RCSL No: 52-AA1192

Edition: October, 1983

Author: Jens Kristian Kjærgård

Title:

RC3502/2 Reference Manual

A/S REGNECENTRALEN
af 1979

Keywords:

Mini Computer, Stack Machine, Multiprogramming, Communication, Real Time Control, Microprogrammed Bit Slice Processor, Modularity, Double Europe Cards, RC3502, RC3502/2.

Abstract:

This is the reference manual for the RC3502/2 minicomputer, which is a stack machine supporting a high level language with multiprogramming and communication facilities. The RC3502/2 is designed primarily for real time control applications.

(354 printed pages).

Printed by A/S Regnecentralen af 1979, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

APPENDICES:

1. INTRODUCTION

The present publication is the Reference Manual for the RC3502/2 Processing Unit, the building block of the RC3502 Minicomputer System. The RC3502 is designed primarily for real-time control applications. Thus the RC3502 is used, for example, as a terminal concentrator, a front-end processor for general-purpose computers, and a node in packet-switching networks and other communication systems. The RC3502 is programmed in the high-level language Real-Time Pascal (PASCAL80).

The rest of this chapter summarizes the main characteristics of the RC3502 minicomputer system.
Chapter 2 contains an overview of the processing unit hardware.
Chapter 3 describes the runtime environment in which RC3502 instructions are executed.
Chapter 4 the instruction fetch.
Chapter 5-14 describes the base instruction set.
Chapter 15 switches and indicators.
Chapter 16 the debug console.
Chapter 17 the actual instruction set.
Chapter 18 gives the instruction execution times.

1.1 Processing Unit

The processing unit contains a 16-bit ALU. The processor can address up to 4M bytes of memory. The processing unit has 128 interruption levels. Each interruption level can be connected to a context set. The instruction set is stack oriented and supports communicating parallel process incarnations. A number of input/output instructions control data transfer between peripherals and the processor.

1.1.1 Context Sets

The processing unit has 122 context sets. A context set contains a register set of eight 16-bits registers, and a register stack of eight 16-bits stackelements and a 4-bits stackpointer. The register set defines a number of memory references, namely, to a process incarnation stack and to the current instruction in the program being executed by the incarnation.

1.1.2 Scheduling and Interruption System 1.1.2

A context switch from one process incarnation to another is reduced to the connection of a new context to the control and arithmetic unit. Each time an instruction is executed, the context set associated with the interrupt signal on the level with the highest priority is selected as the current context, and the instruction pointed out by the register set is executed as the next instruction.

The lowest interruption level (level 0) is shared between a number of process incarnations. These incarnations are partitioned into three priorities:

- coroutine priority
- high timeslice priority
- low timeslice priority

1.1.3 Instruction Set 1.1.3

The design objective was to ensure that operations which are time-critical in realtime applications would be supported efficiently by the instruction set. These operations include data manipulation, procedure entry and exit, and communication between process incarnations.

The machine instructions can be divided into three groups:

o Those which support Real-Time Pascal and similar high-level procedural languages (i.e. this group of instructions specifically supports stack processing).

- Those which support context switching and communication between process incarnations.

- Those which support the runtime system, I/O, and operations necessitated by the underlying hardware.

The instructions have a varying length, namely, one byte for the operation code and zero, one, or more bytes for parameters. The functionality of the instruction set is specified by the Base Instruction Set, which contains the instructions in their longest form. In order to conserve space and increase efficiency, the Actual Instruction Set was developed on the basis of extensive statistical analyses of the way in which large application systems actually use the instructions and their parameters. The Actual Instruction Set is obtained by adding to the Base Instruction Set a subset of the latter in compactly encoded form (i.e. with fewer parameters).

1.1.4 Input/Output 1.1.4

Input/output is the transfer of data between a peripheral and the physical memory addressable by the processing unit.

The actual transfer is basically performed in two modes:

1. Programmed Input/Output

Here the transfer is performed by the processing unit executing input/output instructions. The instructions transfer a single byte, a single word, or a block of bytes or words between the peripheral and the address space, possibly directed by interrupt signals from the peripheral. The peripheral may interrupt once for every byte or word transferred.

2. Direct Memory Access (DMA)

The transfer of data between the peripheral and the address space is performed by a controller without interrupting the processing unit. The transfer is initiated by the processing unit. The controller interrupts the processing unit at the end of the (block) transfer. The controller access the address space by cycle stealing via the backplane bus.

3. Dual-Port Memory

Is used as an alternative to DMA. The main difference is that the processing unit must move data from the controller removing the controller's need to be able to perform DMA-cycles on the backplane bus.

1.1.5 System Start-Up 1.1.5

System start up can be initiated manually by an operator, automatically from hardware, or from software. Controlled by switches, the built-in memory and processing unit tests are executed. The registers are initialized, and a jump is made to a boot program residing in PROM. This program autoloads programs from an external device selected by switches, and includes programs residing in other PROM modules.

1.2 Debugging System 1.2

A control microprocessor makes it possible to inspect and modify the memory and registers of the processing unit and to control the instruction execution of the processing unit.

Furthermore the control microprocessor is connected to the processor front panel containing five switches, five indicators, and a jack.

The switches control the autoload procedures, the built-in test programs, and the speed of the debug console.

The current status of the processing unit is displayed on the indicators.

