-accessible part is addressed independently of dynamic program relocation. Any attempt to violate the protection system leads to a program exception.

2.4.2 Privileged Instructions

Further protection is achieved through privileged instructions that can only be executed within the monitor program. These instructions include input/output functions as well as control of the interruption system, memory protection and dynamic program relocation. Attempts to execute privileged instructions in normal mode leads to program exception.

2.4.3 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 a number of signals simultaneously. Any of these signals interrupts the current program immediately and starts the monitor program. Register dump and register initialization for fast switch to service routine are executed by firmware. Return to program and restoring of registers are carried out by a privileged instruction, return from interrupt. Critical actions can be protected against interruption by partial or total disabling.

2.4.4 Monitor Calls

To ensure the legal interaction between parallel programs and between programs and the input/output system, the monitor program must supply a set of service functions. Activation of these take place by execution of a monitor call instruction. The calling program is interrupted and the monitor program is started as for interrupts, but at a different entry address and with "cause" defined by the monitor call instruction.

2.5 Input/Output System

The input/output system is based on a unified bus, i.e. a common bus for communication between all devices and CPU's 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 processors are unaware of the types of devices attached to the bus. 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 the exception register, which can be tested by the program in order to take appropriate action. Device status will be transferred to the memory and the central processor will be interrupted when the channel program is terminated.

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.

2.5.1 Bus Control

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 processor 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 time-out, 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.

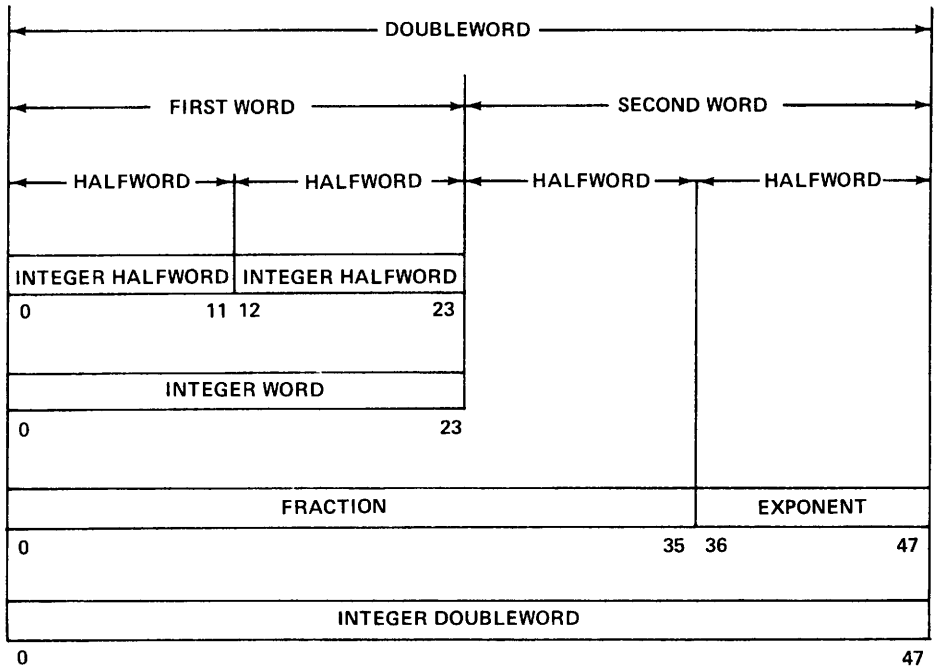
2.5.2 Bus Communication

To facilitate interfacing with other systems, an asyncronous 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.

3 Data Formats and Instructions

3.1 Data Formats

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. The 12-bit halfwords are directly addressable, and may be used as signed numbers. Double words are used to represent integers of 48 bits and floating-point numbers with 36-bit fractions and 12-bit exponents.

3.2 Working Registers

The register structure includes four 24-bit working registers, one of which is specified 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 half-word (or four words) of the primary memory, (see 3.5).

Two adjacent working registers can be used to hold a double-

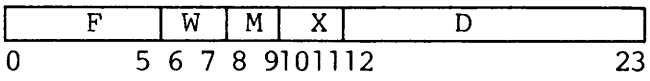
length operand of 48 bits. In double-length operations, the four registers are connected cyclically as follows:

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.

3.3 Instruction Format

The instruction format is divided into an operation halfword and an address halfword, 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 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 halfwords within the program.

A full address of 24 bits is formed by means of the displacement, D, in conjunction with the contents of an index register, X, and the content of the instruction counter, "ic". The generation of the effective address, "addr", is controlled by the address mode field, M, as follows:

```
M = 00    addr = X + D
M = 01    addr = word (X + D)
M = 10    addr = X + IC + D
M = 11    addr = 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.

3.3.1 Notation of Examples

The examples in the remaining part of this manual is given in a primitive pseudo assembly language. Basic instruction format is

```
<mn>   <w>   <d> ; <comments>
```

<mn> is the instruction mnemonic. Instructions and their mnemonics are introduced before they are used in examples. Survey of instruction by mnemonic is given in Appendix 1, which also gives the reference to the detailed instruction descriptions in chapter 10.

<w> specifying the working register as w0, w1, w2 or w3. The <w> may be left out, specifying W-field = 0, but this is only done, when the instruction does not use a working register.

<d> specifies the displacement either as an integer or as the name of a label in which case the displacement is the address of the labelpoint. Labelpoints are given as '<names>' in front of an instruction, a constant or the name of a label.

<comment> may be any text, comments are terminated by new line.

The principles of the instruction mnemonics are:

- Each instruction has a full name describing the instruction as well as possible. Word choice and structure is uniform for the different instructions.

- A short name is defined by the first word and one other significant word of the full name, indicated by underlining. The short name is the normal reference to the instruction.

- The acronym derived from the short name defines the mnemonic.

To show the notation the "register:load" instruction is introduced preliminarily. The "register:load" instruction, "rl", loads the specified working register with the contents of the memory word pointed out by the effective address.

```
constant:  2
addr:      constant
           .
           .
           rl  w1      constant   ; w1 = 2
           rl  w3      addr       ; w3 = address (constant)
```

Constant and addr must represent addresses that can be contained in the D-field, i.e. less than 2048.

Relative addressing is indicated by a point after the instruction mnemonic, while a point after the labelname in an instruction means that the D-field of the instruction is the displacement between the labelpoint and the instruction. Thus, with same data definitions as above:

```
           rl. w1      constant.  ; w1 = 2
           rl. w3      addr.      ; w3 = address (constant)
```

Now the displacement between the data and the instructions must be less than 2048 and greater than or equal to -2048.

Indexing is indicated by placing an index register specification in front of the displacement. Thus, still keeping the data definitions:

```
rl. w3     addr.      ; w3 = address (constant)
rl  w1   x3+0         ; w1 = 2
rl  w2   x3+2         ; w2 = address (constant)
rl. w0   x1+constant. ; w0 = address (constant)
```

Indirect addressing is indicated by placing the address part of the instruction in parenteses. Thus, again with same data definitions:

```
rl. w1    (addr.)     ; w1 = 2
rl. w3 (x1+constant.); w3 = 2
```

3.3.2 Address modification in next instruction

"Address : modify that of next instruction", "am", sets the "after "am" bit in the status register. This will modify the address calculation in the next instruction as follows depending on the M-field:

```
M = 00    addr = X + D + addr
M = 01    addr = word (X + D + addr)
M = 10    addr = X + ic + D + addr
M = 11    addr = word (X + ic + D + addr)
```

I.e. the effective address of the "am" instruction is added to the displacement before it is used in the address calculation. The "after am" bit is cleared, when it has had its effect.

The "am" instruction expands the addressing possibilities as indicated in the following examples, the corresponding datastructures are not shown.

Indirect addressing in two levels:

```
am.       (addr.)     ; addr points to the address of a
                      ; value
rl  w1    (0)         ; w1 = value
```

Indexing by a memory word:

```
am.       (addr.)     ; contents of addr points to first
                      ; word in table
rl  w1     4          ; w1 = third word in table
```

Double indexing:

```
rl. w2      relrecord.; relrecord contains relative
                      ; position of record in block
am.        (relfield.); relfield contains relative
                      ; position of field in record
rl. w0      x2+block. ; block points at blockstart
                      ; w0 = fieldvalue
```

Conditional address calculation is possible through conditioning of the execution of the "am" instruction (see 3.4.2).

3.4 Use of the effective Address as an Operand

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 contents 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.

3.4.1 Load of immediate Values

The "address:load into register" instruction, "al", assigns the effective address to the specified register.

```
c1:       4711
          :
          al  w0     10        ; w0 = 10
          al. w1     c1.       ; w1 = address (4711)
          al  w2  x1-2         ; w2 = w1 - 2
          al  w1  x1+10        ; w1 = w1 + 10
```

The "address complemented:load into register" instruction, "ac", assigns the two's complement of the effective address to the specified register.

```
ac  w1   x1+1        ; w1 = -w1 -1, the ones complement
ac  w0     10        ; w0 = -10
ac. w2    (c1.)      ; w2 = -4711
ac  w3   x1-10       ; w3 = 10 - w1
```

Complementation of the maximum negative number will produce an overflow (see 4.4).

3.4.2 Skip Instructions

"Skip if register high", "sh", compares the value of the specified working register and the effective address interpreted as integers in two's complement representation. If the register value is greater than the effective address the following instruction is skipped.

```
sh  w1     -1        ; if w1 ≤ -1 then
al  w1      0        ; w1 = 0
```

This piece of code ensures that the following instructions are never entered with a negative value of w1. Note that the comment illustrates another way of looking at the skip instruction. The skip instruction and the following instruction is considered one conditional instruction conditioned by the negated condition.

"Skip if register low", "sl", skips the following instruction if the register value is less than the effective address.

```
sl  w1      0        ; if w1 ≥ 0
am         -2        ; then w0 = 43  (iso character
                     ; value of "+")
al  w0     45        ; else w0 = 45  (iso character
                     ; value of "-")
sh  w1     -1        ; if w1 ≤ -1  then
ac  w1   x1          ; w1 = -w1
```

The character value of the sign of the number in w1 is assigned to w0 (an example of conditional address calculation). After that w1 is assigned the absolute value of the number.

"Skip if register equal", "se", skips the following instruction if the register value and the effective address is equal.

"Skip if register nonequal", "sn", skips the following instruction if the register value and the effective address is not equal.

"Skip if selected register bits all ones", "so", uses the effective address as a masc to test selected bits in the working register. If all bits in the working register that correspond to ones in the effective address are one, the following instruction is skipped.

"Skip if selected register bits all zeros", "sz", is analog to "so". If all bits in the working register that corresponds to ones in the effective address are zero, the following instruction is skipped.

```
sz  w1    1        ; if w1 is odd then
al  w1  x1-1       ; make it even.
```

3.4.3 Shift Instructions

"Logical shift of single register", "ls", shifts the content of the specified working register the number of places given by the effective address. If the effective address is negative the shift is a right shift with zeroes shifted in at the most significant bits else it is a left shift with zeroes shifted in at the least significant bits.

"Logical shift of double register", "ld". As "ls" but the shift is performed on the 48 bit double register specified.

"Arithmetic shift of single register", "as". As "ls", but with sign extension for right shifts. Left shifts may produce an overflow (see further 4.4).

"Aritmetic shift of double register", "ad". As "as", but the shift is performed on the 48 bit double register.

3.5 Use of the effective Address to refer to Memory Locations

In instructions for register transfer, instructions for arithmetic and logical operations and some other instructions the effective address is used as a logical address pointing to a memory location.

Memory addresses are always expressed as halfword addresses. The halfword 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 working registers are addressable as the first four words of memory. The programmer can therefore perform 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.

HALFWORD ADDRESS		
0	24 BITS	WORKING REGISTER 0 (w0)
2	24 BITS	WORKING REGISTER 1 (w1)
4	24 BITS	WORKING REGISTER 2 (w2)
6	24 BITS	WORKING REGISTER 3 (w3)

The relation between logical addresses and physical addresses, i.e. true locations in the memory, is defined by the contents of the process definition registers base, cpa, lowlim and uplim. These contents define the memory protection given by the division of the logical address space in process area, common protected area, working registers and nonaccessible area.

Control of the contents of the process definition registersis described in chapter 6.

3.5.1 Process Area

The process area is defined by the value of the limit registers.

Inside the process area the effective address is increased by the contents of the base register before it is used as a physical address. The limit registers contain physical addresses, the process area is therefore defined by

$$lowlim \leq addr + base < uplim$$

The process area must be contained in true memory, this means that the condition

$$8 \leq lowlim \leq uplim < memory\ size$$

must be fulfilled at all times.

The logical process area should be kept constant throughout a program's lifetime to ensure consistent program behaviour. This means that the values "lowlim - base" and "uplim - base" should be kept constant.

The process area allows read- and write access. Doubleword reference with effective address equal to "lowlim - base" will lead to program exception.

3.5.2 Common Protected Area

Effective addresses outside the process area may refer to the common protected area defined by

$$8 \leq \text{addr} < \text{cpa}$$

Inside the common protected area, the effective address is used directly as a physical address and only read access is allowed. The contents of the cpa limit register must fulfill the condition

$$8 \leq \text{cpa} < \text{memory size}$$

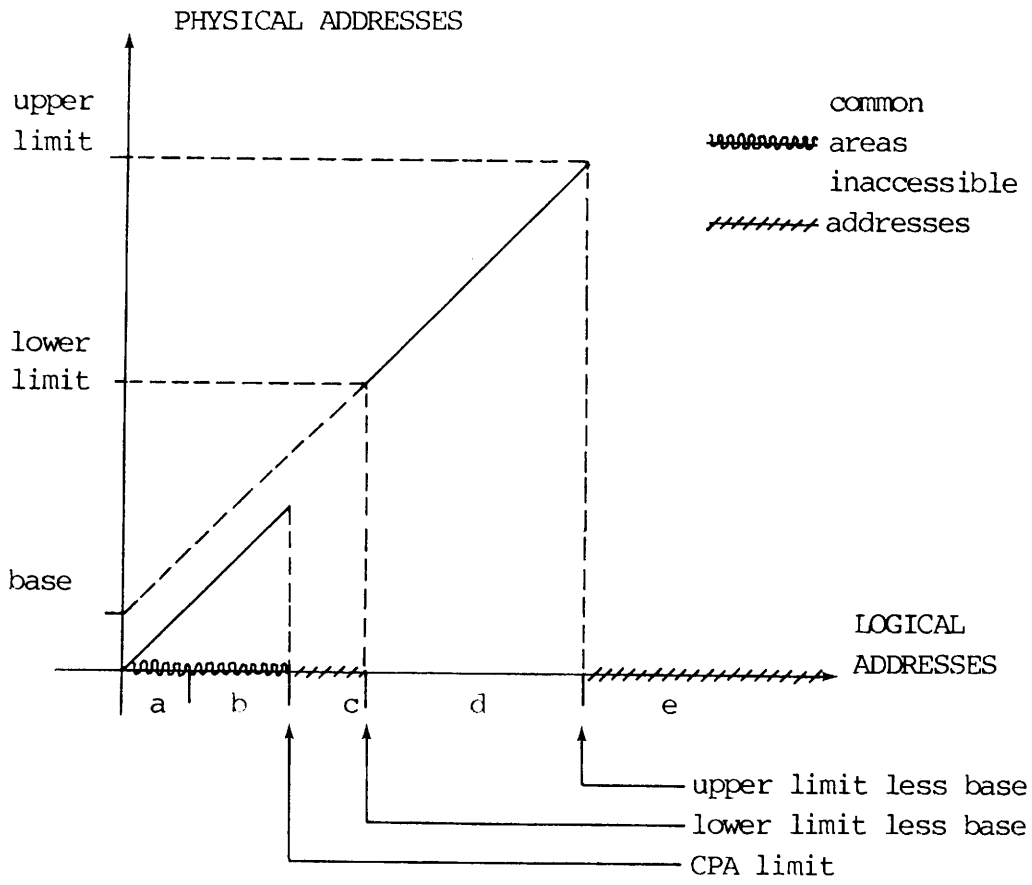
The common protected area may overlap the process area. Consistent program behaviour is only fully ensured in this case if base is kept constant. The overlap belongs to the process area. Double word reference with effective address equal to 8 and write access in the common protected area will lead to program exception.

3.5.3 Working Registers and Nonaccessible Area

As described before effective addresses 0 through 7 refers to the working registers. All other logical addresses outside the process area and the common protected area are considered nonaccessible and reference to them leads to program exception.

3.5.4 Memory Addressing. Example of User Program

The following figure shows the physical address as a function of the logical address for a user program:



The division of the logical address space in this example is as follows:

a. $0 \leq$ logical address < 8

The working registers. Full access.

b. $8 \leq$ logical address $<$ cpa

Common protected area used for references to system parameters. Read-access only. Physical address = logical address.

c. cpa $\leq$ logical addres $<$ lower limit - base
Non-accessible area.

d. lower limit - base $\leq$ logical address < upper limit - base
Process area.

Relocatable (physical address = logical address + base).
Full access.

e. upper limit - base $\leq$ logical address
Non-accessible area.

3.6 Jump Instructions

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 read access. Reference to the non-accessible area leads to program exception.

Subroutine jumps are supported 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 specified 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 jump, i.e. with W field equal to zero.

Conditional jumps are defined by use of the skip instructions.

The "jump with link in register" instruction, "jl", is the basic jump instruction.

The "jump and select disable limit", "jd", and "jump and select enable limit", "je", instructions work like "jl", but have an effect on the interrupt system. A subset of negative effective addresses will, used in connection with the "jd" instruction, define moni- tor calls. These features are described in chapter 6.

3.6.1 Jump, example 1

```
sub:    -                   ; subroutine code that does
        -                   ; not change the contents of w3
        -                   ;
        jl      x3+0        ; return jump

        -                   ;
        jl. w3    sub.      ; subroutine call
        -                   ; return point
```

The example sketches a simple subroutine structure. Parameters are given in the remaining registers.

3.6.2 Jump, example 2

```
sub:    -                   ; subroutine entry point
        -                   ; code
        rl  w1  x3+0        ; load of parameter 1
        -                   ; code
        -                   ;
        sh  w0    -1        ; if result < 0
        am        -2        ; then error return
        jl      x3+6        ; else normal return

        -                   ; code
        -                   ;
        jl. w3    sub.      ; subroutine call with
        47                  ; parameter 1 : number
        text                ; parameter 2 : address of text
        jl.       error.    ; error return point : goto error
        -                   ; normal return point
        -                   ; code
```

This example sketches a little more sophisticated subroutine structure. Parameters are given following the call and an error return is supplied.

3.6.3 Jump, example 3

