H. Isaksson.

DESIGN CONSIDERATIONS FOR THE RC 4000 COMPUTER

Per Brinch Hansen and Peter Kraft

A/S Regnecentralen, Copenhagen

March, 1966

Summary:

The RC 4000 computer which will be developed by Regnecentralen in 1966 is a low-cost computer oriented towards process control applications. The present report deals with the choice of data and instruction formats, storage addressing, arithmetic, input-output control, and special multiprogramming features.

Contents:

1. System RC 4000.

1.1. Introduction.

RC 4000 is a general purpose digital computer designed and manufactured by Regnecentralen. The basic model is a low-cost computer oriented towards real-time computation in industrial control applications. An extended model includes floating-point arithmetic for scientific computation and high-speed input-output devices for commercial data processing.

This report is a preliminary description of the programming structure of the new computer. The rest of section 1 outlines the objectives that guided the design of system RC 4000. This is followed by a short summary of the system. Sections 2 to 7 contain a detailed discussion of the choice of word length, storage addressing, fixed-point arithmetic, input-output control, and special multiprogramming features. Section 8 completes the picture of the system with a formal definition of the instruction set.

The prototype of the RC 4000 is intended for installation in a chemical plant constructed by Haldor Topsøe in Pulawy, Poland. Appendix A gives a brief survey of the main tasks of the computer in the Pulawy installation. This is included as an example of a typical application of system RC 4000 in industrial process control.

1.2. Design Objectives. +)

The structure of system RC 4000 is based on well-known principles selected with the following objectives in mind:

(1) Efficient handling of small integers.

(2) Fixed-point arithmetic with adequate precision for process control and provisions for fixed-point and floating-point arithmetic with double-precision.

(3) Flexible modification of addresses.

(4) Uniform register structure to avoid empty data transfers.

+) The RC 4000 computer may be evaluated in relation to other process control computers available on the market by referring to a recent paper by Edward O. Boutwell: Comparing the Compacts, Datamation, December, 1965.

(5) Instruction set which is conceptually simple without being inefficient.

(6) General input-output control with no restrictions on the kinds of devices that may be connected.

(7) Protection features to ensure absolute monitor control of the system.

(8) Program interruption system providing a constant monitoring of exceptional internal and external conditions.

1.3. System Summary.

Implementation.

Integrated circuits with propagation times of 15 nanoseconds.

Storage.

Magnetic core store with 2 microseconds cycle time. Basic modules of 1 k, 4 k, and 8 k words of 24 bits each. No upper limit to expansion. Standard parity check and protection system.

Speed.

Typical instruction times 6 to 8 microseconds.

Addressing Facilities.

Direct addressing of 12-bit bytes and 24-bit words. Indexing and indirect addressing facilitate table look-up. Relative addressing simplifies program relocation.

Register Structure.

4 directly addressable registers function both as accumulators and index registers. This uniform register structure makes the entire instruction set available for index arithmetic and inter-register operations.

Arithmetic.

Parallel binary two's complement arithmetic. Standard integer arithmetic with 12-bit and 24-bit operands. Optional floating-point arithmetic with 48-bit operands.

Protection System.

Privileged instructions and read-only protection associated with a monitor mode guarantee constant monitor control of the system.

Interruption System.

Program interruption system with up to 24 priority levels controlled by a mask register. optional

Input-output Control.

Standard interface between data channels and input-output devices simplifies future expansion of the array of input-output equipment. Standard low-speed channel with transmission of single words under program control. Optional high-speed channel with transmission of blocks proceeding simultaneously with program execution.

Real-time Clock.

The clock initiates program interruptions at preset time intervals. It is set and sensed under program control.

Standard Peripherals.

Console typewriter - 10 characters per second.
Paper tape reader - 100 characters per second.
Paper tape punch - 20 characters per second.

2. Data Formats.

2.1. Storage of Small Integers.

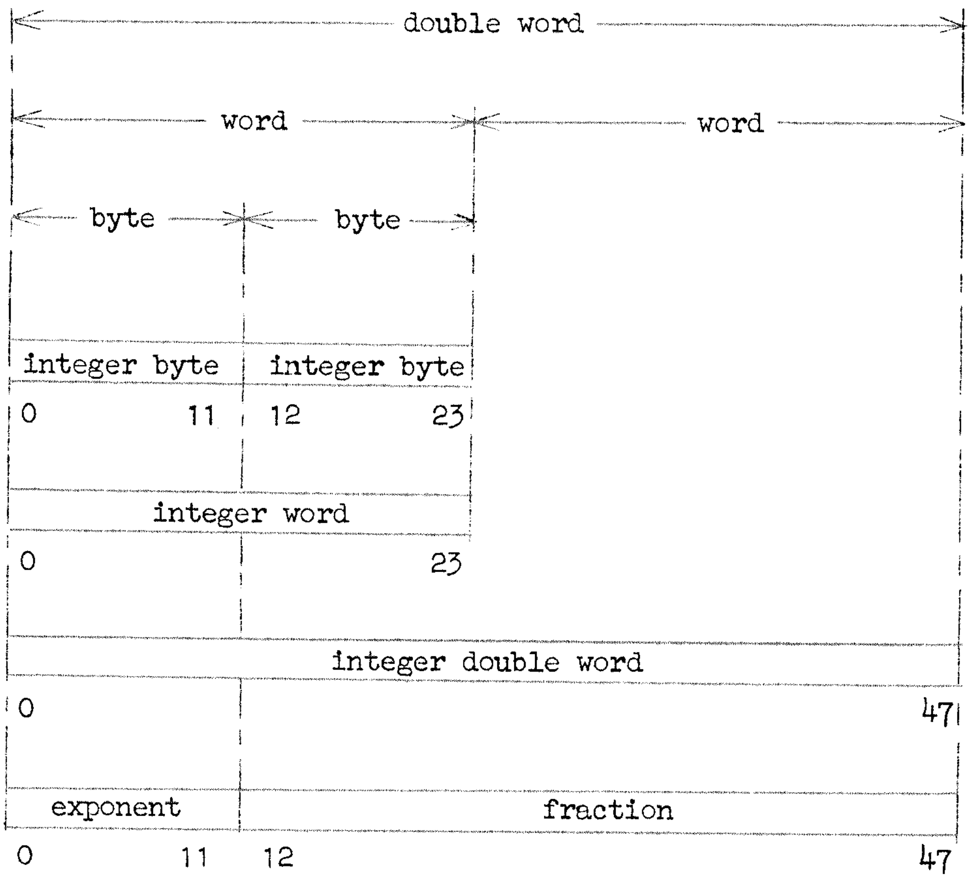
The majority of data used in process applications are measured variables delivered by analog-to-digital converters. In the Pulawy plant system RC 4000 supervises about 700 process variables of 10 - 12 bits each. In order to handle effectively such variables the computer should be able to address directly bytes of 10 to 12 bits.

2.2. Arithmetic Precision.

In a data logging system the processing of most of the process variables is very simple (testing against alarm limits, printing of log sheets). In a closed-loop control system the variables are, however, used in elaborate analysis of the performance of the plant. These computed variables determine the normal arithmetic precision. It is estimated that a precision of 20 to 24 bits will be adequate for process applications. The corresponding double precision of 40 to 48 bits should also be satisfactory for most commercial and scientific computations.