The jack makes it possible to connect a Teletype compatible terminal to the system either locally or remotely via a modem.

1.3 Summary of the Hardware System 1.3

This section describes the most important hardware characteristics of the RC3502 Processing Unit.

1.3.1 Basic Physical Unit 1.3.1

16-bit processor on 3 circuit boards, backplane bus, power supply, and crate with 16 free circuitboard slots for additional hardware modules.

1.3.2 Processing Unit Architecture 1.3.2

o 16-bit arithmetic-logic unit (ALU) built around four AM2901A bit-slice chips.

o Up to 4M bytes of directly addressable memory. The basic memory unit is an 8-bit byte. The processing unit provides operations for manipulating single bit(s) within a byte or word.

o 122 register sets. Each register set associated with a process incarnation.

o The I/O system supports character-oriented and block-oriented peripherals. Programmed I/O between a peripheral and any memory module can be performed, using, for example, 8-channel I/O modules. DMA I/O between a controller and memory can be performed on a cycle stealing basis via the backplane bus or via a dual ported memory situated in the controller.

o A Teletype compatible device can be connected as both a debug and an operator console to the front panel of the processing unit for communication with the control microprocessor (Intel 8085A) on the internal data bus.

1.3.3 Processing Unit Instruction Set 1.3.3

- Stack-oriented instruction set.

- Arithmetic operations with twos complements.

- Instruction format: 1 byte operation code followed by 0, 1, or more bytes as instruction parameters. Operands, moreover, can be located elsewhere in memory (e.g. in the incarnation stack).

- Base Instruction Set comprises instructions, including: push and pop operations; procedure call and return; unconditional jumps, case jump, and conditional jumps; monadic and dyadic operators and operations on sets; indexing of arrays and monitor control and synchronization, including signal and wait operations on queue semaphores.

- I/O instructions include: read status, write control, read/write word, initialize block transfer, read/write block of bytes/words, and clear current interrupt.

- Addressing: direct, local stack frame, global stack frame, an intermediate stack frame, or address on the stack.

2. HARDWARE OVERVIEW 2.

The RC3502 processing unit is microprogrammed and built around an internal data bus as shown in figs. 1-3. This bus is interfaced to:

- the 16-bit arithmetic-logic unit based on four AM2901 bit slice chips
- the control microprocessor
- the register files
- the register stacks
- the prefetch unit
- the interrupt and schedule unit
- the backplane interface.

The flow on the internal data bus is controlled from the RC3502 microprogram, which implements the various features of the processing unit. The execution of the microprogram is controlled by the microsequencer an AM2910 chip.

The microprogram is physically contained in a read-only memory of 2048 60-bit words. The execution time per microinstruction is 217 nanoseconds.

2.1 The Microsequencer 2.1

This unit selects the next microinstruction to be executed. For further information is referred to ref. [1].

2.2 The Arithmetic-Logic Unit 2.2

This unit performs all datamanipulations such as addition, subtraction etc. More complex operations are performed in conjunction with the microsequencer as a sequence of operations i.e. multiplication, division etc. The unit contains sixteen 16-bits registers used for temporary results during execution of a single RC3502 instruction.