```
table:    -                  ; table of records : key,
          -                  ;                    value
          -                  ;
                             ; records
          -                  ;
last:     -                  ; last record : key,
          -                  ;               value
          -                  ;
          al. w1    table.   ; initialize pointer to first
          -                  ; record
loop:     sn  w0 (x1+0)      ; if w0 = key (record) then
          jl.       found.   ; goto found
          al  w1  x1+4       ; increase pointer to next
                             ; record
          sh. w1    last.    ; if table not exhausted then
          jl.       loop.    ; goto loop
          -                  ; not found code
          -                  ;

found:    rl  w2  x1+2       ; load value
          -                  ; found code
          -                  ;
```

The example shows a simple loop. The contents of w0 is searched in a table of double word records containing a key and a corresponding value. If w0 matches one of the keys, control is transferred to the found code which loads the value and proceeds. If the table is exhausted without a match the not-found code after the loop is executed. The end of the loop shows a typical conditional jump.

3.7 Register Transfer Instructions

The "register:load" instruction,"rl", loads the specified working register with the contents of the memory word pointed out by the effective address. If the effective address is 0 through 7, "rl is a register to register transfer. Reference to the non-accessible area leads to program exception.

The "register:store" instruction, "rs", stores the contents of the specified working register in the memory word pointed out b the effective address. If the effective address is 0 through 7,

"rs" is a register to register transfer. Other reference outsid the process area leads to program exception.

The "register : exchange with word" instruction, "rx", exchanges the contents of the specified working register and the memory word pointed out by the effective address. If the effective address is 0 through 7, "rx" exchanges the contents of two registers. Other reference outside the process area leads to program exception.

The "zero-extended halfword : load into register" instruction, "zl", loads the least significant 12 bits of the specified working register with the content of the halfword pointed out by the effective address. The most significant 12 bits are cleared. If the effective address is 0 through 7 "zl" is a half register to register transfer. Reference to the nonaccessible area leads to program exception.

The "extended halfword : load into register" instruction, "el". As "zl" but the most significant 12 bits will contain a sign extension of the loaded halfword.

The "half register : load" instruction, "hl". As "zl" but the most significant 12 bits are left unchanged.

The "half register : store" instruction, "hs", stores the least significant 12 bits of the specified working register in the halfword pointed out by the effective address. If the effective address is 0 through 7, "hs" is a register to half register transfer. Other references outside the process area leads to program exception.

The "double register : load" instruction, "dl", loads the specified double register with the contents of the doubleword pointed out by the effective adress. If the effective address is 0 through 7, "dl" is a double register to double register transfer; in this case it is important to note that the least significant register is assigned before the most significant register is transfered. Reference to the nonaccessible area and to the lower limits of the process area (addr = lowlim - base) and the common protected area (addr = 8) will lead to program exception.

The "double register : store" instruction, "ds", stores the contents of the specified double register in the doubleword pointed out by the effective address. If the effective address is 0

through 7, "ds" is a double register to double register transfer (note as for "dl"). Other references outside the process area and reference to the lower limit of the process area will lead to program exception.

3.8 Logical Operations

The instructions for logical operations combine the contents of the working register with the operand, i.e., the contents of the word pointed out by the effective address, by a logical bit by bit operation. If the effective address is 0 through 7, the result is a combination of two registers. Reference to the non-accessible area leads to program exception.

The "logical and : combine word with register" instruction, "la", executes the logical and- operation defined by

register bit	operand bit	result bit
1	1	1
1	0	0
0	1	0
0	0	0

The "logical or : combine word with register" instruction, "lo", executes the logical or-operation defined by

register bit	operand bit	result bit
1	1	1
1	0	1
0	1	1
0	0	0

The "logical exclusive or : combine word with register" instruction, "lx", executes the logecal exclusive-or-operaton defined by

register bit	operand bit	result bit
1	1	0
1	0	1
0	1	1
0	0	0

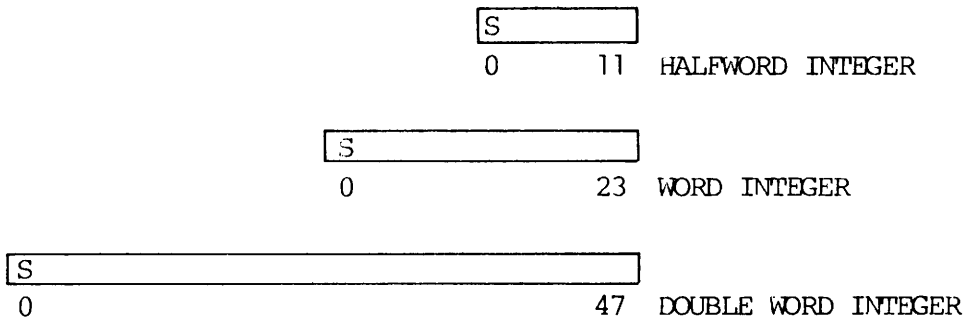
3.9 Skip if no protection

The "skip if word not protected" instruction, "sp", skips the following instruction if the effective address of the instruction points to the process area or to a register, i.e. if storing in the word addressed is allowed.

4 Integer Arithmetic

4.1 Number Representation

The standard arithmetic operands are signed integers of 12, 24 and 48 bits:



Positive numbers are represented in true binary form with a zero in the sign bit. Negative numbers are represented in 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.

4.2 Halfword Arithmetic

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

$$-2^{**}11 = -2048 \leq \text{integer halfword} \leq 2047 = 2^{**}11-1$$

The instruction "extended halfword : load into register", "el" (see 3.7) serves to extend a signed 12-bit halfword towards the left to 24-bits, as it is placed in a working register.

The arithmetic instructions "extended halfword : add to register", "ea", and "extended halfword : subtract from register", "es", adds or subtracts the contents of the halfword pointed out by the effective address to or from the specified working register after sign extension. If the effective address is 0 through 7 the operation is a register- halfregister operation. Reference to non-accessible area leads to program exception. Overflow and carry is registered in the exception register and integer exception

may occur as described in 4.6.

The "half register : store" instruction, "hs", (see 3.7) is used for storing halfword results in memory.

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

4.3 Word Arithmetic

A signed integer represented by a 24 bit word or register is confined to the following range

$$-2^{**}23 = -8388608 \leq \text{integer word} \leq 8388607 = 2^{**}23-1$$

The "word : add to register" instruction, "wa", and the "word : subtract from register" instruction, "ws", adds or subtracts the contents of the word pointed out by the effective address to or from the specified working register. If the effective address is 0 through 7 the operation is a register-register operation. Reference to non-accessible area leads to program exception. Overflow and carry is registered in the exception register and integer exception may occur as described in 4.6.

4.4 Doubleword Arithmetic

A signed integer represented by a 48 bit doubleword or double-register is confined to the following range

$$-2^{**}47 \leq \text{integer double word} \leq 2^{**}47-1$$

Note that bit 24 of the integer i.e. bit 0 of the least significant word or register is not a sign bit, but contains a normal significant digit.

The "add double word to double register" instruction, "aa", and the "subtract double word from double register" instruction, "ss", adds or subtracts the contents of the doubleword pointed out by the effective address to or from the specified doubleregister. If the effective address is 0 through 7, the operation is a doubleregister-doubleregister operation. In this case it is important to remember that the operation on the least significant registers is completed before the operation on the most signifi-

cant registers and carry is executed. Reference to non-accessible area and reference to the lower limits of process area (addr = lowlim - base) and common protected area (addr = 8) leads to program exception. Overflow and carry is registered in the exception register and integer exception may occur as described in 4.6.

4.5 Multiplication and Division

Integer multiplication of the contents of a working register with the contents 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.

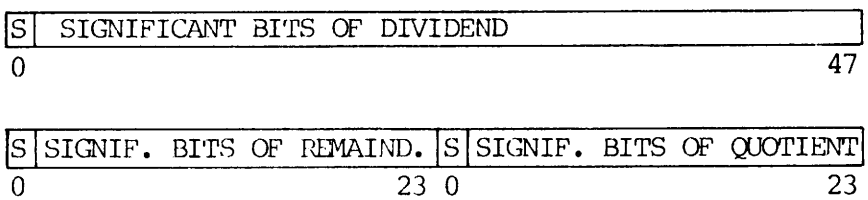
The only exception to this occurs in the multiplication of two maximum negative numbers:

```
(-2**23)*(-2**23) = 2**46
```

This result will be represented as bit 1 = 1, all other bits = 0.

The contents of a double register can be divided by the contents 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 righthand 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)



The "word : multiply by register giving double register" instruction, "wm", multiplies the specified working register by the contents of the word pointed out by the effective address. The result is placed in the corresponding double register. If the ef-

fective address is 0 through 7, the operation is a register-register operation. Reference to nonaccessible area leads to program exception. Exception register is unchanged and integer exception can not occur.

The "word : divide into double register" instruction, "wd", divides the specified double register by the contents of the word pointed out by the effective address. The quotient is placed in the least significant register while the remainder is placed in the most significant register. If the effective address is 0 through 7, the operation is a double register-register operation. Reference of nonaccesible area leads to program exception. Carry = 0 and overflow is registered in the exception register and integer exception may occur as described in 4.6. At overflow the dividend is left unchanged.

If the dividend is represented by an integer word, the sign must be extended before division

```
rl. w1     dividend. ;
al  w0     0         ; sign extension of positive
                     ; dividend
wd. w1     divisor.  ;

rl. w0     dividend. ;
ad  w1    -24        ; sign extension of any dividend
wd. w1     divisor.  ;

rl. w1     dividend. ;
sl  w1     0         ; sign extension of any dividend
am         1         ;
al  w0    -1         ; faster in most models
wd. w1     divisor.  ;
```

4.6 Overflow and Carry Indication

Arithmetic operations except integer multiplication indicate a normal or an exceptional result by setting the right-most two bits of the 3-bit exception register. Physically, the exception register is the last three bits in the status register, but it is treated by special instructions as a logically independent register. See 4.7.

After a normal result, exception bit 22 is set to zero. An integer

overflow will set exception bit 22 to one, and will provoke an integer exception if the "integer exception active" bit in the status register is set, see further chapter 7.

An overflow condition is recognized in the following situations:

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

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

Or the result of doubleword addition or subtraction exceeds the range of a 48 bit signed integer, viz.

$-2^{**}47 \leq$ integer doubleword $\leq 2^{**}47-1$

Note that multiplication can never produce overflow and leaves the exception register unchanged.

(2) The instruction "address complemented : load into register", "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 register. 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 bit 0. For addition this means, that a carry is produced when the sum of the operands interpreted as unsigned numbers of 24 bits (or 48 bits) exceeds $2^{**}24-1$ (or $2^{**}48-1$). For subtraction the carry is to be interpreted as not borrow, i.e. a carry is produced when the register operand interpreted as an unsigned number is greater than or equal to the "memory" operand interpreted as an unsigned number.

The carry indication in the exception register can be used for multiple length arithmetic and in connection with comparison of unsigned numbers, see examples in 4.7.

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

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

4.7 Exception Register Instructions

The "skip if selected exception bits all zero" instruction, "sx", uses the 3 least significant bits of the effective address to test selected bits in the exception register. If all selected bits, i.e. the bits corresponding to 1's in the effective address, are zero the following instruction is skipped.

The "sx" instruction may be used to test for overflow and carry.

```
-                       ; integer operation
sx          2           ; if overflow then
jl.         overflow.   ; goto overflow
-

-                       ; integer operation
sx          1           ; if carry then
jl. w3      carry.      ; call carry action
-                       ;
```

The "exception register : store in halfword" instruction, "xs", stores the exception register in the 3 least significant bits of the halfword pointed out by the effective address. The most significant 9 bits are set to zero. If the effective address is 0 through 7, "xs" is an exception register to halfregister transfer. Other references outside the process area lead to program exception.

The "exception register : load from halfword" instruction, "xl", loads the exception register with the 3 least significant bits of the halfword pointed out by the effective address. If the effective address is 0 through 7, "xl" is a halfregister to exception register transfer. Reference to non-accessible area leads to program exception.

Example 1:

```
;comparison of unsigned integers

          rl. w1      int1.      ;
          ws. w1      int2.      ;
          sx          2.01       ; if int1 >= int2 then
          jl.         greq.      ; goto greq;

          rl. w1      int1.      ;
          ws. w1      int2.      ;
          sx          2.01       ; if int1 < int2 then
          jl.         4          ;
          jl.         less.      ;  goto less;
```

Since the dataformats of RC 8000 is sufficient for nearly all applications little attention has been paid to the ease of implementation of multilength arithmetic. Thus no facility for direct carry propagation is included. The causual reader may skip the following example of the use of exception bit 23 in multilength addition.

Example 2:

```
;Routine for multilength addition
;The routine performs addition of multilength integers
;represented in consequtive doublewords.
;At call the registers must contain
;  w0: length of integers in halfwords:
;      integral multiplum of 4, at least 8.
;  w1: address of last word of first integer
;  w2: address of last word of second integer
;  w3: link
;At return the result will have overwritten the second integer.
;Return is made to link + 2, i.e. the second instruction after
;the call, except in case of total overflow, where return is to
;link.

          0                    ;
dzero:    0                    ; double length zero
done:     1                    ; double length one
stop:     0                    ; saved stop criteria
return:   0                    ; saved return
```

```
multiadd:  ws  w2     0          ; entry point:
           al  w2   x2+4          ; stop criteria;
           ds. w3     return.     ; save stop criteria and return;
           al  w2   x2-4          ;
           wa  w2     0           ; restore second pointer;
           xl.        0           ; clear ex;

next:      dl  w0   x1            ; loop for addition of tail parts
           sx         2.01        ; if ex(23) then
           aa. w0     done.       ; add carry;                    x
           xs.        oldcarry.+1 ; save ex in displ. below       x
           aa  w0   x2            ; add second tailpart           x
           ds  w0   x2            ; save partial result

oldcarry:  al  w0     0-0-0       ; load saved ex from carry
                                  ; addition
           sz  w0     2.01        ; if saved carry then
           xl.        1           ; ex(23):= 1;
           al  w1   x1-4          ;
           al  w2   x2-4          ; decrease pointers;
           sl. w2    (stop.)      ; if more tail parts then
           jl.        next.       ; loop;
           dl  w0   x1            ; load head of first
           sx         2.01        ;
           am         done-zero   ; if ex(23) then                y
           aa. w0     dzero.      ; add carry; note ex always
                                  ; assigned                      y
           xs         3           ; w1:= oldex:= ex;              y
           aa  w0   x2            ; add head of second;           y
           ds  w0   x2            ; save result;
           sx         2.01        ; if overflow then
           al  w1   x1+2          ; oldex. overflow:=
                                  ; -, oldex. overflow;
           rl. w3     return.     ;
           sz  w1     2.01        ; if oldex. overflow then
           jl       x3            ; return to link
           jl       x3+2          ; else return to link + 2;
```

The corresponding routine for multilength subtraction can be obtained by changing instructions marked "x" by:

```
           jl.        4           ; if -, carry then
           ss. w0     done.       ; subtract borrow;
           xs.        oldcarry.+1 ; save ex in displ. below;
           ss  w0   x2            ; subtract tailpart;
```

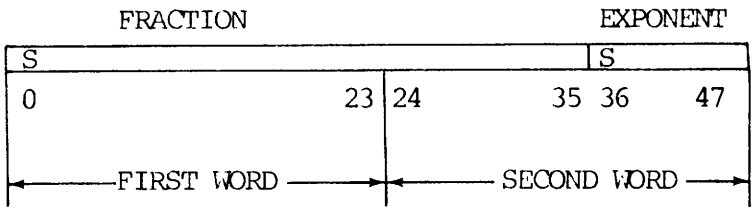
and instructions marked "y" by:

```
        am          dzero-done ; if -, carry then
        ss. w0      done.      ; subtract borrow;
        xs          3          ; w1:= oldex:= ex;
        ss  w0  x2             ; subtract first doubleword;
```

5 Floating-point Arithmetic

5.1 Number Representation

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



The address of a floating-point number refers to the second word of the 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 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)    F negative
                     F =    0*2**(-2048)    F zero
    0.5*2**(-2048) <= F <1*2**2047          F positive
```

or approximately

```
    10**(-616) < abs(F) < 10**616
```

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

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

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

Accordingly, the sign or zero value of a floating-point number

may be determined by examining its first word only. This can be done by means of skip instructions.

The following instructions will load a floating-point number in w0 and w1 and test whether it is negative:

```
dl. w1    float.    ; w0-1 = number   float points to
                      second word
sh  w0    -1          if number < 0 then
jl.       negative.; goto negative
```

5.2 Arithmetic Operations

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. 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 is formed by repetition of an add-and-shift cycle. 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 is formed by the non-restoring division method. Rounding of the quotient is performed as described below. The remainder is thrown away.

5.3 Normalization and Rounding

If the resulting fraction is zero, a floating-point zero with exponent -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.

5.4 Underflow, Overflow and Non-normalized Operands

Underflow and overflow occur when the exponent of the final result (after normalization, rounding and renormalization) is less than -2048 or greater than 2047, respectively. This will be registered in the exception register as shown below and will lead to a floating point exception if the "floating point exception active" bit is set in the status register.

After underflow or overflow, the fraction is correct, while the exponent is taken modulo 4096.

Division by zero leaves the register operand unchanged and is defined as overflow even if zero is divided by zero. Considering the 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. Results of operations on non-normalized numbers must be regarded as undefined, they may be non-normalized and they may be different on different versions of RC 8000. See however the descriptions in chapter 10.

The exception register has the following meaning after a floating point operation:

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

5.5 Floating Point Instruction

The "floating point : add to double register" instruction, "fa", adds the contents of the doubleword pointed out by the effective address to the specified double register. If the effective address is 0 through 7, the operation is a doubleregister - double-register operation. Reference to non-accessible area and to the lower limits of the process area (addr = lowlim - base) and the common protected area (addr = 8) leads to program exception. Underflow and overflow conditions are indicated in the exception register. If the "floating point exception active" bit is set in the status register underflow and overflow will lead to a floating point exception. After underflow and overflow the fraction of the result is correc while the exponent is taken modulo 4096.

The "floating point : subtract from double register" instruction, "fs", subtracts the contents of the doubleword pointed out by the effective address from the specified double register. Otherwise as "fa".

The "floating point : multiply by double register" instruction, "fm", multiplies the specified double register with the contents of the doubleword poin ted out by the effective address. Otherwise as "fa".

The "floating point : divide into double register" instruction, "fd", divides the specified double register with the contents of the doubleword pointed out by the effective address. The quotient is placed in the doubl register, the remainder is thrown away. Otherwise as "fa".

5.6 Number Conversion

Mixed arithmetic is supported by number conversion instructions and normalizing shift instruction. The normalize instructions may of cause be used for other purposes.

The "convert integer to floating point" instruction, "ci", converts the integer value of the specified register to a floating point number and places the result in the corresponding double register. The effective address is used for scaling, such that the value of the result is two to the power of effective address times the integer. Floating point overflow and underflow may occur through scaling and is handled as for the floating point instructions.

The "convert floating point to integer" instruction, "cf", con-

verts the floating point value of the specified double register to an integer and places the result in the least significant register. The effective address is used for scaling, such that the value of the result is two to the power of effective address times the floating point number rounded to nearest integer. Carry = 0 and integer overflow is registered in the exception register. Integer overflow will lead to integer exception if the "integer exception active" bit is set in the status register.

The "normalize single register" instruction, "ns", normalizes the contents of the specified working register by shifting it to the left until bit 0 and bit 1 have opposite values. The negative number of shifts necessary is stored in the halfword pointed out by the effective address. Normalization of zero will give zero and store the value -2048.

Effective address 0 through 7 will cause the negative shift value to be assigned to the corresponding half register. Other references outside the process area will lead to program exception.

The "normalize double register" instruction, "nd", works as "ns" but on the specified double register.