2.3. RC 4000 Data Formats.

The efficient storage of analog input data points to a short word length. In practice the lower bound on the word length is placed by the format of instructions and arithmetic operands. As a compromise the following word structure was chosen:



The basic arithmetic operand is an integer of 24 bits. Small integers are packed with two bytes per word. The bytes are directly addressable. A special instruction LOAD INTEGER BYTE serves to extend a 12-bit byte towards the left to 24 bits as it is introduced into a working register.

Double words are used to represent integers of double precision (48 bits) and floating-point numbers (12-bit exponent and 36-bit fraction).

3. Instruction Format.

3.1. Address Modification.

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

3.1.1. Program Relocation.

The ability to relocate programs in the working store is vital in a machine where the library of programs is kept on a backing store and only brought to the working store when they are active. As an example consider a computer which performs direct control of a plant. In idle intervals the computer may perform other tasks such as translating programs from the assembly language into machine code. This may at any time be interrupted by an exception signal from the plant causing the assembler to be replaced by a process control program. Later the assembler will be brought back into the store and resume the translation. Normally it is not possible to predict the combination of programs and data residing in core at the time of reloading. The programs therefore cannot expect to be loaded into their previous residence areas but must be relocated to storage areas currently available.

Relocation is also desirable in a machine without backing stores. This allows the machine to be loaded with different sets of programs each of which can be written independently of the others i.e. as standard routines.

Dynamic relocation can be implemented either as base addressing (implying an address modification with a register containing the load address of the program), or as relative addressing (implying a modification with the current instruction address). The latter solution is used in the RC 4000.

3.1.2. Table Look-up.

The purpose of data processing is to transform a set of data into a result according to certain rules. In a computer with an addressable store one of the most general ways of specifying the rules of transformation is to use a set of tables. Each piece of data to be transformed is converted to an address which is used to look-up a table to extract a new data value or the address of a work action to be performed. The requirement that addresses can be modified by the values of the data being processed is met efficiently by the use of index registers.

3.2. Register Structure.

The motivations for having registers separated from the main store are: (1) to provide fast access to frequently used data items, and (2) to save instruction length by having implied or truncated addresses of operands. In a single-address computer with an implicitly addressed accumulator these advantages are to some extent illusory. Since all operations destroy the previous contents of the accumulator the programmer is forced to make a lot of storage operations in order to save and restore the accumulator. Such empty transfers also arise in machines where the index registers can only be modified by the accumulator via a transport to the store. An analysis of programs written for the IBM 7090 showed that about 30 percent of the load and store instructions were such superfluous transfers. The experiment also showed a reduction of more than 90 percent of these transfers when four accumulators were introduced instead of one +).

+) G.M. Amdahl, The Structure of System/360 - Processing Unit Design Considerations, IBM Systems Journal 3, no. 2 - 3, 1964.

The register structure in the RC 4000 computer consists of four working registers of 24 bits each. In each instruction one of these registers can be specified as the accumulator. Likewise one of them can be selected as the current index register. By extending the number of accumulators to four and by removing the distinction between accumulators and index registers the full instruction set becomes available for immediate address modification, and empty transfers of registers are reduced considerably. Also the instruction set is simplified since special instructions for the handling of index registers are no longer needed. To facilitate register-to-register operations the working registers are addressable as the first four words of the main store.

word address		
0	24 bits	working register 0
1	24 bits	working register 1
2	24 bits	working register 2
3	24 bits	working register 3

3.3. RC 4000 Instruction Format.

To allow a flexible manipulation of both the operation part and the address part the following instruction format was selected:

F	W	M	X	D
6	2	2	2	12

The instruction word is divided into an operation byte and an address byte of 12 bits each. The operation byte specifies 64 basic operations in the F field of 6 bits. One of the four working registers is specified as the result register in the W field of 2 bits. The current index register is selected by the X field of 2 bits. Only working registers W1, W2, and W3 act as index registers (X=0 indicates no indexing).

A truncated address of 12 bits (the D field) specifies a displacement of 4095 bytes inside the program which is adequate for the majority of addresses. It is, however, insufficient for specifying directly the entire store. A full address of 24 bits is formed by means of the displacement in connection with an index register (X) and the instruction counter (R). The formation of an effective address A is controlled by the address mode field M as follows:

M=00	A = X + D
M=01	A = X + R + D
M=10	A = store[X + D]
M=11	A = store[X + R + D]

In the address calculation the displacement is treated as a 12-bit signed integer which 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, R, and D overflow is ignored.

The last two modes permit indirect addressing in one level. The indirect address is assumed to be a full address of 24 bits. A special instruction MODIFY NEXT ADDRESS acts as a substitute for multilevel indirect addressing.

In storage access operations the effective address is treated as an unsigned integer of 24 bits. Reference to a non-existent storage location will cause a program interruption (see section 7.3).

The use of truncated addresses of 12 bits permits a more flexible instruction format within the given word length of 24 bits. At the same time the extension of addresses in registers to 24 bits allows practically unlimited extension of the working store. Comparison of addresses is also simplified because all effective addresses are positive.

4. Fixed-point Arithmetic.

4.1. Number Representation.

The efficiency of the fixed point arithmetic depends strongly on the representation of signed numbers. The traditional alternatives are:

(1) Separate sign bit plus absolute value.

(2) One's or two's complement.

The representation chosen for the RC 4000 computer is the two's complement. The main virtue of the complement notation is the simple handling of operands with opposite sign as compared to the sign plus magnitude representation. In the latter system an addition of two signed numbers A + B requires a comparison of both signs and magnitudes to determine whether the resulting magnitude should be computed as abs(A) + abs(B), abs(A) - abs(B), or abs(B) - abs(A). In the complement arithmetic the two operands are simply added as if they were both unsigned binary integers of 24 bits. And it is only outside the arithmetic unit the data words are interpreted as signed integers according to the following rule:

```
signed integer:= if data word ≤ 2↑23 - 1
                 then + data word else - (2↑24-data word);
```

The choice of two's instead of one's complement arithmetic was dictated by (1) the unique representation of zero, and (2) the faster addition because there is no need for an extra cycle to add an end-around carry.

The complement notation also facilitates the handling of small integers (byte arithmetic) and large integers (double-precision arithmetic). A small integer can be extended to the standard form of 24 bits simply by a duplication of the sign bit towards the left. Conversely when the high-order digits of a small integer are elided the leading digit in the truncated integer still reflects the sign properly. In the programming of double-precision arithmetic all operations on the low-order fields can be performed as if they were unsigned integers of 24 bits (provided there is an indication of a carry from one word to the next).

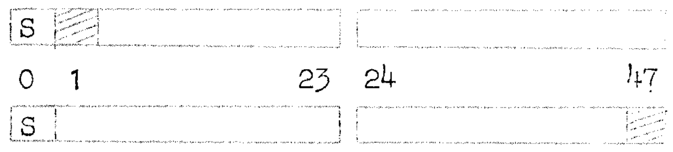
4.2. Point Location.

This section is concerned with the choice of fixed-point arithmetic. The main decision to be taken is whether the arithmetic unit should interpret data words as (1) pure integers, (2) pure fractions, or as (3) numbers with an intermediate point location.