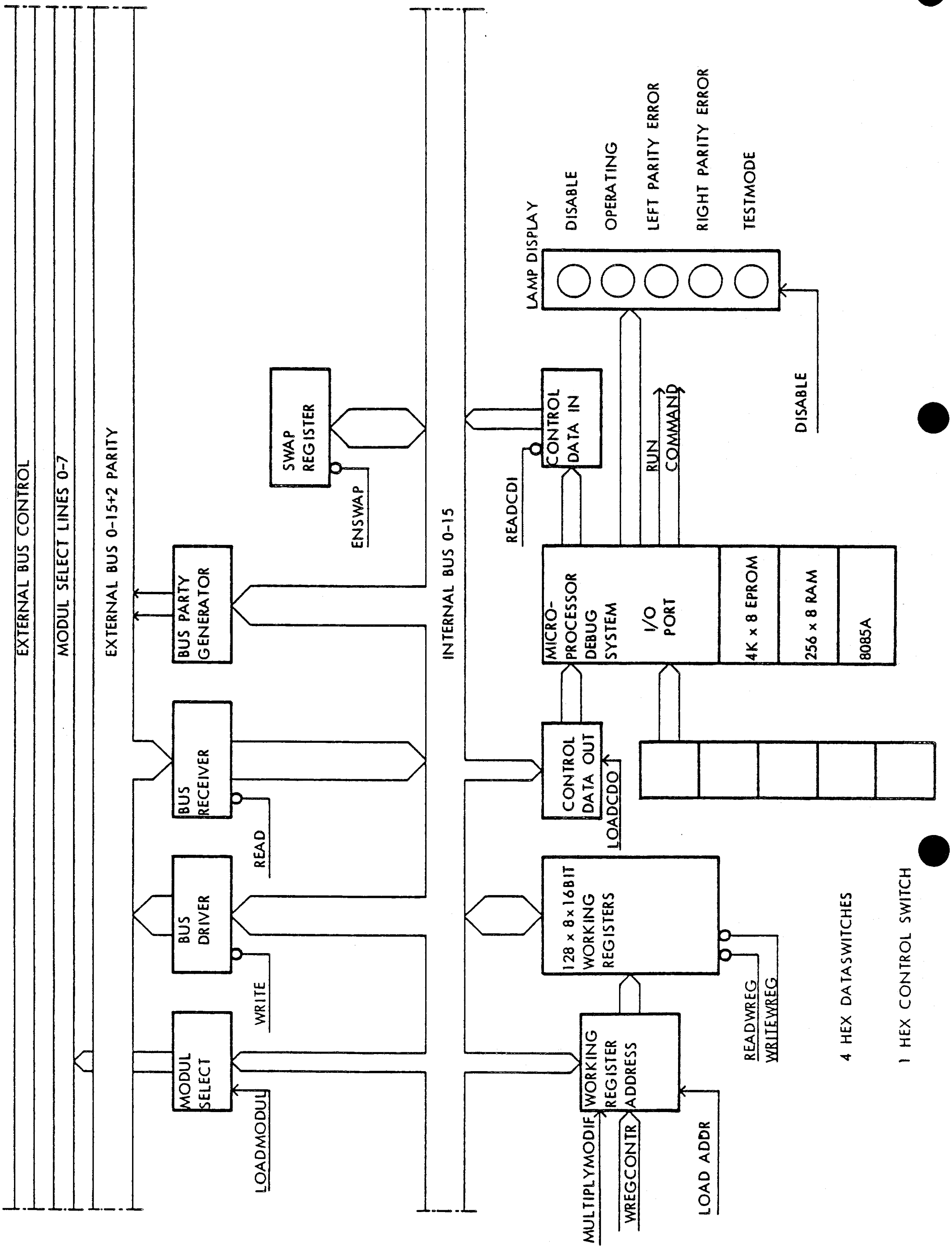


Figure 1: Processing Unit blockdiagram (part 1).

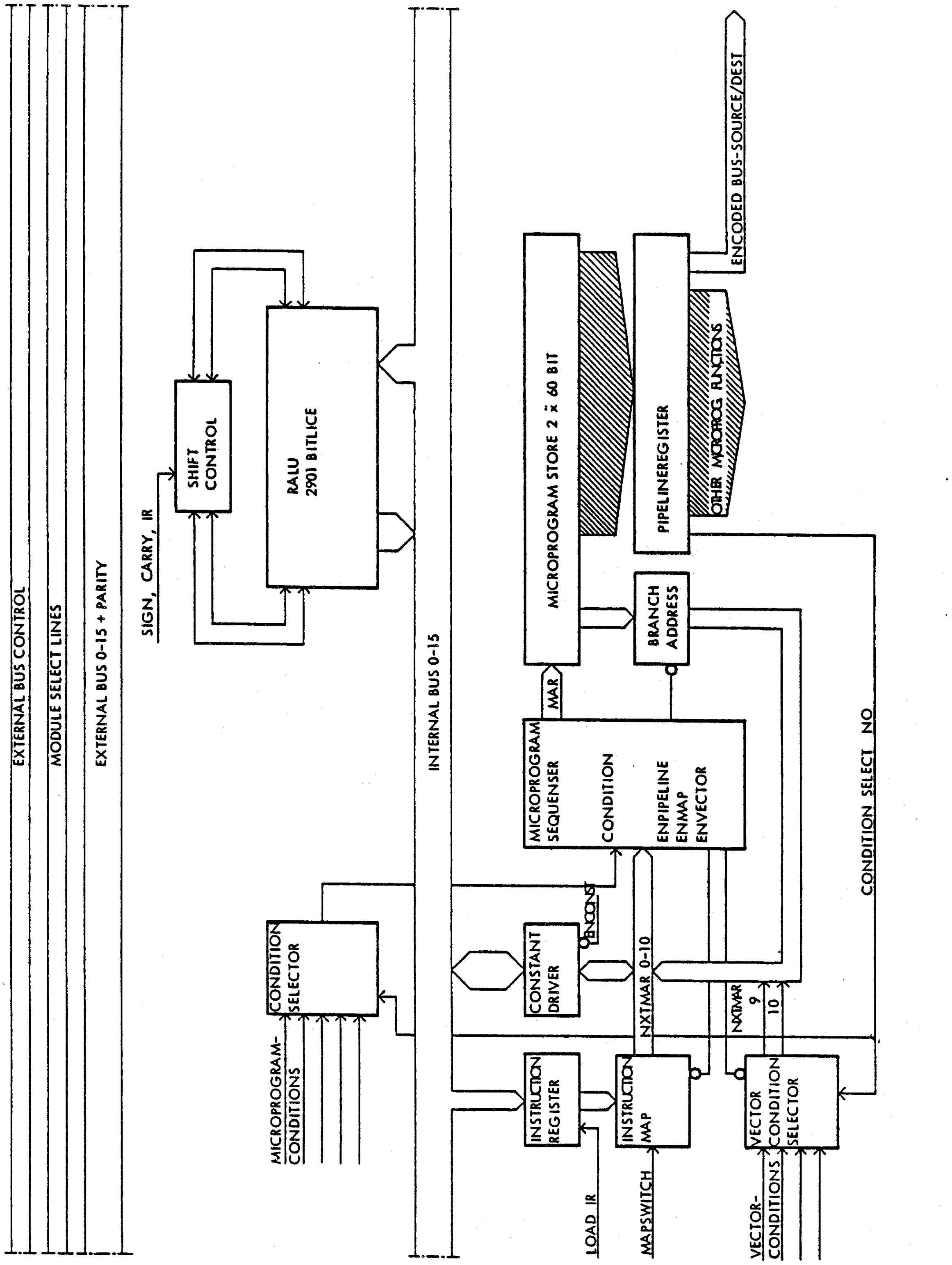


Figure 2: Processing Unit blockdiagram (part 2).