5.7 Examples

```
; variables and constants; addresses of doublewords point at second
; word
fhalf      : 0.5                          ; rounding of floating
                                          ; point
roundconst: d.1024                        ; rounding double integer
                                          ; = bit 37
integer1   : <24 bit integer value>       ; variable
integer2   : <24 bit integer value>       ; -
float1     : <floating point value>       ; -
float2     : <floating point value>       ; -
dinteger   : <48 bit integer value>       ; -
```

```
; mixed arithmetic

            rl. w1      integer1. ;
            ci  w1      0         ;
            rl. w3      integer2. ;
            ci  w1      0         ;
            fd  w1      6         ;
            ds. w1      float1.   ; float1 = integer1/integer2

            rl. w1      integer1. ;
            ci  w1      0         ;
            fa. w1      float1.   ;
            cf  w1      0         ;
            rs. w1      integer2. ; integer2 = integer1 + float1

; conversion of double register to floating point

            dl. w1      dinteger. ;
            nd. w1      3         ; negative shifts are
                                  ; stored as displacement
                                  ; in next instruction
            al  w2      0-0-0     ; note notation for changed
                                  ; displacement
            sn  w2     -2048      ; if dinteger = 0 then
            jl.         assignexp.; goto assignexp; result
                                  ; = 0.0
            al  w2   x2+48        ; exponent = - shifts + 48
            ad  w1     -1         ; prevent rounding overflow

            aa. w1     roundconst.; add bit 37 to round
            nd. w1      3         ; renormalize
            al  w2   x2+0-0-0     ; exponent = exponent
                                  ; -(1 or 0)
assignexp:  hs  w2      3         ; assign exponent
            ds. w1      float1.   ; float1 = dinteger
```

```
; if rounding of large double integers is not needed the code is
; reduced to

          dl. w1     dinteger. ;
          nd. w1     3         ;
          al  w2               ;
          se  w2     -2048     ;
          al  w2     x2+47     ;
          hs  w2     3         ;
          ds. w1     floatl.   ; floatl = dinteger

; conversion of floating point to double integer

          dl. w1     floatl.   ;
          fa. w1     fhalf.    ; round
          el  w2     3         ; load exponent
          ad  w1     -12       ; remove exponent bits
          ad  w1     x2-35     ; shift according to exponent
                               ; overflow may occur
          ds. w1     dinteger. ; dinteger = floatl
```

Without giving examples it may be noted that correct integer arithmetic with 35 significant bits can be simulated by the floating point instructions.

6 Monitor Control

The RC 8000 is designed to operate a multiprogramming system under the control of a monitor program, defining the specific multiprogramming environment. Full monitor control is ensured by the protection system, which consists of the memory protection and the concept af privileged instructions. The memory protection is described in chapter 3.

Privileged instructions are the i-o instructions and the instructions that may change the protection situation. These instructions may only be executed in monitormode.

The monitor program is entered through interrupts and monitor calls. The current user program is suspended, registers and status are dumped and the monitor program is entered according to the cause with memory protection and program relocation disabled. Dump and entry addresses are defined by a system table pointed out by the system information register, "inf". This table and the register must be set up by the monitor program according to monitorstructure and current system status. User programs are reactivated through the privileged "return interrupt" instruction, which restores the dumped registers and the protection situation.

6.1 System Table

The system table consists of 6 consequtive words with the following contents.

```
inf-5 : monitor call service address
inf-3 : interrupt service address
inf-1 : status/intlim initialization
inf+1 : register dump address
inf+3 : exception service address
inf+5 : escape service address
```

The system information register, inf, points to the second halfword of the status/intlim initialization.

The exception and escape service addresses define register dumps and entry points for the current userprogram's use of the exception and escape facilities, see further 6.5.1 and chapter 7.

These addresses are logical addresses, the remaining addresses are physical addresses in true memory.

6.2 Entering the Monitor Program

Entry of the monitor program proceeds in the same way for monitorcalls and interrupts except for the initial assignment of cause and the final selection of service address.

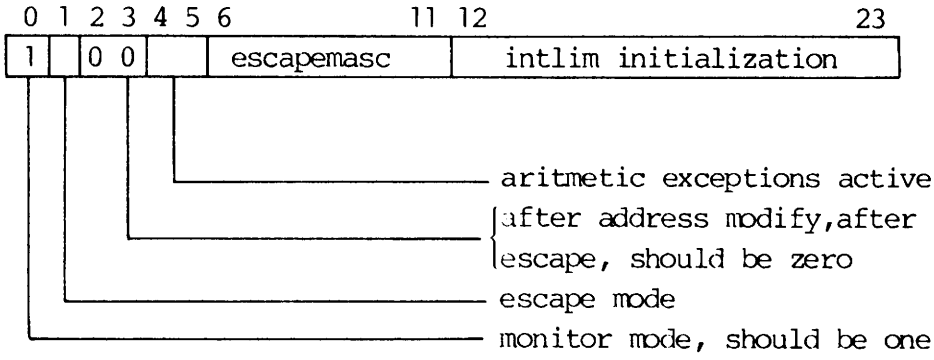
The 8 dynamic registers are dumped starting at the register dump address:

register dump address	+ 0	:	w0
-	+ 2	:	w1
-	+ 4	:	w2
-	+ 6	:	w3
-	+ 8	:	status
-	+10	:	ic
-	+12	:	cause
-	+14	:	addr

The 12 most significant bits of the status register is assigned the value of the most significant halfword of status/intlim initialization. The 12 least significant bits are cleared.

The enable limit in the intlim register is assigned the value of the 12 least significant bits of status/intlim initialization, the disable limit is set to zero, see further 6.6.

Status/intlim initialization:



The remaining process definition registers are initialized as follows:

```
base   = 0
lowlim = 8
uplim  = size, the size register must contain memory
         size
```

This initialization disables program relocation and memory protection.

The working registers are initialized thus:

```
w0 =      unchanged user value
w1 =      register dump address + 16
w2 =      cause
w3 =      unchanged user value
```

After this the relevant service address is fetched from the system table. The inf register is decreased by 12 defining a new system table and the monitor program is started at the service address.

The definition of a new system table and the way status is initialized allows use of the exception and escape facilities in connection with the monitor program and allows special handling of not disabled interrupts during execution of the monitor program. Total disabling is obtained by specifying an intlim initialization zero. Interrupts that can not be disabled, i.e. the internal interrupts see 6.5.1, will halt the cpu if intlim is zero.

Errors during entering the monitor program will increase the dump error count in status, decrease inf by 12 and repeat the action with the new system table. For details see chapter 10.

6.3 Reactivation of User Program

Reactivation of the user program is done by execution of the "return from interrupt" instruction, "ri".

The "inf" register is increased by 12 reselecting the system table valid at monitor entry. The monitor program selects the user program by assigning values to this system table.

The 8 dynamic registers are restored from the register dump pointed out by the register dump address in the systemtable.

The process definition registers are restored from 5 consequtive words pointed out by the sum of the register dump address and the effective address of the ri instruction:

```
register dump addr  + effective address + 0 : cpa
                    -                   + 2 : base
                    -                   + 4 : lowlim
                    -                   + 6 : uplim
                    -                   + 8 : intlim
```

The user program is entered at the dumped instruction counter.

"ri" is a privileged instruction.

6.4 Monitor Calls

Programmed activation of the monitor program from a user program is executed as a special case of the "jump and select disable limit" instruction, "jd", see 6.6. If the effective address of the "jd" instruction fullfill the condition:

$$-2048 \leq addr < -2048 + function\ limit$$

where

$$function\ limit = 2048 - montop$$

a monitor call is executed. The montop register must have been initialized by the monitor program. The address space used for definition of monitor calls would otherwise have lead to program exception, since it is contained in the non-accessible area.

After the common interrupt and monitor call action described in 6.2, the monitor program is entered at the monitor call service address specified in the system table. The cause is defined as function number equal to effective address plus 2048. If the least significant bit of the function number is ignored a fast selection of the monitor function is obtained in the monitor program by

```
call entry    : jl. (x2 + functiontable.) ; switch to function
functiontable: function 0  ; address of code for function 0
               function 2  ;            -                2
                  -        ;            -
                  -        ;            -
```

Parameters and results may be exchanged between the user program and monitor program via the register dump. Return is through the "ri" instruction.

6.5 Interrupts

RC 8000 has a program interrupt system with priority levels. The interrupt levels are described by levelnumbers, lowest numbers having highest priority.

6.5.1 Internal Interrupt

The level numbers 0 through 5 are assigned to program events defined as internal interrupts. These interrupts can not be disabled.

Level no.	meaning
0	program interrupt
1	integer interrupt
2	floating point interrupt
3	system table error, see chapter 10
4	buserror in operand reference
5	buserror in instruction fetch

Program interrupt is defined when a program exception occurs and the exception service address in the system table is zero or when an escape takes place with escape service address zero.

Integer and floating point interrupt ire analogous to program interrupt, but refer to arithmetic exceptions.

Buserror interrupts are usually triggered by memory faults. See further 8.2.

6.5.2 Power Failure and Timer

Level 6 is assigned to powerfailure interrupt. The power failure interrupt leaves at least 1 millisecond to prepare the power restart and force a halt. Level 6 should only be disabled in exceptional situations and only for very short periods.

Level 7 is assigned to timer interrupt. RC 8000 is supplied with a 16 bit real time clock register, rtc, which is increased by one every 0.1 millisecond. A timer interrupt is generated every timer interrupt period, 25.6 milliseconds in most of the RC 8000 models.

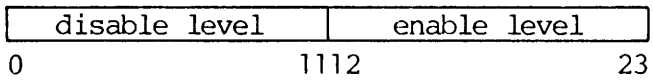
6.5.3 External Interrupts

The remaining levels are used for exteral events. External interrupts are defined by the processors on the unified bus by addressing the cpu and transferring the levelnumber as data. The number of levels available depends on model. Assignment of external levels is under program control, see further chapter 8.

6.6 Disabling of Interrupts

Interrupt request are registered in the interrupt register "intreg", one bit for each level. "Intreg" is continously scanned. When a bit equal one is found an interrupt flag is set and the corresponding level number is saved in the current level register "curlev", and the scan is stopped. Disabling is implemented by restricting the scan to level numbers lower than or equal to a limit defined by the interrupt limit register and the disable bit in the status register.

Interrupt limit register:



When the "disable" bit in "status" is one, the scan limit is defined as the disable limit and when it is zero as the enable limit.

The "disable" bit is controlled by the special jump instructions:

The "jump and select enable limit" instruction, "je", sets the "disable" bit to zero and proceeds as a "jl" instruction.

The "jump and select disable limit" instruction, "jd", tests the effective address against the monitor call definition, 6.3, if it is not a monitor call the "disable" bit is set to one and the "jl" actions are executed.

A normal user program will have "disable level" equal to "enable level" equal to maximum level number, this means no disabling and "je" and "jd" reduced to normal jumps except for the monitor call function of "jd".

The monitor program is entered with disable bit zero, disable level zero and enable level initialized from the systemtable, usually to 6, disabling all levels but power failure.

6.7 Interrupt Response

The interrupt flag is examined at the completion of every instruction. If the flag is set, the cause is defined as two times the saved levelnumber in "curlev" and the corresponding bit in "intreg" is cleared. After the common interrupt and monitor call action the monitor is entered at the interrupt service address defined in the system table, with the interruptflag cleared and the interruptscan restarted according to the new value of "intlim".

6.8 Control of Monitor Registers

The "general put into processor register" instruction, "gp", transfers the contents of the specified working register to the register pointed out by the effective address. "gp" is a privileged instruction.

Effective addres	Effect
26	inf = working register
30	size = working register
32	montop = working register
94	the contents of the working register is interpreted as an interrupt level and the corresponding bit in "intreg" is cleared. Only used at power restart and in testprograms.
other	depends on model, may be destructive, used for technical testprograms.

The "general get from processor register" instruction, "gg", transfers the contents of the register pointed out by the effective address to the specfied working register.

Effective address	Effect
26	working register = inf
30	working register = size
32	working register = montop
100	working register = rtc The contents of the 16 bit real time clock register is transferred to the 16 least significant bits of the working register. Remaining bits are cleared. rtc is unchanged.
other	depends on model. Used for technical testprograms.

7 Exceptions and Escape

An exception occurs, when the protection system is violated, when execution of an unassigned instruction code is attempted and, if wanted, when results of arithmetic operations are outside the range of number representation. These events are caused by programming errors. As an aid in finding these errors the exception concept offers the possibility of register and statusdump combined with activation of a programmed exception routine. A large part of programming errors will result in exceptions, but in many cases the original cause is obscured. The escape facility provides a tool for tracking down these and other errors and for gathering information on program flow and data usage to aid in program tuning.

7.1 Exception

When an exception occurs the exception service address is fetched from the system table. If the address is zero, the monitor program is called through an internal interrupt. If the address is nonzero, the 8 dynamic registers are dumped and the program is restarted at the address following the dump.

7.1.1 Register Dump at Exception

address		contents after exception
exc. service addr.	+ 0:	w0
-	+ 2:	w1
-	+ 4:	w2
-	+ 6:	w3
-	+ 8:	status
-	+10:	ic
-	+12:	cause
-	+14:	addr
-	+16:	<entry point for routine>

The register dump must be inside the process area defined by the limit registers. The registers are dumped with the values they had when the exception occurred. Exact interpretation may require knowledge of the details in instruction excecution given in chapter 10, but in most cases the following general principles are sufficient for the analysis.

The cause register gives the cause of the exception

cause = 0 : program exception i.e. protection violation or unassigned instruction code
cause = 2 : integer exception i.e. overflow
cause = 4 : floating point exception i.e. over- or underflow

The instructon counter points to the word after the instruction causing the exception.

7.1.2 Program Exception

Exception caused by attempts to execute privileged or unassigned instruction codes leaves working registers and status unchanged but the address calculation has been carried out before the exception is recognized and the result is given in the dumped addr register.

At memory protection violation the addr register contains the logical address causing the violation. In multiword instructions this address may point to any of the addressed words. Early steps in the instruction may have changed the contents of the specified working register(s). Violation in indirect address calculation will leave the intermediate address in the addr register and set the after am bit in the dumped status register, this bit is zero at all other exceptions. This indication of exception in indirect addressing is not implemented on RC 8000/45.

7.1.3 Arithmetic Exceptions

At arithmetic exceptions the exception part of the status register expand the cause. For memory referring arithmetic instructions the addr register contains the logical address of the operand, (for doubleword instructions the logical address of the preoperand).

7.1.4 Exception Routine

Before entering the exception routine the registers are initialized thus:

w0: dumped instruction counter
w1: w-field of cause instruction
w2: dumped addr register
w3: cause
ex: dumped exception register

Protection, relocation, escapemode, escapemask, "integer exception

active" and "floating point exception active" are unchanged.

What the exception routine does is up to the programmer.

A primitive exception routine regards any exception as an error, the register dump is output for analysis by the program maintenance staff and the program is aborted after dumping of program and variables.

An advanced exception routine classifies the exceptions as non-repairable, repairable and intentional.

Non-repairable exceptions may be handled as above. Repairable exceptions trigger a corrective action after which return is made to the instruction pointed out by the dumped instruction counter.

Example: repairing of floating point exception:

```
                                    ; replaced by dump of
exaddr:   0                         ; w0
w01:      0                         ; w1
          0                         ; w2
w23:      0                         ; w3
          0                         ; status
ic:       0                         ; ic
          0                         ; cause
          0                         ; addr
                                    ;
entry:    sn  w3     4              ;if cause = 4 then
          jl.        float.         ; goto float
          -                         ;
          -                         ; code for non-repairable error
          -                         ;

float:    ls  w1     1              ; w1=2*w1, w1=rel addr of reg in
                                    ; dump
          al  w2  x1-2              ; w2=w1-2
          sh  w2    -1              ; if w2 < 0 then
          al  w2     6              ; w2=6, w2= rel addr of reg pre
                                    ; in dump
          sx         2.1            ; if underflow then
          jl.        underflow.     ; goto underflow
          rl. w0  x2+exaddr.        ; w0= first part of result,
                                    ; prepare last sign of result
          sl  w0     0              ; if result ≥ 0
          am         pos-neg        ; then w3-0 = maxpos
          am         neg-fzero      ; else w3-0 = macneg
underflow: dl. w0     fzero.        ;  underflow : w3-0=zero
          rs. w3  x2+exaddr.        ; overwrite dumped result
          rs. w0  x1+exaddr.        ; with corrected result in w3-0
          dl. w1  w01.              ; restore w0-1 contents
          dl. w3  w23.              ; restore w2-3 contents
          jl.    (ic.)              ; continue with instruction after
                                    ; exception.

fzero:    0.0                       ; floating point zero, double-
                                    ; word, addr points to second
                                    ; word.
neg:      -1.6x10**616              ; maximum negative floating
                                    ; point number
pos:       1.6x10**616              ; maximum  positive floating
                                    ; point number
```

The coding of this example aims to illustrate practical use of the facilities in address calculation f.ex. the am-construction loading the corrected result. Floating point exceptions are repaired by substituting the result with zero in case of underflow and a maximum number with correct sign in case of overflow.

Intentional exceptions may be produced by inserting illegal instructions as breakpoint to trigger testoutput or similar action. A more sophisticated use is to program in such a way that rare conditions automatically will result in an exception f.ex. through a table lookup that in all normal cases gives a legal address, but in special cases gives an illegal address. This may give a speed-up of a program by eliminating the need for in-line testing for special cases.

Exception routines can as shown return to the program through a normal jump. It is however in some cases useful to consider to return by means of the "return from escape" instruction, see further 7.2.5 and 7.2.6.

7.2 Escape

The escape facility is controlled by the escape mode bit and the escape mask in the status register. If the escape mode bit is zero the escape facility is inactive. If it is one the escape facility is active, which means that every instruction execution include an escapetest which may lead to an escape depending on the result of a logical comparison between the escape masc and an escape pattern defining the properties of the instruction.

7.2.1 Escape in Indirect Address Calculation

The escape action is triggered after the completion of the address calculation and before any instructon execution has taken place. Information is set up for fast analysis and for resuming the execution. There is one exception, if the escape mask specifies escape on memory load, statusbit 10, an escape is triggered by the memory load in indirect address calculation. In this case preparation is made for resuming the address calculation with the load of the effective address, after which a normal instruction escape may take place, see further below.

7.2.2 Escape Mask and Escape Pattern

As described in 2.2.1 the escape mask occupies bit 6-11 of the status register. The meaning of the bits are given below. If a bit in the escape mask is one and escapemode is active the escapeaction will be triggered for all instructions having the corresponding property. The instruction properties are described by their escape patterns.

status-bit	property
6	privileged,jd,je, unassigned
7	may modify ic,jumps, skips, ri and re
8	multiword operand, doubleword instructions, ri and re.
9	stores in memory
10	loads from memory
11	auxiliary bit

The auxiliary bit is used in the escape patterns to ensure that an all one escapemask trigger on all instructions. Escape patterns are given in the instruction tables Appendix 1 to 3.

7.2.3 Escape Action

When the escape action is triggered the escape service address is fetched from the system table. If the address is zero, the monitor program is called through an internal interrupt. If the address is nonzero, the 8 dynamic registers are dumped and the program is restarted at the address following the dump.