We will assume that the arithmetic unit is designed to develop the same result digits regardless of where the binary point is located. For example, multiplication of two data words is assumed to produce a double-length product. The difference between the three types of arithmetic lies in the way the results are aligned to retain the original point location in subsequent operations. In the case of addition and subtraction alignment of the result is trivial. This however, is not true for multiplication.

In arithmetic with an intermediate point location a post-shift is required to align a product. During this shift significant digits may be lost even though the unaligned result does not exceed the capacity of the double-length accumulator. This deficiency makes an intermediate point location undesirable.

Let us now consider integer and fractional arithmetic. In a machine with double-length multiplication and a uniform register structure the practical difference between integer and fractional arithmetic is very small. It is largely a question of whether the programmer chooses to select the low-order or the high-order part of the product for further computation. The only functional difference between integers and fractions may be described as follows. A data word consists of a sign plus 23 binary digits. Multiplication therefore produces a double-length product consisting of a sign plus 46 digits. This leaves a vacant bit-position in a double-length accumulator of 48 bits. In integer arithmetic the vacant position is logically placed at the extreme left following the sign bit, while in fractional arithmetic it would be placed at the extreme right of the accumulator. This representation of double-length products is shown by the following figure. The vacant bit position is indicated by hatching. The difference between integer and fractional arithmetic is seen to be just a matter of shifting double-length products one position.



In the GIER computer the vacant position is placed in bit 24 as a zero. This representation has the deficiency that integer multiplication can only produce low-order products with positive sign. Consideration was also given to a representation where the sign is copied from bit 0 into bit 24 after the multiplication. This representation was abandoned because it makes the programming of multiprecision arithmetic very awkward.

The fixed-point arithmetic in the RC 4000 computer works with integers of the type described above. We prefer integers to fractions for the following reasons:

(1) Most of the variables in process control applications are of integer nature i.e. they are variables with a fixed number of digits to the right of the point and of a magnitude that makes scaling unnecessary. Examples are measured process variables and table addresses.

(2) Aligning the numbers to the right reduces the chance of overflow. Note that with the integer representation chosen multiplication can never lead to overflow.

(3) Fractional computation can still be performed by shifting and normalizing instructions.

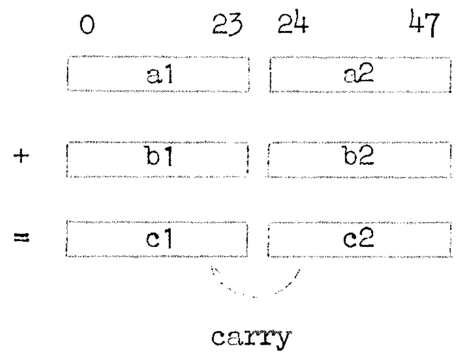
4.3. Multiple Precision.

With the short word length chosen for the RC 4000 computer serious consideration must be given to double-precision arithmetic. In process applications the need for extending precision is probably too limited to justify a complete instruction set for this purpose. We will therefore concentrate on providing a few basic facilities for the programming of multi-precision routines.

Consider first the addition of two double-precision numbers, a and b, composed of double words (a1, a2) and (b1, b2):

$$a + b = (a1 + b1) \times e + a2 + b2$$

where e is the scaling factor $2 \uparrow 24$. The low-order parts (a2 + b2) and the high-order parts (a1 + b1) are added separately as shown by the following figure:



The main problem is to detect a carry from one component word to the next. It is therefore suggested that single-word addition and subtraction give an indication of this lost-carry situation. (Note that lost-carry is different from overflow. Overflow indicates that the data word interpreted as a signed integer exceeds 2↑23 - 1. Lost carry on the other hand signifies that the data word interpreted as an unsigned integer exceeds 2↑24 - 1).

The second basic requirement of multi-precision arithmetic is that multiplication of two single words produces a double-length product to retain all result digits.

5. Exception Register.

Arithmetic instructions and input-output instructions indicate an exceptional outcome by setting a 2-bit exception register. This register can be tested by a single instruction SKIP IF NO EXCEPTIONS. The exception register is set in the following situations:

Integer Arithmetic. After a normal result both exception bits are set to zero. An overflow will set the left exception bit to one and provoke a program interruption as defined in section 7.3. A lost carry sets the right exception bit to one without causing interruption.

Floating-point Arithmetic. A normal result clears both exception bits to zero. Exponent overflow or underflow is registered by the left and right exception bits respectively and is followed by a program interruption.

Input-output. An input-output instruction may be rejected if a peripheral device is disconnected or busy as described in section 6. The cause of rejection is registered in the exception bits.

All other instructions leave the exception register unchanged.

6. Input-output Control.

6.1. I/O Requirements.

The design of the I/O control is based on the following principles:

(1) There should be no restrictions on the kinds of devices that may be connected to the computer.

(2) Program execution should continue while I/O operations are in progress.

(3) I/O exceptions must be under complete program control and may never cause a machine stop.

The I/O equipment in a typical process control installation consists of typewriters, paper tape devices, A/D and D/A converters, and a real-time clock as described in appendix A. Future installations will certainly include other types of devices such as magnetic tapes, disk, or drum. To allow such expansion of the system the connection of I/O devices must be standardized in such a way that the computer is unaware of the types of devices attached to it. This requires that I/O instructions identify devices by addresses only, and that all data channels have a standard width.

In real-time applications it is unacceptable to halt computation while a data transfer is in progress. To avoid this a device must release the computer as soon as an I/O operation has been initiated. The computer will then continue the program while the device completes its operation independently.

When the computer attempts to initiate an I/O operation the device may answer by a rejection indicating that it is occupied with another operation. This information must be available to the program to allow the choice of an alternative course. When an operation is completed the device must also deliver information about exceptional conditions which occurred during the execution. This is necessary because a real-time system cannot rely on the operator to discover and react on such emergencies.

6.2. Low-speed Data Channel.

The I/O devices in industrial control applications are often slow character-oriented devices of the following types:

device:	bits/character	characters/second
typewriter	6	10
paper tape reader	7	100
paper tape punch	7	20
analog input	13	20

These devices are connected to a simple data channel consisting of an external buffer register connected to one of the internal working registers by 24 transmission lines. The data channel may be shared by several devices. Each device has a separate buffer register of 24 bits. The operation of this low-speed data channel will now be discussed in detail.

6.3. I/O Instruction.

The computer recognizes only one I/O instruction, INPUT OUTPUT, which has the standard format of instructions (defined in section 3.3). Here the W field selects the interval working register which will be connected to the data channel. The effective address of the instruction is interpreted in the following way:

not used	command	channel	device
0 11	12 15	16 19	20 23

The device is identified by a channel address and a device address of 4 bits each. The instruction format sets a limit of 16 data channels for the system. To each channel a maximum of 16 devices may be connected. The channel can only sustain data transfers from one device at a time. Operations which do not involve data transfers to working registers may, however, be in progress on several devices on the same channel.

The I/O operation desired is defined by a command code of 4 bits. An I/O operation is initiated by the computer in two stages:

Selection Phase. The control unit of the computer attempts to establish a connection with the device by means of the channel/device address. The success of this selection depends on whether the device is:

(1) free
(2) busy or
(3) disconnected.