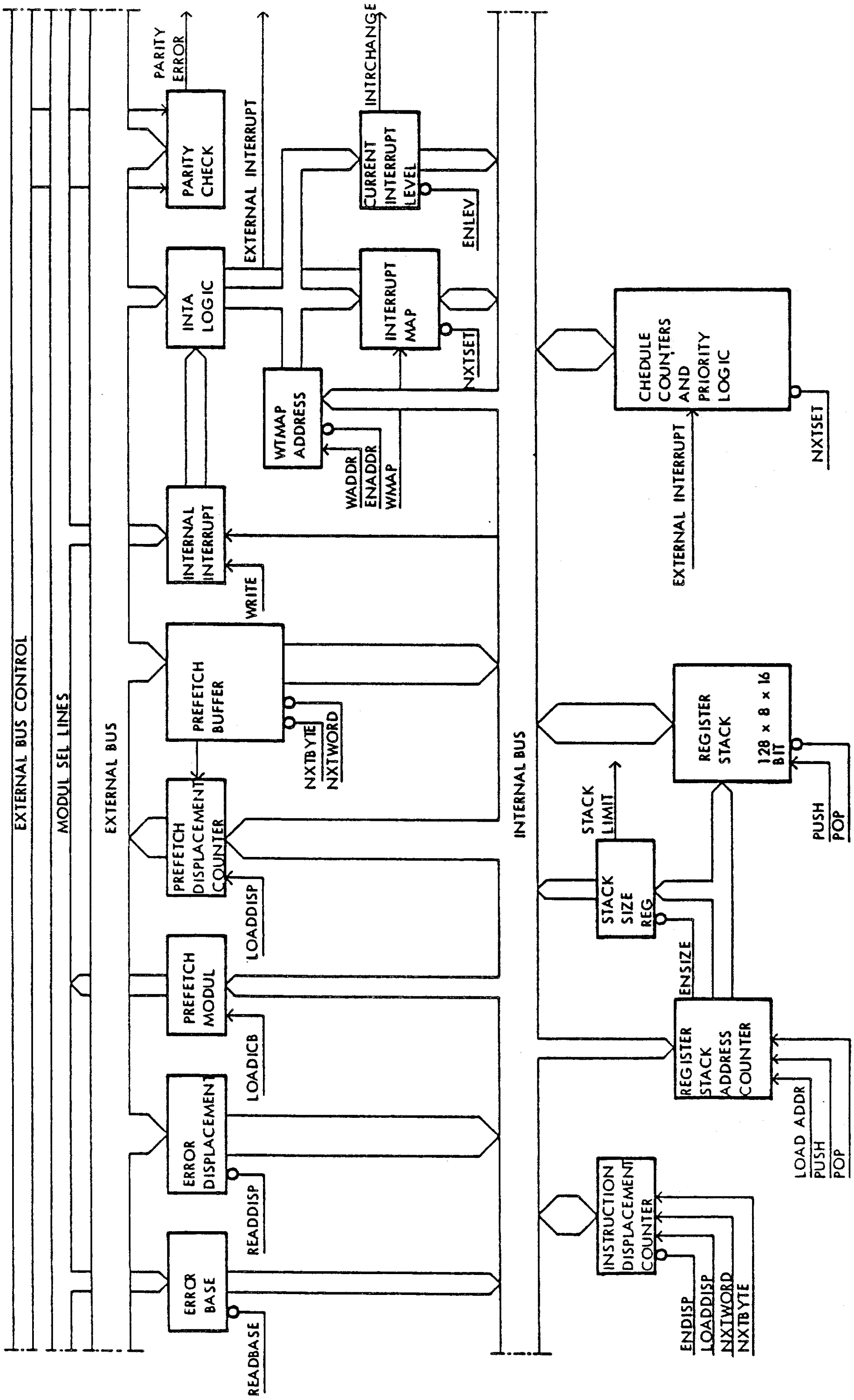


Figure 3: Processing Unit blockdiagram (part 3).

2.3 Control Microprocessor 2.3

An Intel 8085A microprocessor system is connected to the internal data bus of the processing unit. The microprocessor interfaces the switches, indicators, and console jack described in chapter 15 to the processing unit.

If a Teletype (TTY) compatible device is connected to the microprocessor system, the former can be used as a debug console for the processing unit, enabling the operator to examine and modify the contents of memory locations and working registers as described in chapter 16.

The TTY can also be used as an operator console, by employing instructions that work on the micromachine RAM.

2.4 Register Sets 2.4

The RC3502 processing unit contains 122 register sets, each consisting of eight 16-bits words representing a process incarnation. The register files are physically placed in a register array of 1024 16-bits words. The remaining words (1024 - 122*8 = 48) are used as working area for the micromachine.