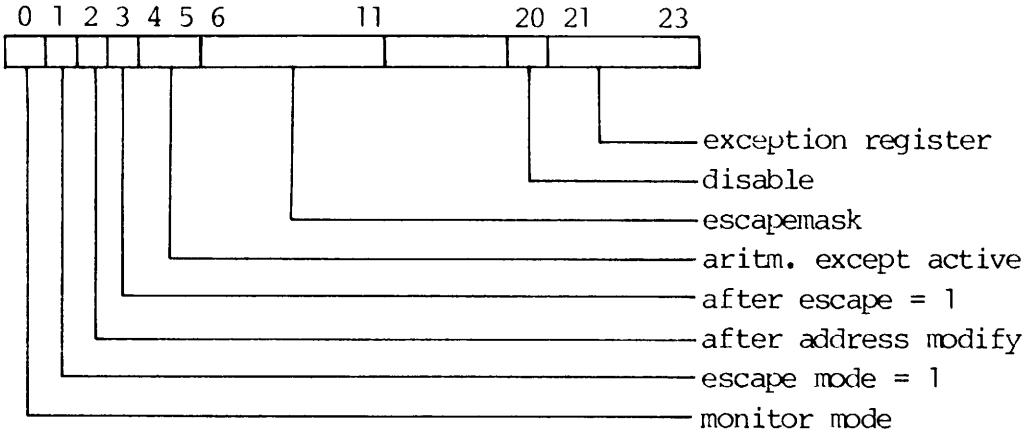
7.2.4 Register Dump at Escape

address			content after escape
escape service addr	+ 0	:	w0
-	+ 2	:	w1
-	+ 4	:	w2
-	+ 6	:	w3
-	+ 8	:	status
-	+10	:	ic
-	+12	:	cause
-	+14	:	addr
-	+16	:	<entry point for routine>

The working registers are dumped with the values they had at completion of the previous instruction.

The status register is dumped in the same way, except that the "after escape" bit is one, and that the "after address modify" bit is zero for a normal escape and one for an escape in indirect address calculation.

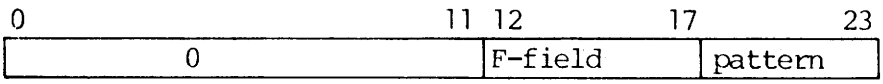
Status register



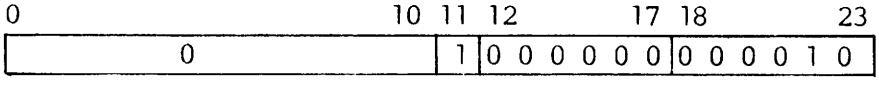
The dumped instruction counter points to the instruction causing the escape.

The dumped cause register specifies the cause by the F-field and the escape pattern of the instruction.

Cause register at normal escape:



Cause register at escape in indirect address calculation:



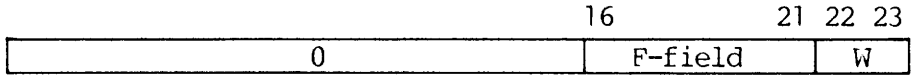
The dumped addr register contains the effective address of the instruction except at escape in indirect address calculation where it contains the logical address pointing to the effective address.

7.2.5 Escape Routine

Before entering the escape routine the working registers are intialized thus:

w0 : dumped instruction counter

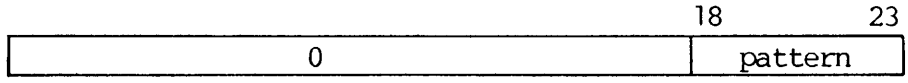
w1 : F-field and W-field of instruction



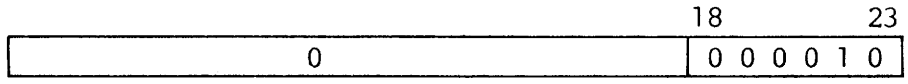
or at escape in indirect address calculation

w2 : dumped address register

w3 : escape pattern



or at escape in indirect address calculaton



Monitormode, disable and exception register are left unchanged, escapemode, escapemask and arithmetic exceptions active are cleared in the status register.

What the exception routine does is up to the programmer. A few examples and hints are given later in this chapter.

7.2.6 Return from Escape

The register dump is prepared for return to the instruction execution through the "return from escape" instruction. If the dump is not modified the instruction is completed as if the escape had not taken place at all.

The "return from escape" instruction, "re", restores the working register, status, ic, cause and addr from 8 consecutive logical addresses starting at the effective address of the instruction. Each of the logical addresses must be either in the process area or in the common protected area or a program exception will take

place. The address calculation in the instruction pointed out by the restored ic will be modified according to the value of the "after address modify" and "after escape" bits in the restored status register as follows:

after address modify	after escape	meaning
0	0	normal address calculation
0	1	no address calculation addr is used directly
1	0	as after an "am" instruction
1	1	addr = word (addr)

In the last case another escape may be triggered by the same instruction depending on escapemode, escapemask and escape pattern. After address calculation the execution of next instruction proceeds normally.

Modification of the register dump or selection of a simulted register dump is possible. This means that the "re" instruction may be looked upon as a combined jump and register initialization. Note however that attempts to switch to monitor mode is ignored, i.e. the monitor mode bit can not be changed from zero to one, but it may be changed from one to zero.

7.2.7 Examples and Hints

The first example shows a simple jump tracing with a window. The aim is to trace the program flow leading to an exception, the cause of which can not be found from the exception information. The exception triggers a program dump for further analysis, so the tracing need only generate memory information. Escape mode, escape serviceaddress and an escape mask only containing "may modify ic" ("status" bit 7) are supposed to have been set up by a monitor call.

```
                                    ; contents at entry of escape
                                    ; routine
escaddr:    0                       ; w0
            0                       ; w1
            0                       ; w2
            0                       ; w3
            0                       ; status
            0                       ; ic
            0                       ; cause
            0                       ; addr

entry:      se  w3      2.010000    ; if not true jump then
            re.         escaddr.    ; return, skips etc are
                                    ; ignored
            rl. w1      pointer.    ; load pointer

            al  w1  x1+4            ; increase pointer
            sl. w1      top.        ;
            al. w1      first.      ; cyclically
            rs. w1      pointer.    ; save increased pointer
            rs  w0  x1              ; save from addr
            rs  w2  x1+2            ; save to address
            re.         escaddr.    ; return

pointer:    top                     ; initial value ensures
                                    ; correct start

first:      0                       ; first word in cyclical buffer
            0                       ; of double words for storing
            -                       ; of trace information
            -                       ;
            0                       ; last doubleword
            0                       ;
top:                                ;
```

Skips are ignored since the escape pattern for skips contain the auxiliary bit. For each true jump the position and the effective address is stored cyclically in the buffer. The programdump triggered by the exception will have the n latest jump leading to the exception displayed in the buffer.

Next example supposes following situation. A programming error is found to be caused by a wrong contents of a memory word with the address, memaddr. Study of the programtext does not reveal the

illegal updating. The escaperoutine of the example identifies all store references to the memory word, but does not include the explicit action on a hit. The escape mask is "stores in memory".

```
                                    ; content at entry of escape
                                    ; routine
escaddr:  0                         ; w0
          0                         ; w1
          0                         ; w2
          0                         ; w3
          0                         ; status
          0                         ; ic
          0                         ; cause
          0                         ; addr

entry:    sz  w2     1              ; make effective address even
          al  w2  x2-1              ;
          sn. w2    (critical.)     ; if effective addr = critical
                                    ; addr
          jl.        hit.           ; then goto hit
          so  w3     2.001000       ; if escape pattern does not
          re.        escaddr.       ; contain multiword bit then
                                    ; return
                                    ; next test is thus only executed
                                    ; for "ds"
          al  w2  x2-2              ;
          se. w2    (critical.)     ; if effective adress of pre-
                                    ; operand <>
          re.        escaddr.       ; critical address then return
hit:      -                         ;
          -                         ; code for a hit



          -                         ;
          re.        escaddr.       ; return
critical: memaddr                   ; address of critical word
```

These simple examples illustrates the use of the escape facility as a debugging tool. Another use of the facility is study of program behaviour with escape routines generating various kinds of statistical information.

The escape facility should not be used as a direct programming tool mainly because of the overhead involved. This argument does not cover use for theoretical studies.

It is tempting but not recommended to use the "re" instruction for initialization of registers and return from subroutines. The catch is the implicit assignment of the status register, which, unless one is very carefull, will eliminate independant use of the escape facility and independant control of the arithmetic exceptions.

There is one important exception to this rule. The "re" instruction is very well suited to return from exception routines. In this case the status register can be assigned based on the dumped status which takes care of above argument. The following example shows the structure of an exception routine handling pagefaults in a paging system programmed in such a way, that a pagefault shows up as a negative effective address in the instruction referring the page. After adjustment of the pagesituation the instruction is repeated with correct effective address.

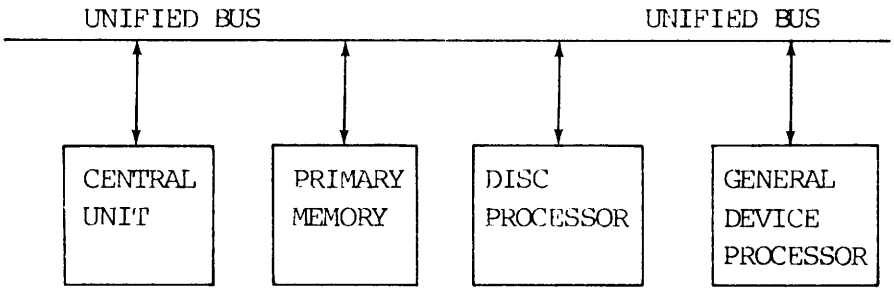
```
                                    ; contents at entry of exception
                                    ; routine
excaddr:    0                       ; w0
            0                       ; w1
            0                       ; w2
            0                       ; w3
status:     0                       ; status
ic:         0                       ; ic
            0                       ; cause
addr:       0                       ; addr

entry:      sl  w2      0           ; if not pagefault then
            jl.         error.      ; goto error
                                    ;
            jl. w3      getpage.    ; call getpage
                                    ; returns with referred page
                                    ; in memory, updated pagetable
                                    ; and w2 pointing to referred
                                    ; word on page
                                    ;
            rs. w2      addr.       ; overwrite dumped addr with
                                    ; corrected
            dl. w2      ic.         ;
            lo. w1      aftesc.     ; set after escape in dumped
                                    ; status
            al  w2  x2-2            ; dumped ic = dumped ic-2
            ds. w2      ic.         ;
            re.         excaddr.    ; return


aftesc:     2.0001 0000 0000 0000 0000 0000 ; after escape bit.
```

In this simple example programming errors leading to negative effective addresses is handled as pagefaults. Further checks are supposed to take place in the getpage routine.

8 Input/Output System

8.1 Main Characteristics

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 are 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 contents to the central processor. The current bus master (see 2.5.1) may interrupt a central processor on the bus by addressing and transferring the level number to a specific register in that processor.

8.2 Input and Output Operations

All input and output operations are handled by two privileged instructions, "data in", "di" and "data out", "do", 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.

8.2.1 Data In Instruction

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

The CPU clears bit 21-23 in the status register (i.e. the exception register) and addresses the device register. If the address is correct, the device places the data on the bus and sends an acknowledge signal. The CPU receives the data and this signal, and checks the data for parity completing the transfer by placing the received data in the specified working register.

If the received data contains a parity error, the parity error status bit is set in the exception register.

If the device accepts the addressing, but is unable to handle the request, it sends a not-acknowledge signal, rejecting the operation. If this signal is received, the communication error status bit is set in the exception register.

If no signal whatsoever is received by the CPU inside a maximum responsetime, the timeout status bit is set in the exception register. Since parity check is carried out independantly on the undefined data, parity error is usually set too. Timeout is caused by no one recognizing the address, i.e. noneexisting device, devicefailure or error in address.

8.2.2 Data Out Instruction

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

The CPU clears the exception register, addresses the device register and places the data in the specified working register on the bus. If the address is correct, the device checks the received data for parity error and sends an acknowledge signal completing the transfer.

If the data contains a parity error, the device sends a not-acknowledge signal rejecting the operation.

If the device is unable to handle the request, it also sends a not-acknowledge signal. In both cases the cause is indicated in the device status register according to device specifications. Communication error and timeout is indicated in the exception register as for Data In. Parity error is not possible.

8.2.3 Exception Indication

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

Bit	Input	Output
21	bus parity error	0
22	bus timeout	bus timeout
23	bus communication error (device dependent)	bus communication error (device dependent)

8.2.4 Memory Addressing

It is inherent in the unified bus concept that addressing of device registers in no way differs from the addressing of memory words. Therefore the data in and data out instructions may be used for memory reference in monitor mode. Bus communication error during input will signal memory error in the addressed word. I/O reference of memory may be used for memory test and for test of memory size and relieves the programmer from handling businterrupts.

The memory references in instruction fetch and operand references follow the same pattern, but the assignment of the exception register is only carried out in case of errors and is accompanied by generation of an internal interrupt as described in chapter 6. The Data In and Data Out instructions differ from this in always assigning the exception register and never generating interrupts.

8.3 Standardized Block-oriented Device Controllers

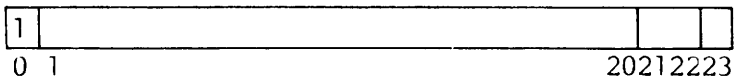
Standardized block-oriented controllers, such as the disc processor and the general device processor, are started by means of an output operation, which addresses the controller as described below. Here, the contents of the working register is irrelevant.

Once started, the controller fetches its commands from a 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.

8.3.1 Device Address

Device addresses have the following format:



Bit 0 Logical 1, indicating I/O address.

Bits 1:20 Device number. 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 number and a sub-device number. In a disc storage system, for example, bits 1:18 may contain the binary number of the addressed disc processsor, preceded by zeros, while bits 19:20 contains the logical number of one of the four disc drives.

The function of direct controller commands is defined by the effective address; the data transferred by the instruction is irrelevant in case of standardized blockoriented devices.

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

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

START CHANNEL PROGRAM causes the address controller to start the channel program pointed out by the first word of the device description (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.

8.3.2 Device Descriptions

The address of the device description is calculated using the device number (bits 1:20) as follows:

device base + device number x 8

The device base, which is common to all devices, is the contents of word 8 in the primary memory.

The device description contains the following:

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

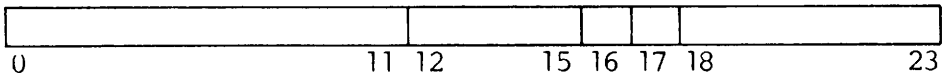
2nd word: Status address. First address of the area in the primary memory to which the standard status information is to be transferred at the termination of the channel program.

3rd word: Interrupt destination. I/O address of the central processor to be interrupted at the termination of the channel program. The format is a device address. In single CPU-systems the CPU will usually have device number 0. In special cases interrupt destination may be a memory address.

4th word: Interrupt level. Interrupt level to be transferred to the central processor when delivering the interrupt, see 6.5.3..

8.3.3 Channel Program

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. I.e. while two parameter words are required, the contents may be irrellevant.

The parameter FIRST CHAR ADDRESS specifies the start address of the memory area to or from which characters (i.e. 8 bit units) are to be transferred or fetched.

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

The parameters DATA 1 and DATA 2, will be interpreted in a device dependent way.

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 for check reading and 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.

8.3.4 Standard Status Information

Standard status information is transferred as 4 words 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. The count may be modified or left undefined by other commands. The stop command will leave it unchanged.

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.

The four standard status words are transferred as the first 12 characters by an unmodified sense command.

9 Power Restart and Autoload

Power restart and autoload are external signals that activate the corresponding actions immediately. Only features common to all models are described here.

9.1 Power Restart

RC 8000 is supplied with a power supervising logic that detects power failure and generates a restart signal on return of power. Built-in delays ensures against the effects of rapid oscillation of power.

A power failure will generate a power failure interrupt while the system still has at least one millisecond of operable time left. On systems with non-volatible primary memory this leaves time to prepare a power restart by saving the inf register or comparable action and force a halt, preventing undefined actions during the failure.

The power restart signal will activate the system at the power restart address, the contents of the word with address = 10. The following register initializations are performed:

register	initialization
status	monitormode, all other bits cleared
base	0, no relocation
lowlim	8
uplim	2047 x 4096, no memory protection
intlim	0, totally disabling
inf	1, non disable events leads to halt

The power restart code must reinitialize the registers montop, size and inf and must clear all bits in the interrupt memory. Further power restart actions are system dependant, but should include triggering of power restart actions in all device handlers and in the system clock routine.

After the specified power restart actions, return to the interrupted action is obtained by the return interrupt instruction.

9.2 Autoload

Autoload triggers the load of one block of program and data from a standard block oriented device. After loading control is transferred to the loaded program, which then continues loading and initialization.

RC 8000 has 2 autoload signals. The first is connected to an operators panel and triggers autoload from device number 4, which usually is a magnetic disc storage. The other triggers autoload from device 2. Device 2 is usually the input channel of a Frontend Processor interface. In this case the second autoload signal is connected to the interface and is activated from the Frontend after local selection of the load medium and preparation of the first block transfer. This connection is reciprocal, such that RC 8000 also can perform an autoload of the Frontend.

The autoload action starts by issuing a system reset signal on the unified bus. After that memory is initialized as described below, and the device corresponding to the autoload signal is activated.

address	initialization	interpretation
8	12-8 * device no.	device base
10	-4096 + 1536	power restart address
12	10	device description, 12-18
14	20	
16	256	channel program, 10-20
18	22	
20	768	
532	jl. 0	end-less loop

The device base value makes the device select the device description starting in address 12 regardless of device number.

The device description selects the channel program starting in address 10. Standard status will be delivered in words 20-24, and the interrupt level 22 will be delivered in memory word address 256, and not as an interrupt, see 8.3.2.

The channel program contains an init device command, command value 6 (parameters are irrelevant), and a read command with first address 22 and a character count of 768 defining load of 256 words to addresses 22-532. The first words of the load will

be interpreted as a continuation of the channel program, while the last word will overwrite the jl. 0 instruction and liberate the cpu from the end-less loop. Note that words 22-26 and 256 of the load will be overwritten when a stop command is reached in the continuation of the channelprogram.

Before the device is started and control is transferred to the end-less loop the following register initializations are performed:

register	initialization	meaning
status	0	user mode
base	0	no relocation
lowlim	534	all memory
uplim	534	writeprotected
cpa	534	only load read accessible
intlim	0	total disable, interrupts cause halt
size	2047x4096	nearly maximum
montop	2047	only monitor call 0 allowed
inf	525	a system table must be loaded just before the instruction overwriting the end-less loop

The initial underprivileged state, which can only be left through a monitor call 0 gives a very high assurance against destructive effects from erroneous autoloads. All errors during the autoload sequence leads to a halt.

If the autoload device is not activated (rejected or bus time out), device number is decreased by 1, device base is adjusted and the new device is activated. In this case the cpu will halt to await a new autoload signal.

Device number 1 is usually unassigned, such that unsuccesfull autoload from device 2 leads to halt.

Devicenumber 3 is usually the output channel of the Frontend interface. In this case the init command value 6 will trigger a Frontend autoload. This allows the operator to perform a system load from a Frontend device by disconnecting device 4 and press

the autoload button on the RC 8000 operator's panel, since the Front End autoload will end by activating the second autoload signal of RC 8000.