The device is considered disconnected if no device responds to the selection. This is either because the channel/device address designates a non-existent channel or device in the installation, or it may be due to power being switched off from the device. The device responds to the selection with a busy-signal if it is in the process of executing a previous I/O operation. In the busy and disconnected states the I/O operation is rejected and the computer immediately proceeds to the next instruction. The cause of rejection is made available to the program in the exception register.

Command Phase. Only if the device responds to the selection with a free-signal is it ready to accept the command. The command code is then transferred to the device on separate signal lines. The device may now respond with a signal demanding an immediate data transfer from the working register to the external buffer register or vice versa. Finally when the operation is initiated the computer proceeds to the next instruction.

6.4. I/O Commands.

The command codes are specific for each type of device. They are, however, all modifications of four basic commands: READ, SENSE, WRITE, and CONTROL.

The read command directs the device to start a transfer of the next character from the external data medium into its buffer register. The computer is released as soon as the operation is initiated (or rejected). The command specifies whether the device must terminate its operation by means of a program interruption.

The sense command is a request to the device to transfer a status and data word to the working register. The right-most bits of this status word contain the last character received in the buffer register, while the left-most bits indicate an exceptional outcome of the last I/O operation such as:

(1) command invalid
(2) intervention required
(3) data invalid
(4) data lost

An invalid command is a command code which cannot be executed by the particular device. Intervention is required if a card stacker is full, if a paper tape device runs out of paper, or if the door of a tape station is open, etc.. Data invalid signifies an attempt to read or write a bit combination which does not belong to the character set of the device, an analog input signal which is out-of-range, or a parity error from a magnetic tape. Data may be lost if the operator fails to respond to a typewriter request within a certain time interval. These conditions cause the device to terminate the operation immediately. The status word can only be sensed if the device is free. If the device is busy or disconnected the command is rejected as described previously. The sense command may therefore be used in connection with the exception register to test whether a device is busy or not.

The write command instructs the device to transfer the contents of the working register to its buffer register and start outputting it on the external medium. The computer is released as soon as the operation has been initiated (or rejected). The command code may or may not specify an end of operation interruption.

The control command causes a transfer of the working register to an external buffer register followed by a control operation of the corresponding device. The contents of the working register is interpreted either as a selection address (of a multiplexer terminal, or a track on a drum or a disk) or as a control code (specifying upspacing of a line printer, rewinding of a magnetic tape, etc.). The computer proceeds to the next instruction as soon as the control operation has been initiated (or rejected). The command code may specify an end of operation interruption.

The operation of the low-speed data channel may now be summarized as follows. The execution of an I/O instruction will always result in the exception register being set to indicate whether the operation specified by the command code was initiated or rejected by the device. The computer will in any case immediately proceed to the next instruction. A succesful input operation from a device is performed by two instructions. The first instruction (read) directs the device to start reading the next character into its buffer register. As soon as this operation has been initiated program execution continues. When the data value is available in the buffer register it may be transferred to a working register by a second instruction (sense). This is done immediately under program control. The data word also contains status bits which indicate whether the input operation was completed succesfully.

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

The end of an I/O operation may either be signalled directly by the device as a program interruption, or it may be interrogated by the program by means of a sense command.

6.5. High-speed Data Channel.

The use of the working registers in the data path is adequate for slow character-oriented peripheral equipment. This type of channel is far cheaper to implement than a channel which performs buffered I/O directly to or from the main store on a cycle-stealing basis. Such channels are, however, necessary to handle devices like disks, drums, magnetic tapes, punch cards, and line printers which transmit large data volumes at high rates. This problem is considered in detail in appendix B. Suffice it here to say that such operations are implemented as control commands with the working register holding the start address of the buffer area while the first location of this area defines the buffer length.

7. Multiprogramming Facilities.

At the Pulawy plant the computer must type out a log of all process variables every hour. This takes 6 - 12 minutes. It is essential that the scanning of points for alarm is repeated every 5 minutes. Concurrently with these two tasks a number of digital registers must be sampled every second. Finally the operator may wish to alter an alarm level from the control typewriter, and this must be interlaced with the scanning cycles.

In process applications the computer thus has a number of concurrent tasks to perform, and these tasks must be repeated at regular intervals if real-time control of the plant is to be sustained. In such a multiprogramming system it is vital that erroneous programs are prevented from interfering destructively with other programs. The different tasks must therefore be coordinated by a monitor program which has complete control of the system. In the following we consider three hardware facilities which are desirable (if not imperative) in order to guarantee constant monitor control: (1) storage protection, (2) privileged instructions, and (3) program interruption.

7.1. Storage Protection.

An erroneous task program may attempt to destroy important process constants or parts of the monitor program. This indicates that some kind of read-only protection of the store is desirable. Some of the possible techniques are:

(1) All storage references are made indirect through tables maintained by the monitor. These lists make available to a task program only those storage areas assigned to that program.

(2) The store is divided into blocks of equal size. Associated with each block is a key register set by the monitor. To each task program a specific key value is assigned, and storing operations are only accepted if the program key matches the storage key.

These ambitious solutions provide a completely safe system whereby a large number of programs are prevented from interfering destructively with each other. They were, however, considered too expensive to include in the RC 4000 computer. We will therefore concentrate on solutions where a number of task programs are prevented from destroying a common area (the monitor) while everything outside this area can be destroyed (including the task programs themselves). This limited protection is adequate if the task programs are debugged systematically; each task program must be debugged with all other task programs residing passive and protected in the store. When the debugging is completed protection should be limited to the supervisory program only.

(3) One solution is to define the monitor area by two boundary registers which all effective addresses are compared against. This solution was rejected as being too expensive and too slow in implementation.

(4) The solution chosen for the RC 4000 consists of extending each storage word with an additional bit. This protection bit does not enter the working registers in the data operations. It is only used by the control unit to test whether the storage word is protected against writing. Attempts to violate the protection is brought to the attention of the monotor by a program interruption. This is admittedly a rather rigid solution, but it is cheap to implement and acceptable from a programming point of view. In this protection system the monitor area consists of all storage words in which the protection bit is set. This area is protected against writing from task programs residing in unprotected areas. It is essential, however, that instructions inside the monitor are allowed to

modify themselves and to assign new values to local variables. The protection rule must therefore be modified in the following way:

1) A protected word may be changed by a protected instruction but not by an unprotected instruction.

2) An unprotected word may always be changed.

Read-only protection is not enough to ensure survival of the monitor. It must also be protected against erroneous calls i.e. jumps which enter the monitor at arbitrary points. This problem is solved by the following convention:

3) Attempts to execute a protected instruction following an unprotected instruction will cause a program interruption.

A normal call of the monitor is made by provoking a program interruption with information about the desired entry loaded in a working register. The program interruption transfers control to a fixed point in the monitor which then decides whether the entry is correct.