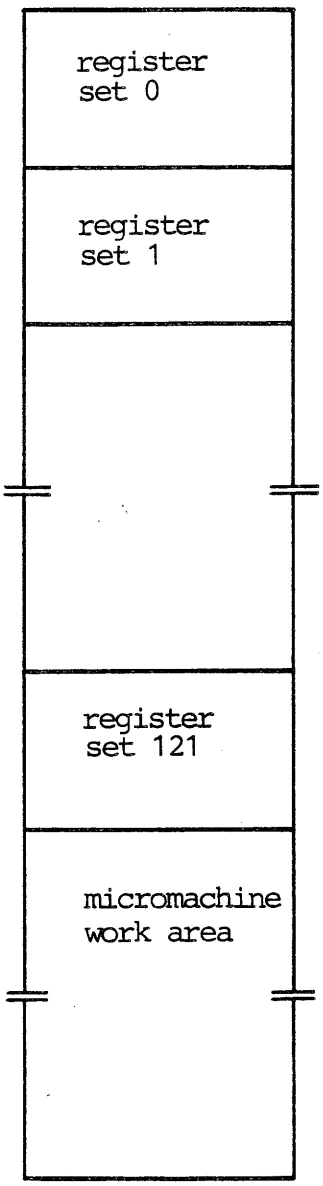


Figure 4: Register array.

2.5 Register Stacks 2.5

To each register set corresponds a register stack, which is an eight element wordstack with a stackpointer. The stackpointer is able to represent a contents of 0 to 8 words in the stack as well as an underflowed or overflowed stack. The later two states are further indicated by a condition to the microsequencer.

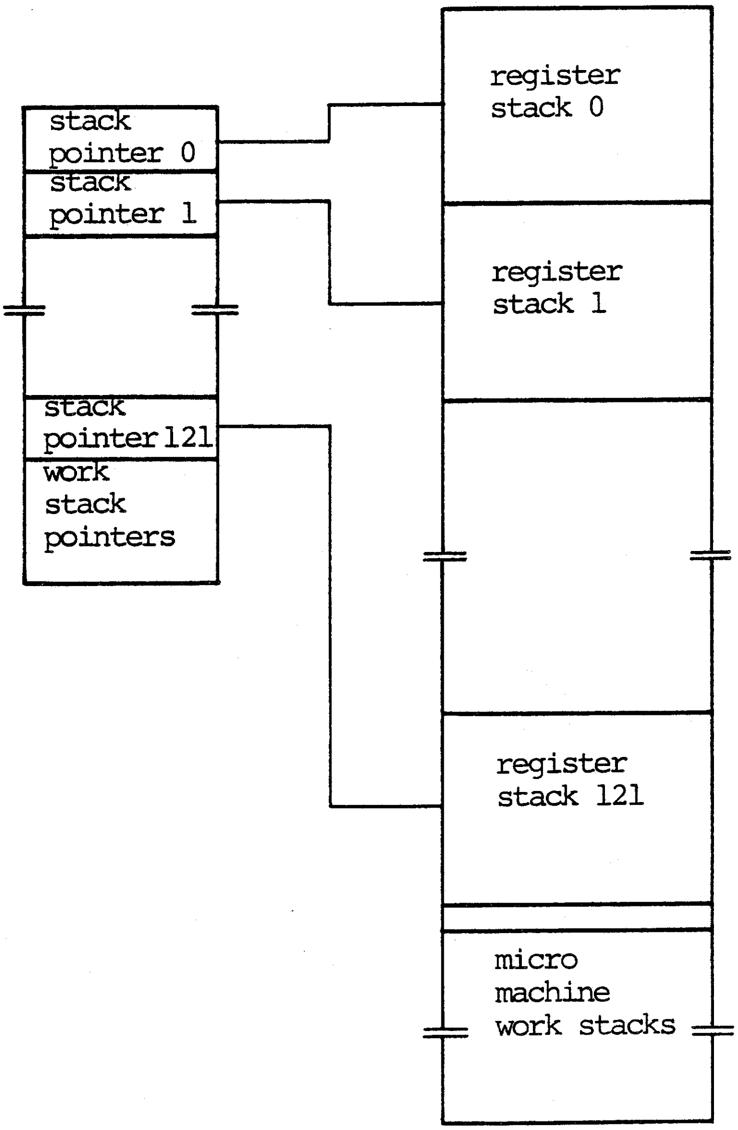


Figure 5: Register stacks.

2.6 The Prefetch Unit 2.6

A prefetch unit, acting as a DMA-device, fetches instructions and parameters ahead of their usage. The prefetch unit accesses memory in wordmode but is able to deliver both 8-bit bytes and 16-bits words independent of word boundaries. Further the prefetch unit automatically increases the instruction counter by one when a byte is retrieved and by two when a word is retrieved. The execution of jump instructions provides that the current value of the instruction counter can be read and a new value loaded. Likewise, the current instruction counter is saved in the current registerset before changing context, and the value of the instructioncounter in the new registerset loaded.

2.7 Interrupt and Schedule Unit 2.7

The function of this unit is to select the next process incarnation to be executed. Prior to every execution of a new instruction the interrupt and schedule unit is examined for the occurrence of an event requesting a new incarnation to be scheduled for execution. Some instructions such as blockmoves, block-I/O etc. can be interrupted during execution of a single instruction and a temporary state saved to allow continuation of the instruction.