The following example gives a rough idea of how an autoload block may be structured (nummeric labels indicate resulting addresses in memory):

```
22:       15*256                ; stop command, will be overwritten
          0                     ; by the 3 last words of the stan-
          0                     ; dard status
check:    <value>               ; checksum of load block adjusted
                                ; for correct value of overwrite
start:    rl  w0     check      ; entry after load
          al  w2     22         ;
next:     ws  w0  x2            ; code for check of load
          al  w2  x2+2          ;
          sh  w2     532        ;
          jl         next       ;
          se  w0     0          ; if checksum error then force
          jl        -1          ; halt
          jd        -2048       ; enter monitor mode at init

init:     -                     ; code for further initialization
          -                     ; and loading
          -                     ;

256:      22                    ; overwritten by interrupt
                                ; level = 22
          -                     ;
          -                     ; more code
          -                     ;

          0                     ; rudimentary system table:
          0                     ; exception and monitorcalls
          0                     ; lead to half after jd - 2048
          init                  ; system table: mon call address
          0                     ;               interrupt address
          2048*4096             ;               mon mode/0
          534                   ;               reg. dump address
          0                     ;               exc address
          0                     ;               esc address
532       jl         start      ; overwrites end-less loop
                                ; goto start
```

10 Formal Description

10.1 Introduction

In this chapter the descriptions in the preceeding chapters are suplemented by a more formal description of the instructions and other functions. The information is given in an algorithmic form.

The presentation aims to give the user an understanding of the functional aspects of the cpu necessary for the full use of the systems facilities.

The presentation does not display actual microprogram structure, which varies from model to model by facilities for parallel elementary operations and by the amount of dedicated logic for special functions.

The algorithms will however reproduce the resulting states relevant to the user in programming, debugging and error diagnosis, except for the indicated deviations in handling of buserrors.

The indication of these and other deviations is given by comments. The format is usually

```
comment or:
   <alternative algorithm>;
```

The reader may ignore the alternative algorithms, since they are relevant only in extremely rare cases.

Arithmetic operations are not shown in full detail, the reader is referred to the general descriptions in chapters 4 and 5.

The descriptions of power restart and autoload are not repeated in algorithmic form, since they are considered sufficiently described in chapter 9.

The chapter is structured as follows:

Section 10.2 gives an explanation of the algorithmic notation.

Section 10.3 describes the operand access actions as procedures to be called from the algorithms. A few other common routines and

exits are also described here.

Section 10.4 describes the instruction sequencing and the control of address calculation, escape facility and instruction execution.

Section 10.5 describes the address calculation action except for the special cases covered in section 10.4.

Section 10.6 describes the instructions in alfabetical order by mnemonics. Exception and internal interrupts are shown as jumps to the following sections.

Section 10.7 describes the interrupt and monitor call actions and the "return from interrupt" instruction.

Section 10.8 describes the escape and exception actions and the "return from escape" instruction.

10.2 Notation

The algorithms in the following sections are given in a high level language. The language is similar to Algol, and should be understandable to most programmers.

The global declarations are given here together with a description of the ad hoc concepts. It is hoped that the meaning of the other language elements can be derived from the context and a general knowledge of high level languages.

10.2.1 Registers

The variables of the language are registers and subfields of registers. Registers are declared by specification of the first and the last bitnumber, which define the length of the register and the numbering of bits in the register, e.g. with the declaration ex(21:23) the second bit in the 3 bit exception register is ex(22).

The bitnumber notation ">n", e.g. ">15", which is used a few times means a fixed model dependent bit number greater than or equal to n.

The register structure is chosen for convenience in description and does not follow an actual implementation. Registers introduced in the preceeding chapters are:

```
register    w0, w1, w2, w3, status, ic, cause, addr,
            cpu, base, lowlim, uplim, intlim, montop,
            size, inf (0:23);
register    rtc (8:23);
```

Additional special registers are:

```
register    curlev (12:23);
            intreg (6:15); comment see 6.6;
register    pic (1:23),
            instruction (0:23),
            F (0:5), W, X, M (0:1), D (0:23),
            instrmask (0:23); comment see 10.4;
```

Auxiliary registers are used when convenient without explicit declaration. The implied declaration is usually auxq (0:23). Other declarations are indicated. Names of auxiliary registers end in a "q".

Subfields of registers are specified by first and last bitnumber, for onebit subfields only one number is given. The numbers refer to the declared numbering of the register.

10.2.2 Booleans

One-bit subfields may be used as booleans and booleans as one bit registers, they are true, if the bit is one, and false, if the bit is zero. The boolean "interruptflag" is introduced in section 6.6. Auxiliary booleans are used without declaration. The booleans used in connection with integer arithmetic and memory access are described below.

10.2.3 Integer arithmetics

Integer addition and subtraction are performed as two's complement operations. They are described by the normal operators "+" and "-". Assignments of the booleans overflow and carry are inplicit in each operation. Note however that composite operations described in one sentence are regarded as one operation with respect to overflow and carry, e.g.:

```
regpre:= regpre + op1q + carry;
```

The result of comparision as in "if addr < cpa then ..." is correct even if the implied subtraction would have caused overflow.

10.2.4 Memory access

Memory access is described by the notation word(address). The

notation implies an assignment of the booleans "bus parity error", "bus time out", "bus communication error" and "buserror", where "buserror" is the logical sum of the other booleans. The generality of the unified bus is illustrated by the use of the same notation in the "data in" and "data out" instructions to describe the access of device registers.

10.2.5 Bitpatterns

Bitpatterns can be built by concatination of registers, subfields and literals. The terms in a concatination are separated by the operator "con". The length of a concatination is the sum of the length of the terms. The number of bits in a literal is usually implicit, but it may be defined by f.ex. -2048(12) giving a literal of 12 bits with the value -2048. Assignment and other operations are performed between bitpatterns of equal length independent of bitnumbering in the operands. Remaining bits are unchanged, e.g. regw(12:23):= opq(1:11) leaves regw(0:11) unchanged. The operator "signextend" extends a bitpattern to the left with the first bit in the pattern until the required length is obtained. The length is defined by the other operands. Subfields with a negative bitlength are empty, they occur in a few algorithms. Signextension of an empty subfield reproduces the bit defined by first bitnumber. If a subfield specifies bitnumbers outside a register the corresponding bits are defined as zeros, e.g. regw(12:35)== regw(12:23) con 0(12).

10.2.6 Abbreviations

Following abreviations are used. The "==" notation means "equivalent to".

References to working registers:

```
regw    == case W of (w0, w1, w2, w3)
regpre  == case W of (w3, w0, w1, w2)
regx    == case X of (0, w1, w2, w3)
dregw   == regpre con regw
```

References to subfields of status:

```
monmode    == status (0)
escmode    == status (1)
after am   == status (2)
after esc  == status (3)
integer exception active        == status (4)
floating point exception active == status (5)
escapemask (0:5)                == status (6:11)
dumperrorcount (0:3)            == status (12:15)
ex (21:23)                      == status (21:23)
```

A register array notation is used in the operand access procedures in section 10.3 and in the algorithms for dump and restore of registers in sections 10.7 and 10.8.

```
reg (addr) == case addr // 2 of
                  (w0, w1, w2, w3, status, ic, cause, addr,
                   cpa, base, lowlim, uplim, intlim)
```

10.3 Common Routines

The procedures in this section describe the fetching of operands from the various areas in logical address space and the storing of results. A few other common actions are described at the end of the section.

All operand procedures will activate the program exception action, section 10.8, on protection violation. The contents of the resulting register dump may be reproduced by following the algorithm leading to the call of the action. This is usually but not necessarily true if the internal interrupt implicit in the "operand error" action, section 10.7, is activated. Minor deviations may occur, because the strict sequential description used in this chapter do not allow for parallel or look-ahead fetching of operands in the faster models. Buserrors at instruction fetch in jumps are classified as operand errors or fetcherrors depending on which classification gives the most efficient implementation of the normal case.

10.3.1 procedure getword;

comment:

This procedure assigns the contents of the word pointed out by the logical address in the "addr" register to the auxiliary register "opq". If the address is non-accessible control is transferred to the "program exception" action, section 10.8.

If a memory or buserror occurs, "operand error" interrupt is invoced, section 10.7;

```
begin
  if lowlim <= addr + base < uplim then
    begin
      opq:= word (addr + base);
      if buserror then goto operanderror
    end
  else
    if 8 <= addr <cpa then
      begin
      opq:= word (addr);
      if buserror then goto operanderror
    end
  else
    if 0 <= addr < 8 then opq:= reg (addr)
    else goto program exception
end;
```

10.3.2 procedure getdouble1;

```
comment:
  This procedure fetches the least significant word of a double
  word operand and assigns the contents to the auxiliary register
  "opq" just like "getword". In addition it prepares the tests in
  "getdouble2" which ensures that the whole doubleword is in one
  subarea of the address space, see "getdouble2" below;

begin
  if lowlim <= addr + base < uplim then
    begin
      opq:= word (addr + base)
      if buserror then goto operand error;
      procareaq:= true
    end
  else
    if 8 <= addr <cpa then
      begin
        opq:= word (addr);
        if buserror then goto operand error;
        cpaareaq:= true
      end
    else
      if 0 <= addr < 8 then
        begin opq:= reg (addr); regareaq:= true end
      else goto program exception
end;
```

10.3.3 procedure getdouble2;

```
comment:
  This procedure fetches the preoperand i.e. the most significant
  word of a doubleword operand and assigns the contents to the
  auxiliary register "oplq". The preoperand must be in the same
  area as the least significant word, else control is transferred
  to the program exception action, section 10.8. Note that the
  "addr" register points to the preoperand at exit;

begin
  addr:= addr-2;
  if procareaq then
    begin
      procareaq:= false;
      if addr + base <lowlim then
        goto program exception;
      oplq:= word (addr+ base);
      if buserror the goto operand error
    end;
  if cpaareaq then
    begin
      cpaareaq:= false;
      if addr < 8 then goto program exception;
      oplq:= word (addr);
      if buserror then goto operand error
    end;
  if regareaq then
    begin
      regareaq:= false;
      if addr < 0 then addr:= 6;
      comment or the equivalent : addr:= addr + 8;
      oplq:= reg (addr)
    end;
end;
```

10.3.4 procedure get nonprotected;

```
comment:
  This procedure is used to fetch operands in instructions, which
  store the result at the same address. Reference outside the
  process area and the register area transfers control to the
  program exception action, section 10.8;
```

```
begin
  if lowlim <= addr + base < uplim then
    begin
      opq:= word (addr + base);
      if buserror then goto operand error
    end
  else
    if 0 <= addr < 8 then opq:= reg(addr)
    else goto program exception
end;
```

10.3.5 procedure get address;

comment:
This procedure is used in indirect address calculation. It performs the escape test and may activate "escape", section 10.8. The address is fetched by a call of "get word". Note the assignments of the "after am" bit in the status register and the special case for model RC 8000/45;

```
begin
  after am:= true; comment not in RC 8000/45;
  if escmode then
    begin
    comment after am:= true in RC 8000/45;
    instrmask:= 0 con 2.1000000000010;
    if instrmask and (0 con escapemask) <> 0 then
      goto escape
    end;
  comment after am:= false in RC 8000/45;
  getword;
  addr:= opq;
  after am:= false
end;
```

10.3.6 procedure get instruction;

comment:
This procedure is used in jumps. It is analogous to "get word", but it has some additional actions in connection with the control of instruction flow, section 10.4.

```
begin
  if addr(23)=1 then addr:= addr-1;
  if lowlim <= addr + base < uplim then
    begin
      aq:= addr + base;
      goto ifetch
    end;
```

```
      if 8 <= addr < cpa then
        begin
          aq:= addr;
          goto ifetch
        end;
      if 0 <= addr < 8 then
        begin
          instruction:= reg (addr);
          pic:= addr (1:23);
          goto after ifetch
        end
      else goto program exception;

    ifetch:
      instruction:= word (aq);
      if buserror then goto operand error;
      comment or:
        begin
          if W <> 0 then rewq:= ic;
          ic:= addr;
          goto fetcherror
        end;
      pic:= aq (1:23);
    after ifetch:
      fetchedq:= true
  end;
```

10.3.7 procedure store word;

```
  comment:
    This procedure stores the contents of the auxiliary register
    "opq" in the word pointed out by the logical address in the
    "addr" register. Reference outside the process area and the
    register area will transfer control to the "program exception"
    action section 10.8;

  begin
    if lowlim <= addr + base < uplim then
      begin
        word (addr + base):= opq;
        if buserror then goto operand error
      end
    else
      if 0 <= addr < 8 then reg(addr):= opq
      else goto program exception
  end;
```

10.3.8 procedure set bus exceptions;

```
comment:
  This procedure assigns "ex", the exception register, according
  to busstatus. It is called after a buserror and from the "data
  in" and "data out" instructions;

begin
  ex(21:23):= 0;
  if bus parity error then ex(21):= 1;
  if bus time out then ex(22):= 1;
  if bus communication error then ex(23):= 1
end;
```

10.3.9 normalize and round;

```
comment:
  This common exit from the floating point instructions re-
  ceives the result as a mantissa in the auxiliary register
  "mantq(0:<38)" and an exponent in the auxiliary register
  "expq". It performs a normalization followed by an addition of
  a bit in position 36 to round the result. After a possible
  renormalization the result is packed into the double register
  specified by the W-field. Exponent overflow is indicated in
  "ex", the exception register, and "floating point exception",
  section 10.8, may be activated. Normal continuation is at "next
  instruction", section 4;

ex(22:23):= 0;
if mantq (0:>38)= 0 then
  begin
    dregw:= 0 con -2048 (12);
    goto next instruction
  end;
while mantq (0) = mantq (1) do
  begin
    mantq (0:>38):= mantq (1:>38) con 0;
    expq:= expq - 1
  end;
sq:= mantq (0);
mantq (0:>38):= mantq (0:>38) + 0(36) con 1(1) con 0;
mantq (0:>38):= sq con mantq (0:>38-1);
expq:= expq + 1;
while mantq(0) = mantq(1) do
  begin
    mantq (0:>38):= mantq (1:>38) con 0;
    expq:= expq - 1
end;
```

```
dregw:= mantq (0:35) con exp (12:23);
if exp (0:11) <> signextend expq (12) then
  begin
    if expq > 0 then ex(22):= 1 else ex (23):= 1;
    if floating point exception active then
      goto floating point exception
  end;
goto next instruction;
```

10.3.10 exit to program:

```
comment:
  This algorithm describes the common exit from the "escape" and
  "exception" actions and the "return from interrupt" and "return
  from escape" instructions, sections 10.7 and 10.8. The instruc-
  tion pointed out by the logical address in "ic" is fetched and
  control is transferred to "next instruction", section 10.4,
  with "fetchedq = true". "Program exception" or "fetcherror"/
  "operand error" may be activated;

if ic(23) = 1 then ic:= ic - 1;
if lowlim <= ic + base < uplim then
  begin
    iq:= ic + base;
    goto exitf
  end;
if 8 <= ic < cpa then
  begin
    iq:= ic;
    goto exitf
  end;
if 0 <= ic < 8 then
  begin
    instruction:= reg (ic);
    pic:= ic (1:23);
    goto after exitf
  end
else begin ic:= oldicq; goto program exception end;

exitf:
instruction:= word (iq);
if buserror then goto fetcherror;
comment or:
  begin
    ic:= oldicq;
    addr:= aq;
    goto operand error
  end;
```

```
pic:= iq (1:23);
after exitf:
fetchedq:= true;
goto next instruction;
```

10.4 Instruction Control

The normal sequential instruction flow is controlled by the physical instruction counter, "pic". "Pic" is a 23 bit register containing a positive physical address. "Pic" is increased by 2 for each instruction. "Ic", the logical instruction counter is increased by 2 at the end of address calculation.

Active skips cause an additional increase of "pic" and "ic". All other deviations from the sequential flow i.e. jumps, monitor calls, interrupts, exceptions, escapes, "return from interrupt" and "return from escape" assigns both "pic" and "ic" and perform relocation and test for protection violation.

This means that sequential programs including skips may cross the upper limits of subareas in the address space undetected. However if return to the sequence after a deviation in program flow is performed in this case, the situation will be recognized and may result in relocation or program exception. The inconvenience to the user in diagnosis of this very special class of programming errors is justified by the speed up of instruction flow and skips.

After test of the "interrupt flag", see section 6.6, which may lead to the activation of the external interrupt action, section 10.7, the instruction control proceeds as follows:

If the instruction is not already fetched by a preceeding action, pic is increased by 2 and the instruction is fetched. A buserror in fetch activates the "fetcherror" action, section 10.7.

After fetch the instruction is unpacked and the address calculation is performed, either by the special "after escape" action, or as one of the normal address calculation actions in section 10.5. The action is selected by the values of "after am" and the M-field of the instruction. All address calculation actions increase ic by 2 and transfer control to the execute action. Indirect addressing may however activate "escape", "program exception" and "operand error" actions.

The "execute" action tests whether the "escape" action, section 10.8, should be activated. If this is not the case, one of the instruction actions in section 10.6 is selected by the F-field. These actions may result in activation of "exception", "monitorcall" or "operand error" actions.

10.4.1 next instruction:

```
if interrupt flag then goto external interrupt;
if fetchedq then fetchedq:= false
else
  begin
    pic:= pic + 2;
    if 0 <= pic < 8 then instruction:= reg (pic)
    else
      begin
        instruction:= word (pic);
        if buserror then goto fetcherror
      end
  end;

F:= instruction  (0:5);
W:= instruction  (6:7);
M:= instruction  (8:9);
X:= instruction (10:11);
D:= signextend instruction (12:23);

if after esc then
  begin
    if after am then
      begin
        after esc:= false;
        comment in RC 8000/45: after am:= false;
        get word;
        addr:= opq;
        after am:= false
      end
    else if -, escmode then after esc:= false;
    ic:= ic + 2;
    goto execute
  end;
```

```
aswitchq:= after am con M;
goto case aswitchq of
  ( direct,   indirect,   relative,   relative indirect,
   amdirect, amindirect, amrelative, amrelative indirect);
execute:
if escmode then
  begin if after esc then after esc:= false end
  else
    begin
      instrmask:= F con case F of

        ( 2.100000,   2.100001,  2.000010,   2.000010,
          2.000010,   2.000010,  2.000010,   2.000010,
          2.000010,   2.000001,  2.000010,   2.000001,
          2.111010,   2.010000,  2.110000,   2.110000,
          2.000010,   2.000010,  2.000010,   2.000010,
          2.000010,   2.010001,  2.011010,   2.000100,
          2.000010,   2.000110,  2.000100,   2.000100,
          2.000001,   2.100001,  2.100000,   2.100000,
          2.000001,   2.000001,  2.000100,   2.000100,
          2.000001,   2.000001,  2.000001,   2.000001,
          2.010001,   2.010001,  2.010001,   2.010001,
          2.010001,   2.010001,  2.010001,   2.100001,
          2.001010,   2.001010,  2.001010,   2.100000,
          2.001010,   2.000001,  2.001010,   2.001100,
          2.001010,   2.001010,  2.100000,   2.100000,
          2.100000,   2.100000,  2.100000,   2.100000);

  comment the escape patterns are also given for each instruction
  in section 10.6 and in appendix 1-3;

  if instrmask and 0 con escapemask <> 0 then
    goto escape
  end
end;
goto case F of
 (u0, do, el, hl, la, lo, lx, wa,
  ws, am, wm, al, ri, jl, jd, je,
  xl, es, ea, zl, rl, sp, re, rs,
  wd, rx, hs, xs, gg, di, u30, u31,
  ci, ac, ns, nd, as, ad, ls, ld,
  sh, sl, se, sn, so, sz, sx, gp,
  fa, fs, fm, u51, fd, cf, dl, ds,
  aa, ss, u58, u59, u60, u61, u62, u63);
```

10.5 Address Calculation

This section describes the normal address calculation actions. The actions are entered from the instruction control in section 10.4 and return to the "execute" action.