7.2. Privileged Instructions.

It is highly desirable that the extension of the protected area can be program controlled by the monitor. This can be achieved by two instructions SET and CLEAR PROTECTION BIT which can only be executed inside the monitor i.e. if they are protected themselves. This concept of privileged instructions must be extended further to prevent that task programs accidently seize control from the monitor. First task programs may not change the status of the interrupt system (for example by disabling it permanently). Secondly it must not be possible for a task program to monopolize an I/O device that is needed by other programs as well (for example the control typewriter). The following classes of instructions should therefore be executable inside the monitor only:

- storage protection control
  - SET PROTECTION BIT
  - CLEAR PROTECTION BIT
- program interruption control
  - LOAD MASK REGISTER
  - JUMP WITH INTERRUPT ENABLED
  - JUMP WITH INTERRUPT DISABLED
- input output control
  - INPUT OUTPUT

The protection system may be summarized as follows:

A program interruption sets the control unit in the monitor mode where all instructions can be executed as long as they are protected. The control unit returns to the task mode when the first unprotected instruction is executed. In the task mode program interruption results if the following is attempted:

(1) Storing into a protected location.

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

(3) Executing a privileged instruction.

7.3. Program Interruption.

7.3.1. General Discussion.

The program interrupt system permits an automatic switching from the current sequence of instructions to another sequence in immediate response to exceptional events. The interrupt system is desirable in a real-time computer for the following reasons:

(1) Monitor Control. The monitor is in danger of loosing control if a task program goes into an endless loop. The minimum requirement here is a real-time clock that returns control to the monitor by means of a program interruption after a preset time interval.

(2) Programming Efficiency. The interrupt system permits an automatic monitoring of hardware malfunction and exceptional conditions generated by the program (such as violation of the storage protection or an arithmetic overflow). Frequent program testing of such rare events is clearly uneconomical. Efficiency also dictates the use of interrupt signals from I/O devices. In a multiprogramming system it is undesirable to remain in a given task program waiting for the completion of an I/O operation. The waiting time may be used to switch to another task program. When the I/O operation is completed the device signals its availability to the monitor by means of a program interruption. Control can then be returned to the program which initiated the operation.

7.3.2. Interruption Logic.

An interrupt system must perform the following functions: (1) Collection of interrupt signals, (2) Interrogation of interrupt signals, (3) Selection among competing interrupt requests, (4) Saving of return information, and (5) Switching to the interrupt service routine. These automatic functions should be restricted to a minimum in order to make the implementation cheap and to maintain the programming flexibility.

The RC 4000 computer can collect up to 24 interrupt levels in an interrupt register. The monitor has selective control over these interrupt lines by means of a mask register. This program controlled register defines for each of the 24 interrupt lines whether an interrupt request will be honoured or ignored. The interrupt register is interrogated once in every instruction cycle. If any of the masked interrupt bits are set the contents of the instruction counter will be stored in a fixed location before branching to an address kept in another fixed location. The problem of simultaneous interrupt signals is handled by selecting the left-most signal for first treatment. This is done by turning the interrupt bit off and storing its register position as an integer (0 - 23) in a third fixed location. The interrupt routine uses this interrupt number to branch to a specific service routine.

Only the instruction counter is stored as return information about the interrupted program. The interrupt service routine is responsible for saving and restoring the contents of all working registers.

The entire interrupt system may be disabled for short intervals where an interruption would be awkward (for example when the base address of the interrupt switch table is changed). When the system is disabled interrupt signals are only collected but not interrogated. The system is automatically disabled when the interrupt routine is entered. It may be enabled again (or disabled) by the monitor by the privileged instructions JUMP WITH INTERRUPT ENABLED (or DISABLED).

7.3.3. Interruption Conditions.

The interruption signals may be classified according to priority as follows: (0) Machine interruption, (1) Instruction interruption, (2) Integer interruption, (3) Floating-point interruption, and (4) - (23) External interruption.

In the event of power being turned off, or when hardware malfunction is detected a machine interruption is initiated. This interruption has the highest priority and is the only interruption signal which cannot be masked off or disabled. The interruption is performed in the following way: when the machine exception is detected the instruction counter is saved in a fixed location and the machine is stopped after activation of an alarm signal for the operator. When the start key on the console is depressed the interruption is completed by storing the interrupt number as zero and jumping to the interruption service routine. The contents of all registers (except the instruction counter) are lost, and any task program interrupted by a machine exception must therefore be completely restarted.

The execution of an unassigned operation code leads to an instruction interruption. This interruption can also be provoked by a storage operation with an undefined address, by execution of a privileged instructtion in the task mode, or by violation of the storage protection. The interrupt number is set to one before switching to the interruption routine.

An integer interruption is created by overflow occurring in integer arithmetic. The interrupt number is set to two.

A floating-point interruption with interrupt number three is created by exponent overflow or underflow occurring in floating-point arithmetic.

The remaining 20 bits in the interrupt register are assigned to external signals. Through these interruptions the computer responds to attention signals from the real-time clock, the console interrupt key, and the standard input-output devices. Up to 24 peripheral devices may be connected to each interrupt bit. Associated with each priority level is an external register of 24 bits. Each device connected to the same interrupt level has a bit position in this register. These digital registers which can be read and cleared by the computer permit identification of individual external signals.

8. Instruction set.

8.1. List of Instructions.

BASIC INSTRUCTION SET

DATA TRANSFER

- load register
- store register
- load half register
- store half register
- exchange register and store

INTEGER ARITHMETIC

- load address
- load address complemented
- add integer word
- subtract integer word
- multiply integer word
- divide integer word
- load integer byte
- add integer byte
- subtract integer byte

LOGICAL OPERATIONS

- logical and
- logical or
- exclusive or
- shift single arithmetically
- shift double arithmetically
- shift single cyclically
- shift double cyclically
- normalize single
- normalize double

SEQUENCING

- modify next address
- execute single instruction
- jump with register link
- skip if register high
- skip if register low
- skip if register equal
- skip if register not equal
- skip if register bits one
- skip if register bits zero
- skip if no exceptions
- skip if no protection

MONITOR CONTROL

- load exception register
- store exception register
- load mask register
- store mask register
- store interrupt register
- jump with interrupt enabled
- jump with interrupt disabled
- clear interrupt bits
- set protection bit
- clear protection bit
- input output

EXTENDED INSTRUCTION SET

- load double register
- store double register
- convert integer to floating
- convert floating to integer
- add floating
- subtract floating
- multiply floating
- divide floating

8.2. Definition of Instructions.

8.2.1. Notation.

The following is a formal definition of the instruction logic. The basic instruction cycle and all operations are described in pseudo-algol with the following notation:

(1) Declarations. A register declaration consists of an identifier followed by a bit size enclosed in parantheses. For example, register A(24), is a declaration of an address register A of 24 bits.

(2) Algorithms. Reference to a subfield inside a register is defined in the following way: the register bits are numbered 0, 1, 2, etc. from left to right. Bit number i in the register A is denoted A(i). The register field from bit i to bit j is described as A(i, j). Storage references to bytes and words are denoted byte[A] and word[A] respectively. In storage operations the effective address A is always a byte address. In word operations A is interpreted as a word address by ignoring the rightmost bit.

For each operation code the normal execution is described. Also listed are the setting of the exception register and the conditions that will cause a program interruption.