There exists 128 interruption levels of which the lowest level (level 0) is shared between a number of process incarnations. If an interrupt has occurred at a given level (> 0) the process incarnation continues execution until either an interrupt at a higher level occurs, the process incarnation performs a wait instruction, or selects a lower level. The highest interruption level is found through the backplane bus, which by means of a daisy- chain request the I/O controller containing the highest interruption level to place its interruption level on the backplane bus.

This level is further mapped upon the actual context through an interruption map, which maps all of the 128 interruption levels upon one of the 122 context sets.

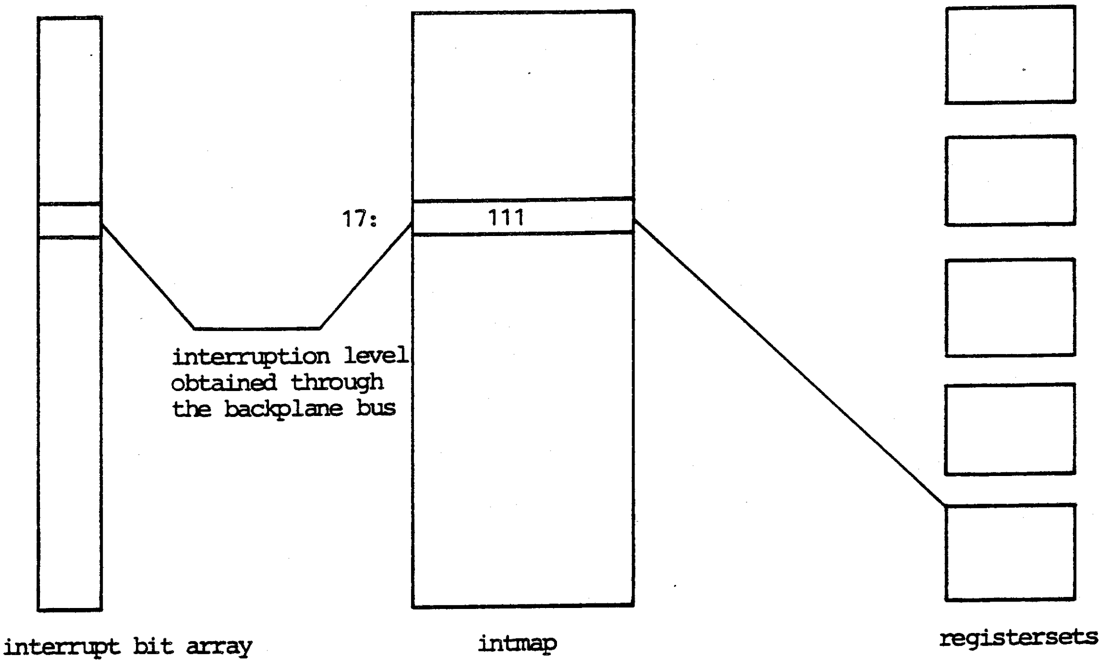


Figure 6: Registerset selection.

Level 0 is shared by a number of process incarnations. Level 0 process incarnations are partitioned into three priorities. If two incarnations with different priority are able to execute, the one with the highest priority is selected.

priority 0: Coroutine priority.
When a process incarnation with this priority is scheduled for execution, no other level 0 process incarnation will be executing before the executing incarnation executes a wait instruction.

priority -1: Highest timeslice priority.
Process incarnations in this class are scheduled according to a round-robin timeslice strategy.

priority -2: Lowest timeslice priority.
As priority -1, but only selected if no incarnation with priority -1 is able to execute.

2.8 The Backplane Bus

All memory and I/O transfers are communicated through the backplane bus.

The backplane bus can be partitioned into:

- 30 control lines
- 16 address/data lines:
  a 16-bit bidirectional bus used for multiplexed address and data information
- 8 module select lines:
  8 lines used to select the various modules connected to the backplane bus.

The values of the module select lines are used as follows:

00
.
. I/O devices
.
7F

80 special purpose memory area (i.e. used for
. memory of dual ported I/O controllers
.
.
9F

A0
.
. PROM or RAM memory modules
.
FF

A memory module occupies two module values, one for reading from memory and one for writing to memory, but is usually referred to by the lowest value, i.e. memory module C0 uses module value C0 for read and C1 for write. This is utilized by using the least significant bit to indicate an undefined address (a nil address).

Memory references are supervised by a parity checker, which checks all data transferred to or from memory. If a parity error is found or no memory module answers, the processing unit is stopped and an error message displayed at the debug console.

2.9 Input/Output 2.9

The RC3502 has three forms of input/output:

1) Serial transfer using I/O modules (i.e. circuit boards for 8 I/O channels each).

2) DMA (direct memory access) transfer directly via the backplane bus.

3) Dual-port memory on an intelligent controller.

2.9.1 Serial Transfer 2.9.1

A peripheral device is connected to the processing unit by means of a 4 pairs cable, which is transformer coupled at both ends. This ensures high noise immunity.