"Operand error", "escape" and "program exception", sections 10.7 and 10.8, may be activated by the indirect addressing actions through the call of "get address", section 10.3.

```
direct:
  addr:= regx + D;
  ic:= ic + 2;
  goto execute;

indirect:
  addr:= regx + D;
  ic:= ic + 2;
  get address;
  goto execute;

relative:
  addr:= ic + regx + D;
  ic:= ic + 2;
  goto execute;

relative indirect:
  addr:= ic + regx + D;
  ic:= ic + 2;
  get address;
  goto execute;

am direct:
  addr:= addr + regx + D;
  after am:= false;
  ic:= ic + 2;
  goto execute;

am indirect:
  addr:= addr + regx + D;
  ic:= ic + 2;
  get address;
  goto execute;
```

```
am relative:
  addr:= addr + ic + regx + D;
  after am:= false;
  ic:= ic + 2;
  goto execute;

am relative indirect:
  addr:= addr + ic + regx + D;
  ic:= ic + 2;
  get address;
  goto execute;
```

10.6 Instruction Execution

The instruction actions are entered from section 10.4, after address calculation, section 10.5, has been performed. Normal return is to "next instruction", section 10.4, but "operand error", "monitor call" and "exception", sections 10.7 and 10.8, may be activated.

The instructions are described in alphabetical order by mnemonics. The return from interrupt, ri, and return from escape, re, instructions are described in section 10.7 and 10.8 respectively. Floating point instructions return to "next instruction" through the common exit, "normalize and round", in section 10.3.

10.6.1 aa:

comment:
"Add double word to double register",
adds the contents of the doubleword pointed out by the effective address to the specified doubleregister. If the effective address is 0 throgh 7, the operation is a double register-double register operation. In this case it is important to remember that the operation on the least significant registers is completed before the operation on the most significant registers and carry is executed. Reference to non-accessible area and reference to the lower limits of process area (addr = lowlim - base) and common protected area (addr = 8) leads to program exception. Overflow and carry is registered in the exception register and integer exception may occur.
Numeric code: 56
Escape pattern: 2.001010;

```
get double1;
regw:= regw + opq;
get double2; comment does not change carry;
regpre:= regpre + oplq + carry;
ex(22):= overflow;
ex(23):= carry;
if overflow and integer exception active then
  goto integer exception; comment "addr" points to preoperand;
goto next instruction;
```

10.6.2 ac:

```
comment:
 "Address complemented: load into register", assigns the two's
  complement of the effective address to the specified register.
  Complementation of the maximum negative number will produce
  overflow. Complementation of a non-zero number produces a
  carry.
  Numeric code:   33
  Escape pattern: 2.000001;
regw:= 0 - addr;
ex(22):= overflow;
ex(23):= carry;
if overflow and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.3 ad:

```
comment:
 "Arithmetic shift of double register", shifts the contents of
  the specified double register the number of places given by the
  effective address. If the effective address is negative the
  shift is a right shift with signextension, else it is a left
  shift with zeros shifted in at the least significant places.
  Left shifts produce integer overflow if one or more significant
  bits are lost.
  Numeric code:   37
  Escape pattern: 2.000001;
ex(22:23):= 0;
if abs(addr) > 48 then addr:= sign(addr) * 48;
if addr = 0 then goto next instruction;
if addr < 0 then
  dregw:= signextend dregw (0:47+addr);
```

```
else
  begin
    if dregw (0:addr) <> signextend dregw (addr) then
      ex(22):= 1;
    dregw:= dregw (addr: 47 + addr);
    addr:= -addr
  end;
if ex(22) and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.4 al:

comment:
"Address : load into register", assigns the effective address to the specified register.
Numeric code: 11
Escape pattern: 2.000001;

```
regw:= addr;
goto next instruction;
```

10.6.5 am:

comment:
"Address : modify that of next instruction", sets the "after am" bit in the "status" register. The effect is described in sections 10.4 and 10.5.
Numeric code: 9
Escape pattern: 2.000001;

```
after am:= true;
goto next instruction;
```

10.6.6 as:

comment:
"Arithmetic shift of single register", shifts the contents of the specified register the number of places given by the effective address. If the effective address is negative the shift is a right shift with signextension, else it is a left shift with zeros shifted in at the least significant places. Left shifts produce overflow if one or more significant digits are lost.
Numeric code: 36
Escape pattern: 2.000001;

```
ex(22:23):= 0;
if abs(addr) > 48 then addr:= sign(addr) * 48;
if addr = 0 then goto next instruction;
if addr < 0 then
  regw:= signextend regw (0 : 23 + addr);
```

```
else
  begin
    if regw (0:addr) <> signextend regw (addr) then
      ex(22):= 1;
    regw:= regw (addr : 23 + addr);
    addr:= -addr
  end;
if ex(22) and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.7 cf:

comment:
"Convert floating point to integer", converts the floating point value of the specified double register to an integer and places the result in the least significant register. The effective address is used for scaling, such that the value of the result is two to the power of effective address times the floating point number rounded to nearest integer. Carry = 0 and integer overflow is registered in the exception register. Integer overflow may lead to integer exception.
Numeric code: 53
Escape pattern: 2.000001;

```
ex(22:23):= 0;
expq:= signextend dregw (36:47) + addr;
if expq < 0 then
  begin
    dregw (24:47):= 0;
    goto next instruction
  end;
if expq > 23 then
  begin comment non-normalize numbers may be considered zero;
    if dreg (0 : > 1) = 0 then
      dregw (24:47) = 0
    else
      begin
        ex(22):= 1;
        if integer exception active then
          goto integer exception
      end;
    goto next instruction
  end;
  roundq:= dregw (24);
  dregw (24:47):= dregw (0:23);
  for expq:= expq -23 step 1 until -1 do
```

```
    begin
      roundq:= dregw (47);
      dregw (24:47):= dregw (24) con dregw (24:46)
    end;
dregw (24:47):= dregw (24:47) + 0 con roundq;
ex(22):= overflow;
if overflow and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.8 ci:

comment:
"Convert integer to floating point", converts the integer value of the specified register to a floating point number and places the result in the corresponding double register. The effective address is used for scaling, such that the value of the result is two to the power of effective address times the integer. Floating point overflow and underflow may occur through scaling and is handled as for the floating point instructions by normalize and round, section 10.3.9.
Numeric code: 32
Escape pattern: 2.000001;

```
expq:= addr + 23;
mantq (0 : > 38):= regw (0 : > 38);
goto normalize and round;
```

10.6.9 di:

comment:
"Data in", receives data from a device address on the unified bus in the specified register. See furher section 8.2. "Di" is a privileged instruction.
Numeric code: 29
Escape pattern: 2.100001;

```
if -, monmode then goto program exception;
regw:= word (addr);
set bus exceptions; comment section 10.3.8;
goto next instruction;
```

10.6.10 dl:

comment:
"Double register: load", loads the specified double register with the contents of the doubleword pointed out by the effective address. If the effective address is 0 throgh 7, "dl" is a double register to double register transfer; in this case it is important to note that the least significant register is

```
    assigned before the most significant register is transferred.
    Reference to the nonaccessible area and to the lower limits of
    the process area (addr = lowlim - base) and the common protec-
    ted area (addr = 8) will lead to program exception.
    Numeric code:   54
    Escape pattern: 2.001010;
  getdouble1;
  regw:= opq;
  getdouble2;
  regpre: op1q;
  goto next instruction;
```

10.6.11 do:

```
  comment:
   "Data out", transfers the contents of the specified register to
    a device address on the unified bus. See further section 8.2.
    "Do" is a privileged instruction.
    Numeric code:   1
    Escape pattern: 2.100001;
  if -, monmode then goto program exception;
  word (addr):= regw;
  set bus exceptions; comment section 10.3.8;
  goto next instruction;
```

10.6.12 ds:

```
  comment:
   "Double register : store", stores the contents of the specified
    double register in the doubleword pointed out by the effective
    address. If the effective address is 0 through 7, "ds" is a
    double register to double register transfer; note that the
    least significant register is assigned before the most signifi-
    cant register is transferred. Other references outside the pro-
    cess area and reference to the lower limit of the process area
    will lead to program exception.
    Numeric code:   55
    Escape pattern: 2.001100;
  if lowlim <= addr + base < uplim then
    begin
      word (addr + base):= regw;
      if buserror then goto operand error;
      addr:= addr - 2;
      if addr + base < lowlim then
        goto program exception;
      word (addr + base):= regpre;
      if buserror then goto operand error;
```

```
    goto next instruction
  end;
if 0 <= addr < 8 then
  begin
    reg (addr):= regw;
    addr:= addr - 2;
    if addr < 0 then addr:= 6;
    reg (addr):= regpre
  end
else goto program exception;
goto next instruction;
```

10.6.13 ea:

comment:
"Extended halfword: add to register", adds the contents of the halfword pointed out by the effective address to the specified register after sign extension. If the effective address is 0 through 7 the operation is a register-halfregister operation. Reference to non-accessible area leads to program exception. Overflow and carry is registered in the exception register and integer exception may occur.
Numeric code: 18
Escape pattern: 2.000010;

```
getword;
if addr(23) = 0 then
  opq:= signextend opq (0:11)
else
  opq:= signextend opq (12:23);
regw:= regw + opq;
ex(22):= overflow;
ex(23):= carry;
if overflow and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.14 el:

comment:
"Extended halfword: load into register", loads the least significant 12 bits of the specified register with the contents of the halfword pointed out by the effective address. The most significant 12 bits will contain a signextension. If the effective address is 0 throgh 7, "el" is a half register to register transfer. Reference to non-accessible area leads to program exception.

```
  Numeric code:   2
  Escape pattern: 2.000010;
get word;
if addr(23)= 0 then
  regw:= signextend opq (0:11)
else
  regw:= signextend opq(12:23);
goto next instruction;
```

10.6.15 es:

comment:

"Extended halfword: subtract from register", subtracts the contents of the halfword pointed out by the effective address from the specified register after sign extension. If the effective address is 0 throgh 7 the operation is a register-halfregister operation. Reference to non-accessible area leads to program exception. Overflow and carry is registered in the exception register and integer exception may occur.

```
  Numeric code:   17
  Escape pattern: 2.000010;
get word;
if addr(23)= 0 then
  opq:= signextend opq(0:11)
else
  opq:= signextend opq(12:23);
regw:= regw - opq;
ex(22):= overflow;
ex(23):= carry;
if overflow and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.16 fa:

comment:

"Floating point: add to double register", adds the contents of the doubleword pointed out by the effective address to the specified double register. If the effective address is 0 through 7, the operation is a double register-double register operation. Reference to non-accessible area and to the lower limits of the process area (addr = lowlim - base) and the common protected area (addr = 8) leads to program exception. Underflow and overflow conditions are indicated in the exception register. If the "floating point exception active" bit is set in the "status" register, underflow and overflow will lead to a floating point exception. After underflow and overflow the fraction of the result is correct while the exponent is taken modulo 4096.

```
    Some rules concerning the result of operations on non-nor-
    malized numbers may be extracted from the algorithm.
    Numeric code:   48
    Escape pattern: 2.001010;
  getdouble1;
  getdouble2; comment "addr" now points to preoperand;
  mantq (0 : > 38):= dregw (0:35) con 0;
  expq:= signextend dregw (36:47);
  mantlq (0 : > 38):= oplq con opq(0:11) con 0;
  explq:= signextend opq(12:23);
  expq:= expq - explq;
  if expq <= -38 then
    begin comment register operand irrelevant;
      mantq:= 0;
      expq:= explq;
      goto add
    end;
    comment or:
      begin
        ex(22:23):= 0;
        dregw:= mantlq (0:35) con explq (12:23);
        goto next instruction
      end;
  if expq >= 38 then
    begin comment "memory" operand irrelevant;
      mantlq:= 0;
      expq:= signextend dregw (36:47);
      goto add
    end;
    comment or:
      begin
        ex(22:23):= 0;
        goto next instruction
      end;
  if expq < 0 then
    begin
      mantq (0 : > 38):= signextend mantq (0 : > 38 + expq);
      expq:= explq
    end
  else
    begin
      mantlq (0 : > 38):= signextend mantlq (0 : > 38 - expq);
      expq:= signextend dregw (36:47)
    end;
```

```
add:
mantq:= mantq + mantlq;
sq:= if overflow then -, mantq(0) else mantq(0);
mantq:= sq con mantq (0 : > 38-1);
expq:= expq + 1;
goto normalize and round; comment section 10.3.9;
```

10.6.17 fd:

```
comment:
 "Floating point: divide into double register", divides the spe-
  cified double register with the contents of the double word
  pointed out by the effective address. If the effective address
  is 0 through 7, the operation is a double register-double re-
  gister operation. Reference to non-accessible area and to the
  lower limits of the process area (addr = lowlim - base) and the
  common protected area (addr = 8) leads to program exception.
  Underflow and overflow conditions are indicated in the excep-
  tion register. If the "floating point esception active" bit is
  set in the status register, underflow and overflow will lead to
  a floating point exception. After underflow and overflow the
  fraction of the result is correct while the exponent is taken
  modulo 4096. The result of operation on non-normalize operands
  must be regarded as undefined.
  Numeric code:   52
  Escape pattern: 2.001010;
getdouble1;
getdouble2; comment "addr" now points to preoperand;
mantq (0 : > 38):= dregw (0:35) con 0;
expq:= signextend dregw (36:47);
mantlq (0 : > 38):= oplq con opq (0:11) con 0;
explq:= signextend opq (12:23);
if mantlq (0 : > 1) = 0 then
  begin comment zero divisor, the zero check may be true for
              various non-normalized numbers;
    ex(22):= 1;
    ex(23):= 0;
    if floating point exception active then
      goto floating point exception;
    goto next instruction
  end;
if mantq (0 : > 1) = 0 then
  begin comment zero dividend;
    ex(22:23):= 0;
    dregw:= 0(36) con -2048(12);
    goto next instruction
  end;
```

```
expq: expq - explq;
mantq:= mantq divided by mantlq;
adjust expq;
comment details are model and option dependent;
goto normalize and round; comment section 10.3.9;
```

10.6.18 fm:

```
comment:
 "Floating point: multiply by double register", multiplies the
  specified double register by the contents of the double word
  pointed out by the effective address. If the effective address
  is 0 through 7, the operation is a double register-double re-
  gister operation. Reference to non-accessible area and to the
  lower limits of the process area (addr = lowlim - base) and the
  common protected area (addr = 8) leads to program exception.
  Underflow and overflow conditions are indicated in the except-
  tion register. If the "floating point exception active" bit is
  set in the status register, underflow and overflow will lead to
  a floating point exception. After underflow and overflow the
  fraction of the result is correct while the exponent is taken
  modulo 4096. The result of operation on non-normalized operands
  must be regarded as undefined.
  Numeric code:   50
  Escape pattern: 2.001010;
getdouble1;
getdouble2; comment addr now points to preoperand;
mantq (0 : > 38>= dregw (0:35) con 0;
expq:= signextend dregw (36:47);
mantlq (0 : > 38):= oplq con opq (0:11) con 0;
explq:= signextend opq (12:23);
expq:= expq + explq;
mantq:= mantq multiplied by mantlq;
adjust expq;
comment details are model and option dependent;
goto normalize and round; comment section 10.3.9;
```

10.6.19 fs:

```
comment
 "Floating point: subtract from double register", subtracts the
  contents of the doubleword pointed out by the effective address
  from the specified double register. If the effective address is
  0 through 7, the operation is a double register-double register
  operation. Reference to the non-accessible area and to the
  lower limits of the process area (addr = lowlim - base) and the
```

```
   common protected area (addr = 8) leads to program exception.
   Underflow and overflow conditions are indicated in the excep-
   tion register. If the "floating point exception active" bit is
   set in the status register, underflow and overflow will lead to
   a floating point exception. After underflow and overflow the
   fraction of the result is correct while the exponent is taken
   modulo 4096. Some rules concerning the results of operations on
   non-normalized numbers may be extracted from the algorithm.
   Numeric code:   49
   Escape pattern: 2.001010;
getdouble1;
getdouble2; comment addr now points to preoperand;
mantq (0 : > 38):= dregw (0:35) con 0;
expq:= signextend dregw (36:47);
mantlq (0 : > 38):= oplq con opq (0:11) con 0;
explq:= signextend opq (12:23);
expq:= expq - explq;
if expq <= -38 then
  begin comment register operand irrelevant;
    mantq:= 0;
    expq:= explq;
    goto sub
  end;
if expq >= 38 then
  begin comment "memory" operand irrelevant;
    mantlq:= 0;
    expq:= signextend dregw (36:47);
    goto sub
  end;
  comment or:
    begin
      ex(22:23):= 0;
      goto next instruction
    end;
if expq < 0 then
  begin
    mantq (0 : > 38):= signextend mantq (0 : > 38 + expq);
    expq:= explq;
  end
else
  begin
    mantlq (0 : > 38):= signextend mantlq (0 : > 38 - expq);
    expq:= signextend dregw (36:47)
  end;
```

```
sub:
mantq:= mantq - mantlq;
sq:= if overflow then -, mantq(0) else mantq(0);
mantq:= sq con mantq (0 : > 38 - 1);
expq:= expq + 1;
goto normalize and round; comment section 10.3.9;
```

10.6.20 gg:

```
comment:
 "General get from processor register", assigns the contents of
  the processor register pointed out by the effective address to
  the specified register. "gg" gives access to the monitor regis-
  ters "inf", "size", "montop" and "rtc". Further effective ad-
  dresses may give access to other registers in specific models.
  If the effective address do not point to a processor register
  "gg" is a dummy instruction.
  Numeric code:   28
  Escape patter: 2.000001;
if addr = 26  then regw:= inf;
if addr = 30  then regw:= size;
if addr = 32  then regw:= montop;
if addr = 100 then regw:= 0 con rtc;
if addr = "other" then regw:= "other register";
goto next instruction;
```

10.6.21 gp:

```
comment:
 "General put into processor register", assigns the contents of
  the specified register to the processor register pointed out by
  the effective address. "gp" gives access to the monitor regis-
  ters "inf", "size" and "montop" and allows the clearing of bits
  in the interrupt register "intreg". Further effective addresses
  may give access to other registers in specific models. If the
  effective address do not point to a processor register "gp" is
  a dummy instruction. "gp" is a privileged instruction.
  Numeric code:   47
  Escape pattern: 2.100001;
if -, monmode then goto program exception;
if addr = 26 then inf:= regw;
if addr = 30 then size:= regw;
if addr = 32 then montop:= regw;
if addr = 94 then intreg (regw):= 0;
comment only used in special cases as after
  power restart. Normal clearing of interrupt bits is performed
  by the interrupt logic.
  The addressing of bits is modulo a model dependent power of
  two;
```

```
if addr = "other" then "other register":= regw;
goto next instruction;
```