8.2.2. Declarations.

```
........
register A(24)        [illegible]
comment the effective address part of the current instruction;

........
register byte operand(12)
comment temporary anonymous register. Also called: shifts, selected;

........
register disabled(1)
comment defines the disabled/enabled status of the interruption system;

........
register exception(2)
comment the exception register;
```

```
. . . . . . . . .
register instruction(24)
comment the current instruction. Subfields are referred to as: F field, W
field, M field (= R bit + I bit), and D field as defined in section 3.3;

. . . . . . . . .
register instruction counter(24)
comment the address of the current instruction;

. . . . . . . . .
register mask register(24)
comment the interrupt mask register;

. . . . . . . . .
register monitor mode(1)
comment defines the monitor/task mode status of the protection system;

. . . . . . . . .
register operand(24)
comment temporary anonymous register. Also called: last A, requests, ex-
change, selected bits;

. . . . . . . . .
register prefixed address(1)
comment Boolean set by the operation Modify Next Address;

. . . . . . . . .
register W(24)
comment the working register selected by the current instruction;

. . . . . . . . .
register W last(24)
comment the working register preceding the W register i.e. with address =
W field - 1;

. . . . . . . . .
register W pair(48)
comment the register pair W last and W considered as a double-length re-
gister. The four working registers are connected cyclically as double-
length registers i.e. W0 - W1, W1 - W2, W2 - W3, and W3 - W0;

. . . . . . . . .
register X(24)
comment the index register selected by the current instruction;

. . . . .
store byte[1:storage capacity];

. . . . . .
switch operation:= Load Register, Store Register, etc...;
```

```
. . . . . . . . .
constant storage capacity
comment a fixed address defining the storage capacity;

. . . . . . . . .
constant interrupt number, return address, service address
comment fixed addresses of storage locations used by the interruption sy-
stem;
```

8.2.3. Basic Instruction Cycle.

```
    Next instruction:
          A:= instruction counter:= instruction counter + 1;
    Fetch Instruction:
          Interruption Service; Test Address;
          monitor mode:= monitor mode ∧ protection[A];
          Test Protection; instruction:= word[A];
    Decode Instruction:
          A(12, 23):= D field;
          for i:= 0 step 1 until 11 do A(i):= A(12);
          if prefixed address then
          begin A:= A + last A; prefixed address:= false end;
          if X field ≠ 0 then A:= A + X;
          if R bit = 1 then A:= A + instruction counter;
          if I bit = 1 then A:= word[A];
          goto operation[F field];
```

8.3.4. Interruption Service.

```
    . . . . . . . . . .
    procedure Interruption Service;
    begin if disabled ∨ prefixed address then goto Exit;
          requests:= interrupt register ∧ mask register;
          if requests = 0 then goto Exit;
          for i:= 0 step 1 until requests(i) = 1
          do selected:= i;
          interrupt register(selected):= 0;
          word[interrupt number]:= selected;
          word[return address]:= instruction counter;
          A:= instruction counter:= word[service address];
          monitor mode:= disabled:= true;
    Exit:
    end;
```

8.3.5. Start and Stop.

```
procedure Power on;
begin wait: if ¬ start key then goto wait;
      word[interrupt number]:= 0;
      A:= instruction counter:= word[service address];
      monitor mode:= disabled:= true;
      prefixed address:= false;
      goto Fetch Instruction;
end;

procedure Machine Exception;
begin word[return address]:= instruction counter;
      Power On;
end;
```

8.3.6. Exception Routines.

```
procedure Instruction Exception;
begin interrupt register(1):= 1; goto Next Instruction;
end;

procedure Test Mode;
begin if ¬ monitor mode then Instruction Exception;
end;

procedure Test Protection;
begin if ¬ monitor mode ∧ protection[A]
      then Instruction Exception;
end;

procedure Test Address;
begin if A < 0 ∨ A > storage capacity
      then Instruction Exception;
end;

procedure Test Integer Result;
begin exception(0):= overflow; exception(1):= lost carry;
      if overflow then interrupt register(2):= 1;
end;
```

8.3.7. Instruction Execution.

Load Register:

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

```
        Test Address; W:= word[A];
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address.

Note: When $A < 8$ the operation is equivalent to a register to register transport.

Store Register:

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

```
        Test Address; Test Protection; word[A]:= W;
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Note: When $A < 8$ the operation is equivalent to a register to register transport.

Load Half Register:

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

```
        Test Address; W(12, 23):= byte[A];
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address.

Note: When $A < 8$ the operation is equivalent to moving 12 bits from the left or right side of one register to the right side of another register.

Store Half Register:

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

```
        Test Address; Test Protection;
        byte[A]:= W(12, 23);
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Note: When A < 8 the operation is equivalent to moving 12 bits from the right side of one register to the left or the right side of another register.

Exchange Register and Store:

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

```
Test Address; Test Protection;
exchange := word[A]; word[A]:= W; W:= exchange;
goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Note: When A < 8 the operation is equivalent to exchanging the contents of two registers.

Load Address:

Load the W register with the effective address.

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

Exception: unchanged.

Interruption: none.

Note: When the same register is specified by the W and X fields the operation is equivalent to incrementing the register by the value of the D field.

Load Address Complemented:

Load the W register with the two's complement of the effective address.

```
W:= -A; goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

Note: When the same register is specified by the W and X fields and the D field is zero the operation is equivalent to changing the sign of the register.

Add Integer Word:

The storage word addressed is added to the W register, and the sum is placed in the register. The storage word remains unchanged.

```
          Test Address; W:= W + word[A];
          Test Integer Result;
          goto Next Instruction;
```

Exception: integer overflow and lost carry.

Interruption: (1) undefined address, (2) integer overflow.

Note: When $A < 8$ the operation is equivalent to a register to register addition.

Subtract Integer Word:

The storage word addressed is subtracted from the W register and the difference is placed in the register. The storage word remains unchanged.

```
          Test Address; W:= W - word[A];
          Test Integer Result;
          goto Next Instruction;
```

Exception: integer overflow and lost carry.

Interruption: (1) undefined address, (2) integer overflow.

Note: When $A < 8$ the operation is equivalent to a register from register subtraction.

Multiply Integer Word:

The W register is multiplied by the storage word addressed. The 48-bit signed product is placed in the register pair W - 1 and W. Overflow cannot occur.

```
          Test Address; Wpair:= W × word[A];
          goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address.

Note: When $A < 8$ the operation is equivalent to a register by register multiplication.

Divide Integer Word:

The register pair W - 1 and W is divided by the storage word addressed. The 24-bit signed qoutient is placed in the W register, while the remainder is placed in the preceding register. The remainder has the same sign as the dividend except when zero. An overflow is registered if the divisor is zero or if the qoutient exceeds 24 bits. In this case the result is unpredictable.

```
        Test Address;
        W:= sign(Wpair/word[A]) × entier(abs(Wpair/word[A]));
        W last:= W pair - W × word[A];
        Test Integer Result;
        goto Next Instruction;
```

Exception: integer overflow.

Interruption: (1) undefined address, (2) integer overflow.

Note: When A < 8 the operation is equivalent to a register by register division.

Load Integer Byte:

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

```
        Test Address; W(12, 23):= byte[A];
        for i:= 0 step 1 until 11 do W(i):= W(12);
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address.

Note: When A < 8 the operation is equivalent to moving 12 bits from the left or right side of one register to the right side of another register followed by an extension to 24 bits.