The cable is connected to a serial controller in the RC3502. The following description includes this serial controller in the RC3502 processing unit, since it is reflected in the instruction set.
The transfer of data is performed in serial mode regardless of whether the connected device is serial or parallel. The data word transferred consists of a 4-bit header and from 0 to 16 data bits:

HEADER				DATA
1			1	(0 to 16 data bits)

The header contains the following information:

TRANSFER FROM PROCESSING UNIT TO DEVICE

Output Header	1	0	0	1	read data
	1	0	1	1	read status
	1	1	0	1	write data
	1	1	1	1	write control

TRANSFER TO PROCESSING UNIT FROM DEVICE

Output Header	1	0	0	1	16 bits (word)
	1	0	1	1	EOI (end of information)
	1	1	0	1	8 bits (byte)
	1	1	1	1	not used

In principle each device contains four 16-bit registers: one status register, one control register, and two data registers (one for each direction of the flow). In a given device, however, one or more of these registers may be omitted, some may be combined into one register, or the register size may be reduced from 16 bits right down to 1 bit. The processing unit initiates an I/O instruction by selecting the I/O cable leading to the device addressed, whereupon the data is transferred.

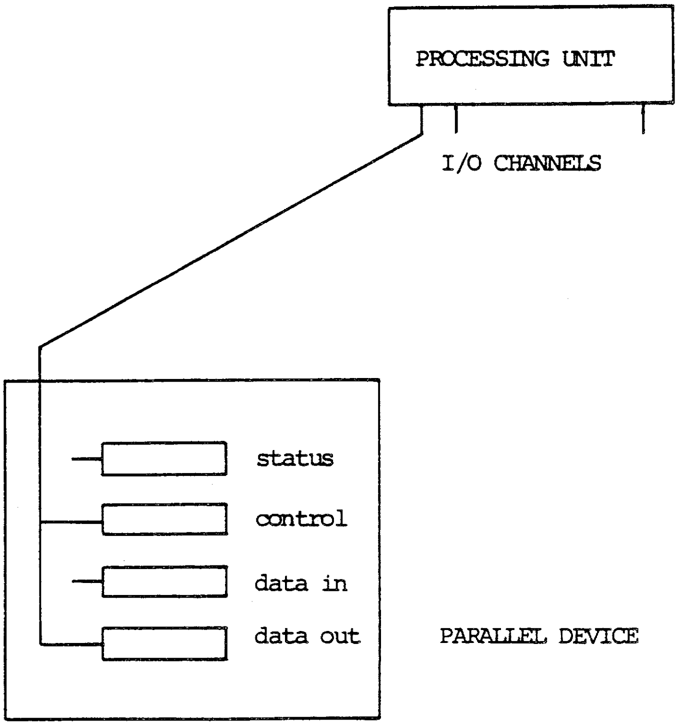


Figure 7: Device connected by I/O Modules.

Communication may be initiated either by a program or by an interrupt from a device. An interrupt is detected on the data line when a 1 bit is sent to the processing unit and it has not requested data.

The data flow between the processing unit and the four device registers is illustrated by four general I/O commands: read status, write control, read data, and write data. These instructions describe the possible pattern of execution for the I/O channels and the device controllers. Specific details about the storage of data are given in chapter 10, where the I/O instructions are defined.

2.9.2 DMA Transfer 2.9.2

By using the DMA facilities in the backplane bus, an interface with a high transfer rate, e.g. a high-speed communication controller, can transfer data directly between the peripheral equipment and the RC3502 memory. When the interface wishes to use the backplane bus, it sends a request to the processing unit, which releases the bus at a suitable moment. The interface then has access to all backplane signals.

2.9.3 Dual-Port Memory 2.9.3

To eliminate the need for requesting the bus, it is possible to use dual-port memory on a controller, i.e. both the RC3502 processing unit and the controller are able to read and write in this memory. The RC3502 processing unit is not interfered with when the controller uses the dual-port memory.

3. RUNTIME ENVIRONMENT 3.

In this chapter the runtime environment, in which RC3502 instructions are executed, is described. The actual micromachine operations are described as well as the notation used in the following chapters describing the functionality of the instruction set.

3.1 Basic Formats 3.1

The basic memory quantities in the RC3502 are the 8-bit byte, and the 16-bit physical word.

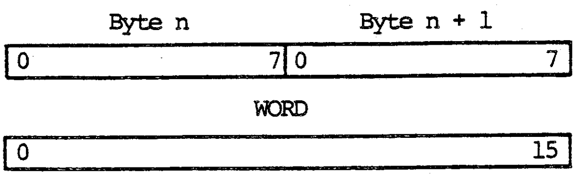


Figure 8: Byte and word.

The bits in a byte are numbered 0..7 with bit 0 as the most significant bit. The bits in a physical word are numbered 0..15 with bit 0 as the most significant bit.

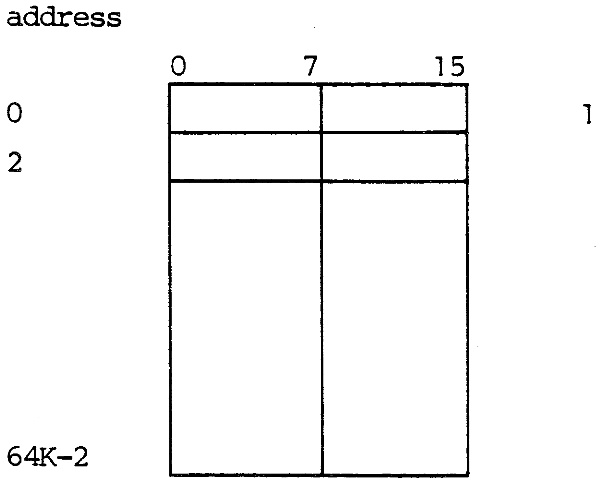


Figure 9: A memory module.