10.6.22 hl:

```
comment:
 "Half register: load", loads the 12 least significant bits of
  the specified register with the contents of the halfword poin-
  ted out by the effective address. If the effective address is 0
  through 7 "hl" is a half register to register transfer. Refe-
  rence to the non-accessible area leads to program exception.
  Numeric code:   3
  Escape pattern: 2.000010;
getword;
if addr(23) = 0 then
  regw:= regw (0:11) con opq (0:11)
else
  regw:= regw (0:11) con opq (12:23);
goto next instruction;
```

10.6.23 hs:

```
comment:
 "Half register: store", stores the least significant 12 bits of
  the specified register in the halfword pointed out by the ef-
  fective address. If the effective address is 0 through 7, "hs"
  is a register to half register transfer. Other references out-
  side the process area leads to program exception.
  Numeric code:   26
  Escape pattern: 2.000100;
get nonprotected;
if addr(23) = 0 then
  opq (0:11):= regw (12:23)
else
  opq (12:23):= regw (12:23);
store word;
goto next instruction;
```

10.6.24 jd:

```
comment:
 "Jump and select disable limit", sets the "disable" bit in the
  status register to one, and transfers control to the instruc-
  tion pointed out by the effective address. If the W-field is
  non-zero the link, i.e. the logical address of the next in-
  struction, is assigned to the specified register. Reference to
  the non-accessible area leads to program exception, unless the
  effective address is between -2048 and -"montop". In this case
```

```
    "jd" is a monitor call with cause = addr + 2048.
    Numeric code:   14
    Escape pattern: 2.110000 (in RC 8000/45: 2.100000);
  if -2048 <= addr < -montop then
    begin
      cause:= addr + 2048;
      goto call; comment section 10.7.4
    end;
  comment in RC 8000/45: disable:= true;
  get instruction;
  disable:= true;
  if W <> 0 then regw:= ic;
  ic:= addr;
  goto next instruction;
```

10.6.25 je:

```
  comment:
    "Jump and select enable limit", sets the "enable" bit in the
    "status" register to zero, and transfers control to the in-
    struction pointed out by the effective address. If the W-field
    is non-zero the link, i.e. the logical address of the next in-
    struction, is assigned to the specified register. Reference to
    the non-accessible area leads to program exception.
    Numeric code:   15
    Escape pattern: 2.110000 (in RC 8000/45: 2.100000);
  comment in RC 8000/45: disable:= false;
  get instruction;
  disable:= false;
  if W <> 0 then regw:= ic;
  ic:= addr;
  goto next instruction;
```

10.6.26 jl:

```
  comment:
    "Jump with link in register", transfers control to the instruc-
    tion pointed out by the effective address. If the W-field is
    non-zero the link, i.e. the logical address of the next in-
    struction, is assigned to the specified register. Reference to
    the non-accessible area leads to program exception.
    Numeric code:   13
    Escape pattern: 2.010000;
  get instruction;
  if W <> 0 then regw:= ic;
  ic:= addr;
  goto next instruction;
```

10.6.27 la:

```
comment:
  "Logical and: combine word with register", combines the con-
  tents of the word pointed out by the effective address with
  the contents of the specified register by a logical and opera-
  tion. The result is assigned to the specified register. If the
  effective address is 0 through 7 the result is a combinition of
  two registers. Reference to the non-accessible area leads to
  program exception.
  Numeric code:   4
  Escape pattern: 2.000010;
getword;
regw:= regw and opq;
goto next instruction;
```

10.6.28 ld:

```
comment:
  "Logical shift of double register", shifts the contents of the
  specified double register the number of places given by the ef-
  fective address. If the effective address is negative the shift
  is a right shift with zeros shifted in at the most significant
  bits, else it is a left shift with zeros shifted in at the
  least significant bits.
  Numeric codes:   39
  Escape pattern: 2.000001;
if abs(addr) > 48 then
  begin comment fast clear of a double register;
    dregw:= 0;
    goto next instruction
  end;
dregw:= dregw (addr: 47 + addr);
goto next instruction;
```

10.6.29 lo:

```
comment:
  "Logical or: combine word with register", combines the contents
  of the word pointed out by the effective address with the con-
  tents of the specified register by a logical or operation. The
  result is assigned to the specified register. If the effective
  address is 0 through 7 the result is a combinition of two re-
  gisters. Reference to the non-accessible area leads to program
  exception.
  Numeric code:   5
  Escape pattern: 2.000010;
```

```
getword;
regw:= regw or opq;
goto next instruction;
```

10.6.30 ls:

```
comment:
  "Logical shift of single register", shifts the contents of the
  specified register the number of places given by the effective
  address. If the effective address is negative the shift is a
  right shift with zeros shifted in at the most significant bits,
  else it is a left shift with zeros shifted in at the least sig-
  nificant bits.
  Numeric code:   38
  Escape pattern: 2.000001;
if abs(addr) > 48 then addr:= sign(addr) * 48;
regw:= regw (addr: 23 + addr);
goto next instruction;
```

10.6.31 lx:

```
comment
  "Logical exclusive or: combine word with register", combines
  the contents of the word pointed out by the effective address
  with the contents of the specified register by a logical exclu-
  sive or operation. The result is assigned to the specified re-
  gister. If the effective address is 0 through 7 the result is a
  combinition of two registers. Reference to the non-accessible
  area leads to program exception.
  Numeric code:   6
  Escape pattern: 2.000010;
getword;
regw:= regw exclusive or opq;
goto next instruction;
```

10.6.32 nd:

```
comment:
  "Normalize double register", normalizes the contents of the
  specified double register by shifting it to the left until bit
  0 and bit 1 have opposite values. The negative number of shifts
  necessary is stored in the halfword pointed out by the effec-
  tive address. Normalization of zero will give zero and store
  the value -2048. Effective address 0 through 7 will cause the
  negative shift value to be assigned to the corresponding half
  register. Other references outside the process area will lead
  to program exception.
```

```
    Numeric code: 35
    Escape pattern: 2.000100;
  if dregw = 0 then
    shq:= -2048
  else
    begin
      shq:= 0;
      while dregw(0) = dregw(1) do
        begin
          dregw:= dregw (1:47) con 0;
          shq:= shq -1
        end
    end;
  get nonprotected;
  if addr(23) = 0 then
    opq (0:11):= shq (12:23)
  else
    opq (12:23):= shq (12:23);
  store word;
  goto next instruction;
```

10.6.33 ns:

```
  comment:
    "Normalize single register", normalizes the contents of the
    specified register by shifting it to the left until bit 0 and
    bit 1 have opposite values. The negative number of shifts ne-
    cessary is stored in the halfword pointed out by the effective
    address. Normalization of zero will give zero and store the
    value -2048. Effective address 0 through 7 will cause the ne-
    gative shift value to be assigned to the corresponding half
    register. Other references outside the process area will lead
    to program exception.
    Numeric code:   34
    Escape pattern: 2.000100;
  if regw = 0 then
    shq:= -2048
  else
    begin
      shq:= 0;
      while regw(0) = regw(1) do
        begin
          regw:= regw (1:23) con 0;
          shq:= shq - 1
        end
    end;
```

```
get nonprotected;
if addr (23) = 0 then
  opq (0:11):= shq (12:23)
else
  opq (12:23):= shq (12:23);
store word;
goto next instruction;
```

10.6.34 re:

```
comment:
  "Return from escape", is described in section 10.8.5;
goto re algorithm;
```

10.6.35 ri:

```
comment:
  "Return from interrupt", is described in section 10.7.5;
goto ri algorithm;
```

10.6.36 rl:

```
comment:
  "Register : load", loads the specified register with the con-
  tents of the memory word pointed out by the effective address.
  If the effective address is 0 through 7, "rl" is a register to
  register transfer. Reference to the non-accessible area leads
  to program exception.
  Numeric code:   20
  Escape pattern: 2.000010;
getword;
regw:= opq;
goto next instruction;
```

10.6.37 rs:

```
comment:
  "Register : store", stores the contents of the specified regis-
  ter in the memory word pointed out by the effective address. If
  the effective address is 0 through 7, "rs" is a register to re-
  gister transfer. Other reference outside the process area leads
  to program exception.
  Numeric code:   23
  Escape pattern: 2.000100;
opq:= regw;
store word;
goto next instruction;
```

10.6.38 rx:

```
comment:
  "Register : exchange with word", exchanges the contents of the
  specified register and the memory word pointed out by the ef-
  fective address. If the effective address is 0 through 7, "rx"
  exchanges the contents of two registers. Other reference out-
  side the process area leads to program exception.
  Numeric code:   25
  Escape pattern: 2.000110;
get nonprotected;
oplq:= opq;
opq:= regw;
store word;
regw:= oplq;
goto next instruction;
```

10.6.39 se:

```
comment:
  "Skip if register equal", skips the next instruction if the
  contents of the specified register is equal to the effective
  address.
  Numeric code:   42
  Escape pattern: 2.010001;
if regw = addr then
  begin
    pic:= pic + 2;
    ic:= ic + 2
  end;
goto next instruction;
```

10.6.40 sh:

```
comment:
  "Skip if register high", compares the contents of the specified
  register and the effective address as signed integers. If the
  register value is greater than the effective address the next
  instruction is skipped.
  Numeric code:   40
  Escape pattern: 2.010001;
if regw > addr then
  begin
    pic:= pic + 2;
    ic:= ic + 2
  end;
goto next instruction;
```

10.6.41 sl:

```
comment:
  "Skip if register low", compares the contents of the specified
  register and the effective address as signed integers. If the
  register value is less than the effective address the next in-
  struction is skipped.
  Numeric code:   41
  Escape pattern: 2.010001;
if regw < addr then
  begin
    pic:= pic + 2;
    ic:= ic + 2
  end;
goto next instruction;
```

10.6.42 sn:

```
comment:
  "Skip if register nonequal", skips the next instruction if the
  contents of the specified register and the effective address
  are not equal.
  Numeric code:   43
  Escape pattern: 2.010001;
if regw <> addr then
  begin
    pic:= pic + 2;
    ic:= ic + 2
  end;
goto next instruction;
```

10.6.43 so:

```
comment:
  "Skip if selected register bits all ones", uses the effective
  address as a mask to test selected bits in the specified regis-
  ter. If all bits in the register that correspond to ones in the
  effective address are one, the next instruction is skipped.
  Numeric code:   44
  Escape pattern: 2.010001;
if regw = regw or addr then
  begin
    pic:= pic + 2;
    ic:= ic + 2
  end;
goto next instruction;
```

10.6.44 sp:

```
comment:
  "Skip if word not protected", skips the next instruction if the
  effective address points to the process area or to a register,
  i.e. if storing in the word addressed is allowed.
  Numeric code:   44
  Escape pattern: 2.010001;
if lowlim <= addr + base < uplim
  or 0 <= addr < 8 then
  begin
    pic:= pic + 2;
    ic:= ic + 2
  end;
goto next instruction;
```

10.6.45 ss:

```
comment:
  "Subtract double word from double register", subtracts the con-
  tents of the doubleword pointed out by the effective address
  from the specified double register. If the effective address is
  0 through 7, the operation is a double register-double register
  operation. In this case it is important to remember that the
  operation on the least significant registers is completed be-
  fore the operation on the most significant registers and carry
  is executed. Reference to non-accessible area and reference to
  the lower limits of process area (addr = lowlim -  base) and
  common protected area (addr = 8) leads to program exception.
  Overflow and carry is registered in the exception register and
  integer exception may occur.
  Numeric code:   57
  Escape pattern: 2.001010;
get double1;
regw:= regw - opq;
get double2; comment does not change carry;
regpre:= regpre - oplq - 1 + carry;
ex(22):= overflow;
ex(23):= carry;
if overflow and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.46 sx:

```
comment:
  "Skip if selected exception bits all zeros", uses the 3 least
  significant bits of the effective address as a mask to test se-
```

```
    lected bits in the exception register. If all selected bits,
    i.e. the bits corresponding to ones in the effective address,
    are zero the next instruction is skipped.
    Numeric code:   46
    Escape pattern: 2.010001;
  if ex and addr (21:23) = 0 then
    begin
      pic:= pic + 2;
      ic:= ic + 2
    end;
  goto next instruction;
```

10.6.47 sz:

```
  comment:
    "Skip if selected register bits all zeros", uses the effec-
    tive address as a mask to test selected bits in the specified
    register. If all bits in the register, that correspond to ones
    in the effective address are zero, the next instruction is
    skipped.
    Numeric code:   45
    Escape pattern: 2.010001;
  if regw and addr = 0 then
    begin
      pic:= pic + 2;
      ic:= ic + 2
    end;
  goto next instruction;
```

10.6.48 u0: u30: u31: u51: u58: u59: u60: u61: u62: u63:

```
  comment:
    "Unassigned instruction codes", provoke a program exception.
    They are reserved for implementation of further instructions as
    options or in later models.
    Numeric codes:  0, 30, 31, 51, 58, 59, 60, 61, 62, 63
    Escape pattern: 2.100000;
  goto program exception;
```

10.6.49 wa:

```
  comment:
    "Word : add to register", adds the contents of the word pointed
    out by the effective address to the specified register. If the
    effective address is 0 through 7 the operation is a register-
    register operation. Reference to non-accessible area leads to
    program exception. Overflow and carry is registered in the ex-
    ception register and integer exception may occur.
```

```
  Numeric code:   7
  Escape pattern: 2.000010;
get word;
regw:= regw + opq;
ex(22):= overflow;
ex(23):= carry;
if overflow and integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.50 wd:

comment:

"Word: divide into double register", divides the specified double register by the contents of the word pointed out by the effective address. The quotient is placed in the least significant register while the remainder is placed in the most significant register. If the effective address is 0 through 7, the operation is a double register-register operation. Reference to the non-accessible area leads to program exception. Carry = 0 and overflow is registered in the exception register and integer exception may occur. At overflow the dividend is left unchanged. Implementation details of the division differ from model to model and are not shown.

```
  Numeric code:   24
  Escape pattern: 2.000010;
ex(22:23):= 0;
getword;
if opq = 0 then goto set overflow;
quotq:= dregw//opq;
if overflow then goto set overflow;
regpre:= dregw mod opq;
regw:= quotq;
goto next instruction;
set overflow:
ex(22):= 1;
if integer exception active then
  goto integer exception;
goto next instruction;
```

10.6.51 wm:

comment:

"Word : multiply by register giving double register", multiplies the contents of the specified register by the contents of the word pointed out by the effective address. The result is placed in the corresponding double register. If the effective

```
    address is 0 through 7, the operation is a register-register
    operation. Reference to non-accessible area leads to program
    exception. Exception register is unchanged and integer excep-
    tion can not occur. Implementation details of the multiplica-
    tion differ from model to model and are not shown.
    Numeric code:   10
    Escape pattern: 2.000010;
  get word;
  dregw:= regw * opq;
  goto next instruction;
```

10.6.52 ws:

```
  comment:
    "Word : subtract from register", subtracts the contents of the
    word pointed out by the effective address from the specified
    register. If the effective address is 0 through 7 the operation
    is a register- register operation. Reference to non-accessible
    area leads to program exception. Overflow and carry is regis-
    tered in the exception register and integer exception may
    occur.
    Numeric code:   8
    Escape pattern: 2.000010;
  get word;
  regw:= regw - opq;
  ex(22):= overflow;
  ex(23):= carry;
  if overflow and integer exception active then
    goto integer exception;
  goto next instruction;
```

10.6.53 xl:

```
  comment:
    "Exception register : load from halfword", loads the exception
    register with the 3 least significant bits of the halfword
    pointed out by the effective address. If the effective address
    is 0 through 7, "xl" is a halfregister to exception register
    transfer. Reference to non-accessible area leads to program
    exception.
    Numeric code:   16
    Escape pattern: 2.000010;
  get word;
  if addr(23) = 0 then
    ex(21:23):= opq(9:11)
  else
    ex(21:23):= opq(21:23);
  goto next instruction;
```

10.6.54 xs:

```
comment:
  "Exception register: store in halfword", stores the exception
  register in the 3 least significant bits of the halfword poin-
  ted out by the effective address. The most significant 9 bits
  are set to zero. If the effective address is 0 through 7, "xs"
  is an exception register to halfregister transfer. Other refe-
  rences outside the process area leads to program exception.
  Numeric code:   27
  Escape pattern: 2.000100;
get nonprotected;
if addr(23) = 0 then
  opq(0:11):= 0 con ex(21:23)
else
  opq(12:23):= 0 con ex(21:23);
store word;
goto next instruction;
```

10.6.55 zl:

```
comment:
  "Zero-extended halfword: load into register", loads the least
  significant 12 bits of the specified register with the contents
  of the halfword pointed out by the effective address. The most
  significant 12 bits are cleared. If the effective address is 0
  through 7, "zl" is a half register to register transfer. Refe-
  rence to the non-accessible area leads to program exception.
  Numeric code:   19
  Escape pattern: 2.000010;
get word;
if addr(23) = 0 then
  regw:= 0 con opq(0:11)
else
  regw:= 0 con opq(12:23);
goto next instruction;
```

10.7 Monitor Calls and Interrupt

This section describes the actions on monitor calls and interrupts and the "return from interrupt" instruction. The actions are based on the contents of the system table pointed out by the information register "inf". The contents of "inf" must be odd to support the selection of the correct service address. The memory addresses involved in reading from the system table and in access of the registerdump are physical addresses. They are assumed to be addresses in true memory, if not the result is usual-

ly a buserror leading to an internal interrupt in the monitor program. The cause will be defined as "operand error" (cause = 2 * 4) or "system table error" (cause = 2 * 3) depending on the model facilities for checking the physical address before access without delaying the action. The casual reader is however advised to ignore the error cases of the algorithms, since these are of interest to monitor programmers and maintenance staff only.

10.7.1 external interrupt:

comment:
This action is entered from "next instruction", 10.4.1, if the "interrupt flag" is set. The "cause" register is set to two times the interrupt level, i.e. at least 12, and the corresponding interrupt bit is cleared. The action continues at "service" the common action for entry of the monitor program. The entry address will be the interrupt service address in the systemtable;

```
cause:= 2 * curlev;
intreg (curlev):= 0;
goto service;
```

10.7.2 fetch error:

comment:
This internal interrupt action is entered from "next instruction", 10.4.1, or in some cases from jump actions, when the instruction fetch results in a buserror. The "cause" register is set to two times the interrupt level i.e. 10. The exception register will describe the error.