Add Integer Byte:

The storage byte addressed is extended towards the left to 24 bits and added to the W register. The sum is placed in the register. The storage byte remains unchanged.

```
        Test Address; operand(12, 23):= byte[A];
        for i:= 0 step 1 until 11 do operand(i):= operand(12);
        W:= W + operand; Test Integer Result;
        goto Next Instruction;
```

Exception: integer overflow and lost carry.

Interruption: (1) undefined address, (2) integer overflow.

Note: When A < 8 the operation is equivalent to adding 12 bits from the left or right side of one register to 24 bits in another register.

Subtract Integer Byte:

The storage byte addressed is extended towards the left to 24 bits and subtracted from the W register. The difference is placed in the register. The storage byte remains unchanged.

```
        Test Address; operand(12, 23):= byte[A];
        for i:= 0 step 1 until 11 do operand(i):= operand(12);
        W:= W - operand; Test Integer Result;
        goto Next Instruction;
```

Exception: integer overflow and lost carry.
Interruption: (1) undefined address, (2) integer overflow.
Note: When $A < 8$ the operation is equivalent to subtracting 12 bits from the left or right side of one register from 24 bits in another register.

Logical And:

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

```
        Test Address; W:= W ∧ word[A];
        goto Next Instruction;
```

Exception: unchanged.
Interruption: (1) undefined address.
Note: When $A < 8$ the operation is equivalent to an AND combination of two registers.

Logical Or:

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

```
        Test Address; W:= W ∨ word[A];
        goto Next Instruction;
```

Exception: unchanged.
Interruption: (1) undefined address.
Note: When $A < 8$ the operation is equivalent to an OR combination of two registers.

Exclusive Or:

The W register is combined with the storage word addressed by a logical EXCLUSIVE OR operation, and the result is placed in the register. The storage word remains unchanged.

```
Test Address; W:= W ≢ word[A];
goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address.

Note: When A < 8 the operation is equivalent to an EXCLUSIVE OR combination of two registers. When all bits in the word addressed are ones the operation is equivalent to a logical negation of the register.

Shift Single Arithmetically:

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

```
overflow:= false;
if abs(A) > 48 then A:= sign(A) × 48;
if A < 0 then
begin for i:= 1 step 1 until -A do W(1, 23):= W(0, 22);
end else
begin for i:= 1 step 1 until A do
    begin lost:= W(0);
        W(0, 22):= W(1, 23); W(23):= 0;
        overflow:= overflow ∨ lost ≢ W(0);
     end;
end;
Test Integer Result; goto Next Instruction;
```

Exception: integer overflow.

Interruption: (2) integer overflow.

Shift Double Arithmetically:

Same as Shift Single Arithmetically performed with the register pair W - 1 and W.

Shift Single Cyclically:

Shift the contents of the W register cyclically the number of places specified by the effective address A. If A is negative then rotate right, otherwise rotate left. If abs(A) > 48 the operation is suppressed and a program interruption occurs.

```
if abs(A) > 48 then Instruction Exception;
if A < 0 then
begin for i:= 1 step 1 until -A do
      begin around:= W(23); W(1, 23):= W(0, 22);
            W(0):= around;
      end
end else
begin for i:= 1 step 1 until A do
      begin around:= W(0); W(0, 22):= W(1, 23);
            W(23):= around;
      end
end;
goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) shift specification.

Shift Double Cyclically:

Same as Shift Single Cyclically performed with the register pair W - 1 and W.

Normalize Single:

Shift the contents of the W register left with zero insertion until bit 0 is different from bit 1. The number of shifts performed is stored as a negative integer in the storage byte addressed. If W = 0 the number of shifts is set to -2↑11.

```
Test Address; Test Protection;
if W = 0 then shifts:= -2↑11
else begin for shifts:= 0, shifts - 1
           while W(0) = W(1) do
           begin W(0, 22):= W(1, 23); W(23):= 0 end;
     end;
byte[A]:= shifts; goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Normalize Double:

Same as Normalize Single performed with the register pair W - 1 and W.

Modify Next Address:

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

```
        last A:= A; prefixed address:= true;
        goto Next Instruction;
```

Exception: unchanged.

Interruption: disabled until the next instruction has been executed.

Note: A sequence of modify next instructions like the following:

```
        modify next (word address)
        modify next (0)
        modify next (0)
        etc...
```

is equivalent to multilevel indirect addressing.

Execute Single Instruction:

Fetch the instruction specified by the effective address and execute it as if it had been located where the EXECUTE instruction is.

```
        Test Address; goto Fetch Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address, protection violation, or privileged instruction.

Note: When the instruction fetched has relative addressing specified the effective address will be interpreted relatively to the location of the execute instruction. There are no restrictions on the type of instruction fetched (except those imposed by the protection system). It may be another execute instruction, or it may be a jump instruction preventing the program from continuing with the following instruction.

Jump with Register Link:

If the W field $\neq$ 0 the instruction counter is stored in the W register. Following this a jump is made to the effective address.

```
Test Address; Test Protection;
if W field ≠ 0 then W:= instruction counter;
goto Fetch Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Note: When the W field = 0 the operation is equivalent to a simple unconditional jump which leaves all working registers unchanged. When the W field ≠ 0, the operation is a subroutine jump which leaves the return address in the W register. A return jump is performed as a simple jump with the same register specified in the X field.

<u>Skip if Register High:</u>

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

```
if W > A then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

<u>Skip if Register Low:</u>

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

```
if W < A then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

<u>Skip if Register Equal:</u>

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

```
if W = A then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

Skip if Register Not Equal:

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

```
if $W \neq A$ then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

Skip if Register Bits One:

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

```
selected bits:= $\neg W \wedge A$;
if selected bits = 0 then
instruction counter:= instruction counter + 1;
goto Next Instruction.
```

Exception: unchanged.

Interruption: none.

Skip if Register Bits Zero:

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

```
selected bits:= $W \wedge A$;
if selected bits = 0 then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

Skip if No Exceptions:

Use the right-most two bits of the effective address as a mask to test the exception register. If the selected exception bits are zero then skip the following instruction. The exception register remains unchanged.

```
selected bits:= exception ∧ A(22, 23);
if selected bits = 0 then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: none.

Skip if No Protection:

Interrogate the protection bit of the storage word addressed. If it is zero then skip the following instruction.

```
Test Address;
if ¬ protection[A] then
instruction counter:= instruction counter + 1;
goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address.

Load Exception Register:

Insert the right-most two bits of the storage byte addressed into the exception register. This is a privileged instruction.

```
Test Mode; Test Address;
byte operand:= byte[A];
exception:= byte operand(10, 11);
goto Next Instruction;
```

Exception: Set as defined above.

Interruption: (1) undefined address or privileged instruction.

Store Exception Register:

Store the exception register in the right-most two bits of the storage byte addressed. The left-most ten bits of the storage byte are set to zero.

```
        Test Address; Test Protection;
        byte operand(0, 9):= 0;
        byte operand(10, 11):= exception;
        byte[A]:= byte operand;
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Load Mask Register:

Insert the storage word addressed in the interrupt mask register. Bit 0 of the mask register is always set to one. This is a privileged instruction.

```
        Test Mode; Test Address;
        mask register:= word[A];
        mask register(0):= 1;
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or privileged instruction.

Store Mask Register:

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

```
        Test Address; Test Protection;
        word[A]:= mask register;
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Store Interrupt Register:

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

```
        Test Address; Test Protection;
        word[A]:= interrupt register;
        goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or protection violation.

Jump with Interrupt Enabled:

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

```
Test Mode; Test Address; disabled:= false;
if W field ≠ 0 then W:= instruction counter;
goto Fetch Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or privileged instruction.

Jump with Interrupt Disabled:

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

```
Test Mode; Test Address; disabled:= true;
if W field ≠ 0 then W:= instruction counter;
goto Fetch Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or privileged instruction.

Clear Interrupt Bits:

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

```
Test Mode;
interrupt register:= interrupt register ∧ ¬ A;
goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) privileged instruction.

Set Protection Bit:

The protection bit of the storage word addressed is set. This is a privileged instruction.

```
Test Mode; Test Address;
protection[A]:= true;
goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or privileged instruction.

Clear Protection Bit:

The protection bit of the storage word addressed is cleared. This is a privileged instruction.

```
          Test Mode; Test Address;
          protection[A]:= false;
          goto Next Instruction;
```

Exception: unchanged.

Interruption: (1) undefined address or privileged instruction.

Input Output:

Executed as described in section 6.3. This is a privileged instruction.

Exception: device disconnected and device busy.

Interruption: (1) privileged instruction.

Appendix A: Project Pulawy.

A1. Process Control Tasks.

The prototype of the RC 4000 computer is intended for installation in a chemical plant constructed by Haldor Topsøe in Pulawy, Poland. The plant is composed of 13 wide-spread units: 3 gas preparation units, 3 ammonia units, 4 nitric acid units, and 3 ammonia nitrate units. The end product consists of bags with ammonia nitrate. The plant is operated manually by operators under supervision of the computer. The main tasks of system RC 4000 as defined by Haldor Topsøe are: +)

(1) Alarm Monitoring.

Every 5 minutes about 350 process variables will be scanned and checked against prescribed alarm-limits. Alarm conditions are brought to the attention of the operators by audible and visible alarm signals followed by a print-out of the process variables to be corrected. Under alarm-free conditions the variables will be scanned at the rate of 10 - 20 points per second, and the complete scan thus lasts 17 - 35 seconds. In case of alarms the scanning rate drops to 1 point per second limited by the alarm strip printer.

(2) Data Logging.

Every hour a record of about 700 process variables is typed out at a rate of 1 - 2 points per second. The entire scan therefore lasts 6 - 12 minutes.

(3) Process Evaluation.

An important function of the computer is to perform the guarantee test of the entire plant by providing management with regular information about production and consumption figures.

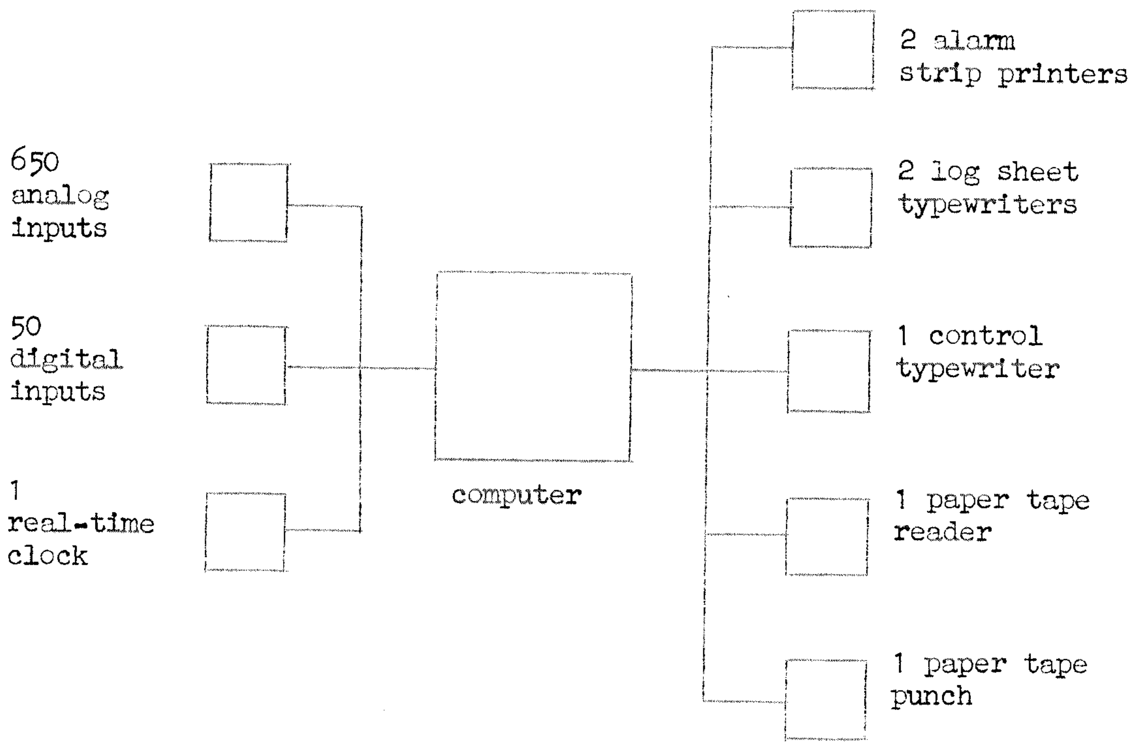
+) For a more detailed description please refer to: John Saietz, Z.A. Pulawy II, Preliminary Description of the Data-logger Installation, Haldor Topsøe, File 580 K, June, 1965.

(4) Self-checking.

From the above it is clear that the RC 4000 exercises supervisory rather than direct control of the plant since all process corrections are made manually by operators. In the event of hardware malfunction or programming errors the plant can still be controlled manually while the computer system is repaired. The minimum safety demand is that the computer is able to detect and report such malfunction. In idle intervals the computer will therefore perform self-checking of basic hardware functions such as the instruction logic, the core store, and all addressable devices and registers.

A2. System Configuration.

The following figure shows the configuration of peripheral devices as defined by Haldor Topsøe:



(1) Analog Inputs.

An analog-to-digital converter is connected to about 650 input signals derived from such sources as thermocouples, resistance thermometers, pH analyzers, and flow transducers. The selection of an input point is performed by a relay multiplexer with a switching rate of 10 - 20 points per second.

(2) Digital Inputs.

Discrete events such as the number of bags with end product and the kilowatthours consumed are counted in 1-bit registers. These registers must be read and cleared by the computer every second. The on-off status of various contacts such as alarm indicators and operator switches will also be registered and sensed by the program.

(3) Real-time Clock.

The real-time clock provides a time statement for the printed records and initiate task programs at various preset intervals.

(4) Alarm Printers.

Two strip printers (1 line per second) are provided for the printing of alarm values.

(5) Typewriters.

Two typewriters (10 characters per second) print out the log sheets. A third typewriter is used for communication with the operator.

(6) Paper Tape I/O.

Programs and process constants are input from a paper tape reader (100 characters per second) and output on a paper tape punch (20 characters per second).