The instruction counter "ic" will contain the logical address of the instruction. The action continues at "service" the common action for entry of the monitor program. The entry address will be the interrupt service address in the "system table". If "intlim" is zero, defining total disable, the CPU will halt;

```
cause:= 2 * 5;
set busexceptions;
if intlim = 0 then halt;
goto service;
```

10.7.3 operand error:

comment:
This internal interrupt action is entered from instructions and indirect address calculation when a memory reference results in a buserror. The cause register is set to two times the inter-

rupt level i.e. 8. "Ex", the exception register will describe the error. The address register "addr" will contain the logical address of the memory reference. The action continues at "service" the common action for entry of the monitor program. The entry address will be the interrupt service address in the "system table". If "intlim" is zero, defining total disable, the CPU will halt;
cause:= 2 * 4;
set bus exceptions;
if intlim = 0 then halt;
goto service;

10.7.4 call:
comment:
This action is entered from the special monitor call case of the "jd" instruction, 10.6.24. The "cause" register contains the function number i.e. the effective address plus 2048. The action continues at "service" the common action for entry of the monitor program. The entry address will be the monitor call address in the system table;
inf:= inf - 1;
comment: the least significant bit of "inf" is used as a flag to control the selection of service address;

service:
comment:
This common action for entry of the monitor program dumps the eight dynamic registers, at the "registerdump address" in the "system table" and initializes the dynamic registers for fast monitor action. For details, see 6.2 or follow the algorithm. Buserrors during the action will activate the "dumpfault" action, see below;

base:= 0;
lowlim:= 8;
uplim:= size;
initq:= word (inf);
if buserror then halt;
comment if the status/intlim initialization cannot be accessed a total disable situation must be assumed. Inf must point to a system table in true memory or must be initialized to guaranty a buserror as after autoload and power restart;
intlim:= 0 con initq(12:23);
comment the interruptflag is cleared and the interrupt scan is resumed as a function of this assignment;

```
regdumpaddrq:= word (inf + 2);
if buserror then goto dumpfault;
for iq:= 0 step 2 until 14 do
  begin
  word (regdumpaddrq + iq):= reg(iq);
  if buserror then goto dumpfault;
  comment the "register dump address" in the systemtable must
    point to eight consequtive words in true memory;
  end;
ic:= word (inf - 3);
if buserror then goto dumpfault;
comment the entry address is either the "monitor call service
  address" or "interrupt service address" in the "system table"
  depending on the flag in the least significant bit in "inf";
w1:= regdumpaddrq + 16;
w2:= cause;
if dumperrorcount <> 0 then
  begin comment see dumpfault below;
    w2:= 4
  end;
status:= initq (0:11) con 0;
inf:= inf - 12;
inf:= inf (0:22) con 1;
comment the new "system table" is selected and the flag for
  selection of service address is reinitialized;
pic:= ic;
instruction:= word (pic);
if buserror then goto fetcherror;
fetched:= true;
goto next instruction;

dumpfault:
comment:
  This action is called if an interrupt or monitor call cannot be
  handled according to the current "system table" because of a
  buserror. A new "system table " is selected and "dump error
  count" is increased by one. After that the action continues at
  "service" simulating an internal interrupt in the monitor pro-
  gram. The register dump will be as for the original event ex-
  cept for the increase in "dump error count". The "w2" initiali-
  zation will be changed to 4 under the assumption, that a moni-
  tor program will never provoke a floating point interrupt;
inf:= inf - 12;
inf:= inf (0:22) con 1;
dumperrorcount: dumperrorcount + 1;
```

```
if dumperrorcount = 0 then halt;
comment halt on "dump error count" overflow, i.e. the incredible
  situation where 16 system table levels can not handle the
  event;
if intlim = 0 then halt;
goto service;
```

10.7.5 ri algorithm:

```
comment:
"Return from interrupt" is mainly used by the monitor program to
  reactivate user programs after interrupts and monitor calls.
  Besides that it is used in systeminitialization, to start new
  user programs and to enforce breaks in user program execution.
  For details, see 6.3 or follow the algorithm.

  Buserrors during the restoring of registers will be handled
  analogous to operand errors except for the possibility of cause
  = 2 * 3 i.e. "system table error".

  Exit to the user program is performed by "exit to program"
  10.2.10. This action is entered with the auxiliary registers
  "oldicq" pointing after the "ri" instruction and "aq" pointing
  after the 5 words defining the reinitialization of the "process
  definition registers". "ri" is a privileged instruction.
  Numeric code:   12
  Escape pattern: 2.111010;
if -, monitor mode then goto program exception;
inf:= inf + 12; comment select new system table;
displq:= addr;
oldicq:= ic;
regdumpaddrq:= word (inf + 2);
if buserror then
  begin
    addr:= inf;
    goto ri error
  end;
for iq:= 0 step 2 until 14 do
  begin comment assignment of w0, w1, w2, w3,
  status, ic, cause, addr;
  reg(iq):= word (regdumpaddrq + iq);
  if buserror then
    begin
      addr:= regdumpaddr + iq + 2;
      goto ri error
    end
  end;
```

```
regdumpaddrq:= regdumpaddrq + displq;
for iq:= 0 step 2 until 8 do
  begin comment assignment of cpa,
    base, lowlim, uplim, intlim;
    reg (iq + 16):= word (regdumpaddrq + iq);
    if buserror then
      begin
        addr:= regdumpaddrq + iq + 2;
        goto ri error
      end
  end;
aq:= regdumpaddrq + 10;
dumperrorcount:= 0;
goto exit to program;

ri error:
comment analogous to "operand error", but "addr" do not point
  directly to the error;
cause:= 2 * 4; comment or: cause:= 2 * 3, "ex" undefined;
set busexceptions;
ic:= oldicq;
inf:= inf - 12; comment reselect old system table;
if intlim = 0 then halt;
goto service;
```

10.8 Exception and Escape

This section describes the actions for handling exceptions and escapes and the "return from escape" instruction. The exception and escape actions access the "system table" for the logical service addresses. Concerning these accesses, see the introduction to 10.7.

10.8.1 program exception:

comment:

This action is called in the following cases:

- The memory protection is violated, "addr" contains the logical address of the operand or of the preoperand.
- Execution of an unassigned instructioncode, "addr" contains the effective address.
- Execution of a privileged instruction without monitor mode. "addr" contains the effective address.

In all cases "ic" contains the logical address of the next in-

```
    struction. The "cause" register is set to zero and the action
    continues at "exception" below;
  cause:= 0;
  goto exception;
```

10.8.2 integer exception:

```
  comment:
    This action is called when an integer operation or an arithme-
    tic shift results in an overflow and the "integer exception ac-
    tive" bit in the "status" register is set, see 4.6 which also
    describes the setting of bits in the exception register. For
    memory referring instructions "addr" will contain the logical
    address of the operand, for doubleword instructions the preope-
    rand. "ic" contains the logical address of the next instruc-
    tion.
    The "cause" register is set to 2 and the action continues at
    "exception" below;
  cause:= 2;
  goto exception;
```

10.8.3 floating point exception:

```
  comment:
    This action is called when a floating point operation results
    in overflow or underflow and the "floating point exception ac-
    tive" bit is set in the "status" register, see 5.4 which also
    describes the setting of bits in the exception register. "addr"
    contains the logical address of the preoperand (for "ci" the
    scaling factor). "ic" contains the logical address of the next
    instruction.
    The "cause" register is set to 4 and the action continues at
    "exception" below;
  cause:= 4;

  exception:
  comment:
    The "exception service address" is read from the "system
    table". The "status" assignment following the register dump is
    prepared. The action continues at "dump registers" below;
  statq:= status;
  statq (2:3):= 0; comment "after am" and "after esc" will be
    cleared before exit to the exception routine;
  regdumpaddrq:= word (inf + 4);
  if buserror then goto service address error;
  goto dump registers;
```

10.8.4 escape:

```
comment:
  This action is called after the address calculation, when "es-
  capemode", "escapemask" and "escapepattern" defines an escape.
  "addr" contains the effective addr. "instrmask" contains the
  concatination of the F-field and the "escapepattern" of the in-
  struction. For the special case of escape in indirect address
  calculation see 10.3.5. "Ic" is decreased by 2 such that it
  contains the logical address of the instruction. "After escape"
  is set to true. The "escape service address" is read from the
  "system table". The "status" and "cause" assignments following
  the register dump is prepared. The action continues at "dump
  registers" below;
ic:= ic - 2;
after esc:= true;
statq:= status;
statq(1:11):= 0; comment "escape mode",
  "after am", "after escape", "integer exception active",
  "floating point exception active" and "escape mask" will be
  cleared before exit to the escape routine;
cause:= instrmask;
regdumpaddrq:= word (inf + 6);
if buserror then goto service address error;
goto dump registers;

service address error:
comment the effect is analogous to the effect of an "operand
  error" in the "monitor" program. "addr" will point to the
  "system table";
addr:= inf;
cause:= 2 * 4; comment or: cause:= 2 * 3, "ex" undefined;
set busexceptions;
inf:= inf - 12; comment select the "monitor" system table;
if intlim = 0 then halt;
goto service; comment 10.7.4;

dump registers:
comment:
  This common action for exceptions and escapes dumps the 8 dy-
  namic registers in 8 consequtive words starting at the service
  address, and initializes the dynamic registers for fast action
  on the event. For details, see 7.1.4 and 7.2.4 or follow the
  algorithm. If the service address is zero the event is trans-
  formed to an internal interrupt, for exceptions with unchanged
  "cause", for escapes with "cause = 0", i.e. "program interrupt".
```

```
    If the register dump is outside the process area an internal
    interrupt with "cause = 2 * 3", i.e. "system table error" is
    activated. Exit to the program at "service address + 16" is
    performed by "exit to program", 10.3.10. This action is entered
    with the auxiliary register "oldicq" pointing to the cause and
    "aq" = service address + 16;
  if regdumpaddrq = 0 then
    begin
      if cause > 4 then cause:= 0;
      comment the cause is always greater
        than 4 for escapes;
      if intlim = 0 then halt;
      goto service; comment 10.7.4
    end;
  for iq:= 0 step 2 until 14 do
    begin
      if lowlim <= regdumpaddrq + iq + base < uplim then
        begin
          word (regdumpaddrq + iq + base):= reg (iq);
          if  buserror then
            begin
              comment analogous to "operand error" but "addr" will
                contain the logical address of the error increased
                by 2;
              addr:= regdumpaddrq + iq + 2;
              cause:= 2 * 4;
              set busexceptions;
              if intlim = 0  then halt;
              goto service; comment 10.7.4;
            end
        end
      else
        begin comment system table error;
          addr:= regdumpaddrq + iq + 2;
          cause:= 2 * 3;
          if intlim = 0 then halt;
          goto service; comment 10.7.4;
        end
    end;
  w0:= ic;
  w1:= 0 (4) con cause (0:17) con W;
  comment exceptions: w1:= W, escapes: w1:= F con W;
  w2:= addr;
  cause:= 0 con cause (18:23);
```

```
comment exceptions: cause unchanged
        escapes: cause:= escape pattern;
w3:= cause;
oldicq:= ic;
aq:= regdumpaddrq + 16;
ic:= aq;
status:= statq; comment subfields of "status" are cleared as
  prepared at entry;
goto exit to program;
```

10.8.5 re algorithm:

comment:

"Return from escape" initializes the 8 dynamic registers from 8 consequtive logical addresses starting at the effective address. Each of the logical addresses must be either in the process area or in the common protected area or a "program exception" will take place. For further details, see 7.2.5 or follow the algorithm. Exit to the program at the new contents of "ic" is performed by "exit to program", 10.3.10. This action is entered with the auxiliary registers "oldicq" pointing after the "re" instruction and "aq" equal the effective address increased by 16;

```
regdumpaddrq:= addr;
oldicq:= ic;
monmq:= monitor mode; comment used to prevent illegal setting of
  "monitor mode";
for iq:= 0 step 2 until 14 do
  begin
    if lowlim <= regdumpaddrq + iq + base < uplim then
      begin
        reg(iq):= word (regdumpiq + iq + base);
        if buserror then goto re error
      end
    else
      if 8 <= regdumpaddrq + iq < cpa then
        begin
          reg(iq):= word (regdumpaddrq + iq);
          if buserror then goto re error
        end
      else
        begin
          ic:= oldicq;
          goto program exception;
          comment "addr" still contains the original effective
            address;
        end;
```

```
if -, monmq then monmode:= false;
dumperrorcount:= 0; comment prevents erronous "w2:= 4" in
  later interrupts or monitor calls;
goto exit to program;

re error:
ic:= oldicq;
addr:= regdumpaddrq + iq + 2;
cause:= 2 * 4;
set busexceptions;
if intlim = 0 then goto halt;
goto service; comment 10.7.4;
```

A1 Appendix 1

A1.1 Instructions in Alphabetic Order by Mnemonics

MN	NC	NAME	EP	REF
aa	56	add double word to double register	2.001010	4.4
ac	33	address complemented: load into register	2.000001	3.4.1
ad	37	arithmetic shift of double register	2.000001	3.4.3
al	11	address: load into register	2.000001	3.4.1
am	9	address: modify that of next instruction	2.000001	3.3.2
as	36	arithmetic shift of single register	2.000001	3.4.3
cf	53	convert floating point to integer	2.000001	5.5
ci	32	convert integer to floating point	2.000001	5.5
di	29	data in	2.100001	8.2.1
dl	54	double register: load	2.001010	3.2
do	1	data out	2.100001	8.2.2
ds	55	double register: store	2.001100	3.7
ea	18	extended halfword: add to register	2.000010	4.2
el	2	extended halfword: load into register	2.000010	3.7
es	17	extended halfword: subtract from register	2.000010	4.2
fa	48	floating point: add to double register	2.001010	5.4
fd	52	floating point: divide into double register	2.001010	5.4
fm	50	floating point: multiply by double register	2.001010	5.4
fs	49	floating point: subtract from double register	2.001010	5.4
gg	28	general get from processor register	2.000001	6.8
gp	47	general put into processor register	2.100001	6.8
hl	3	half register: load	2.000010	3.7
hs	26	half register: store	2.000100	3.7
jd	14	jump and select disable limit	2.110000 *)	3.6, 6.4 6.6
je	15	jump and select enable limit	2.110000 *)	3.6, 6.6
jl	13	jump with link in register	2.010000	3.6

*) In model RC 8000/45: 2.10000

MN	NC	NAME	EP	REF
la	4	logical and: combine word with register	2.000010	3.8
ld	39	logical shift of double register	2.000001	3.4.3
lo	5	logical or: combine word with register	2.000010	3.8
ls	38	logical shift of single register	2.000001	3.4.3
lx	6	logical exclusive or: combine word with register	2.000010	3.8
nd	35	normalize double register	2.000100	5.5
ns	34	normalize single register	2.000100	5.5
re	22	return from escape	2.011010	7.2.5
ri	12	return from interrupt	2.111010	6.3
rl	20	register: load	2.000010	3.7
rs	23	register: store	2.000100	3.7
rx	25	register: exchange with word	2.000110	3.7
se	42	skip if register equal	2.010001	3.4.2
sh	40	skip if register high	2.010001	3.4.2
sl	41	skip if register low	2.010001	3.4.2
sn	43	skip if register nonequal	2.010001	3.4.2
so	44	skip if selected register bits all ones	2.010001	3.4.2
sp	21	skip if word not protected	2.010001	3.9
ss	57	subtract double word from double register	2.001010	4.4
sx	46	skip if selected exception bits all zeros	2.010001	4.7
sz	45	skip if selected register bits all zeros	2.010001	3.4.2
wa	7	word: add to register	2.000010	4.3
wd	24	word: divide into double register	2.000010	4.5
wm	10	word: multiply by register giving double register	2.000010	4.5
ws	8	word: subtract from register	2.000010	4.3
xl	16	exception register: load from halfword	2.000010	4.7
sx	27	exception register: store in halfword	2.000100	4.7
zl	19	zero-extended halfword: load into register	2.000010	3.7

A2 Appendix 2

A2.1 Instructions in Order of Numeric Code

NC	OC	MN	EP	EX
0	0	-	100000	
1	1	do	100001	
2	2	el	000010	L
3	3	hl	000010	L
4	4	la	000010	L
5	5	lo	000010	L
6	6	lx	000010	L
7	7	wa	000010	L I
8	10	ws	000010	L I
9	11	am	000001	
10	12	wm	000010	L
11	13	al	000001	
12	14	ri	111010	L P
13	15	jl	010000	L
14	16	jd	1 0000	L
15	17	je	1 0000	L
16	20	xl	000010	L
17	21	es	000010	L I
18	22	ea	000010	L I
19	23	zl	000010	L
20	24	rl	000010	L
21	25	sp	010001	
22	26	re	011010	L
23	27	rs	000100	S
24	30	wd	000010	L I
25	31	rx	000110	S
26	32	hs	000100	S
27	33	xs	000100	S
28	34	gg	000001	
29	35	di	100001	
30	36	-	100000	
31	37	-	100000	
32	40	ci	000001	F
33	41	ac	000001	I
34	42	ns	000100	S
35	43	nd	000100	S
36	44	as	000001	I
37	45	ad	000001	I

NC	OC	MN	EP	EX
38	46	ls	000001	
39	47	ld	000001	
40	50	sh	010001	
41	51	sl	010001	
42	52	se	010001	
43	53	sn	010001	
44	54	so	010001	
45	55	sz	010001	
46	56	sx	010001	
47	57	gp	100001	P
48	60	fa	001010	L F
49	61	fs	001010	L F
50	62	fm	001010	L F
51	63	-	100000	
52	64	fd	001010	L F
53	65	cf	000001	I
54	66	dl	001010	L
55	67	ds	001100	L
56	70	aa	001010	L I
57	71	ss	001010	L I
58	72	-	100000	
59	73	-	100000	
60	74	-	100000	
61	75	-	100000	
62	76	-	100000	
63	77	-	100000	
(64	100	indir	000010)	

EX: Potential exceptins; notation means:

L: reference to "non accessible area" leads to program exception

S: reference outside "process area" leads to program exception

I: may cause integer exception

F: may cause floating point exception

P: use with monitormode false leads to program exception.

A3 Appendix 3

A3.1 Instructions in Order of Escape Pattern

EP	NC	MN	EP	NC	MN
000001	9	am	001010	48	fa
	11	al		49	fs
	28	gg		50	fm
	32	ci		52	fd
	33	ac		54	dl
	36	as		56	aa
	37	ad		57	ss
	38	ls			
	39	ld			
	53	cf	001100	55	ds
000010	2	el	010000	13	jl
	3	hl			
	4	la	010001	21	sp
	5	lo		40	sh
	6	lx		41	sl
	7	wa		42	se
	8	ws		43	sn
	10	wm		44	so
	16	xl		45	sz
	17	es		46	sx
	18	ea			
	19	zl	011010	22	re
	20	rl			
	24	wd	100000	0	-
	(64	indirect)		30	-
				31	-
000100	23	rs		51	-
	26	hs		58	-
	27	xs		59	-
	34	ns		60	-
	35	nd		61	-
				62	-
000110	25	rx		63	-

EP	NC	MN	EP	NC	MN
100001	1	do			
	29	di			
	47	gp			
110000 *)	14	jd			
	15	je			
111010	12	ri			

*) In RC 8000/45 "jd" and "je" has the escape pattern 100000

A4 Appendix 4

Memory locations supporting firmware facilities.

Fixed addresses:

```
0:      w0              ; The working registers are addressable
2:      w1              ; as the first locations of memory
4:      w2              ; see 3.2
6:      w3              ;

8:      device base     ; Points to the device descriptions, see
                        ; 8.3.2

10:     power restart address; Points to the power restart action,
                              ; see 9.1
```

Systemtable, see 6.1, pointed out by the contents of the "inf" register:

```
inf-5: monitor call service address     ; see 6.4
inf-3: interrupt service address        ; see 6.7
inf-1: status/intlim initialization     ; see 6.2
inf+1: register dump address            ; see 6.2
inf+3: exception service address        ; see 7.1
inf+5: escape service address           ; see 7.2.3 ff.
```

RETURN LETTER

Title: **RC 8000 Computer Family, Reference Manual** RCSL No.: 42-i 1235

A/S Regnecentralen maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Name: Title:

Company:

Address:

Date:

Thank you

. . . . . Fold here . . . . .

. . . . . Do not tear - Fold here and staple . . . . .

Affix
postage
here

A/S REGNECENTRALEN
Information Department
Falkoner Alle 1
DK-2000 Copenhagen F.
Denmark