A memory module contains 64K bytes. A physical word occupies two consecutive bytes, where the address of the lowest addressed byte must be even. The address of a physical word is the address of the lowest addressed byte.

Upon these basic quantities the following types are built.

```
byte = 0..255;
   The denotation of an 8-bit byte.

word = 0..65535;
   As opposed to the physical word, this logical 16-bit quan-
   tity is just the concatenation of two bytes, and can be
   placed in two consecutive physical words. The address of a
   word is the address of the lowest addressed byte (the first
   byte).

integer = -32768..32767;
   The signed perception of a word.

bit = 0..1;
   Denotes a single bit within a byte or word. A bit must
   always be referred to through the byte or word containing
   the bit, i.e. as an element of a packed record.

basetype = packed record
                    onebit: bit;
                    module: 0..63;
                    nilbit: bit;
                  end;
   Used to designate a memory module. Onebit must always be 1.
   Module is the actual module number. If nilbit equals one an
   undefined module is designated.

adr = record
        base: basetype;
        disp: word;
      end;

addr = record
         nullbyte: byte;
         base: basetype;
         disp: word;
       end;
```

Designates a memory address. The first byte in the declaration of addr is usually dummy and mostly used in structures, where wordalignment is mandatory.

(An object is said to be word aligned, if it is assumed that the address of the first byte of the object is even.)

```
   double = record
                 w1  : word;
                 w2  : word;
              end;
```

The detailed description of the layout of variables is found in ref. [2].

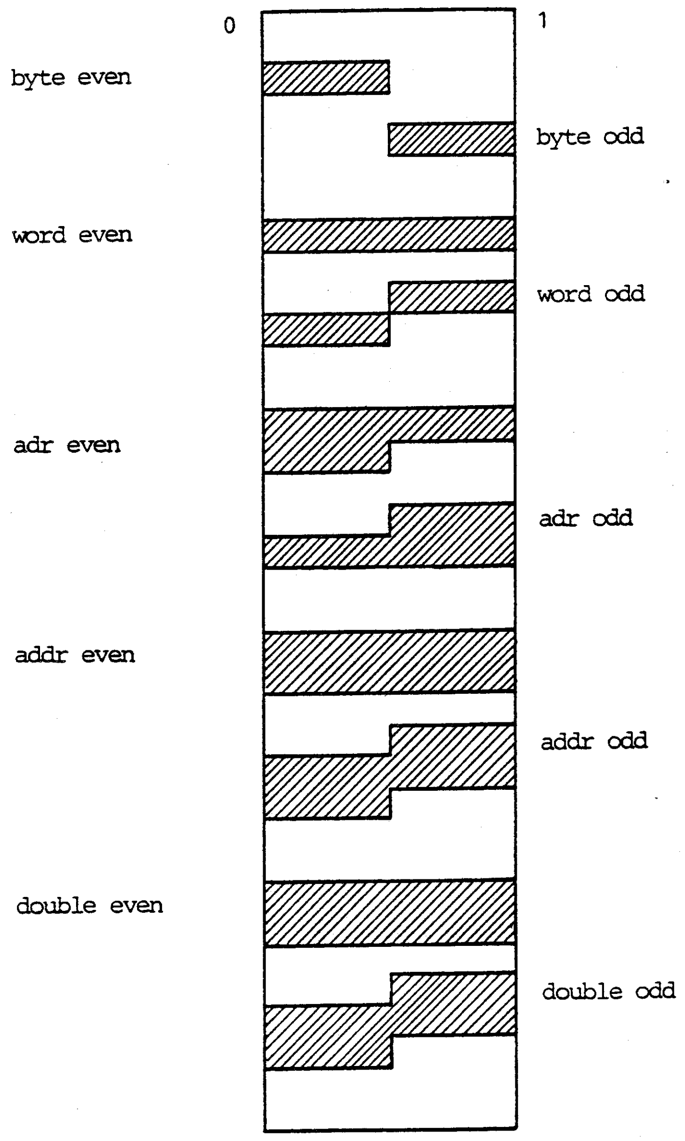


Figure 10: Memory layout for various types.

The following notations are used when referring to the contents of a memory structure.

membyte (a: adr)

Designates the contents of a byte.

memword (a: adr)

Designates the contents of a word.

mem (a: adr)

Designates the contents of a physical word and is only used where the word referred is known to be word aligned.

memadr (a: adr)

Designates the contents of an adr.

memaddr (a: adr)

Designates the contents of an addr.

memdouble (a: adr)

Designates the contents of a double.

3.2 Incarnation Stacks 3.2

The execution of an RC3502 machine instruction presupposes a certain environment. An important part of this environment is a number of stacks, one for each process incarnation. The stack for a process incarnation is allocated as a consecutive number of bytes within a single memory module.

A stack contains one stack frame for each uncompleted routine call. A stack frame is a number of consecutive storage locations. The stack frame contains the parameters and the local variables for the routine call.

A stack frame includes the following areas:

1) Actual parameters

This area contains the values or the addresses of the actual parameters for the routine call.

2) Anonymous parameters

This area contains the information needed to access non-local objects from the body of the routine (static link) and to return from the routine call to the point of call (dynamic link and return address). The layout of the anonymous parameters is shown in chapter 7.

3) Local objects

This area contains the storage locations of the objects (e.g. variables) declared in the body of the routine.

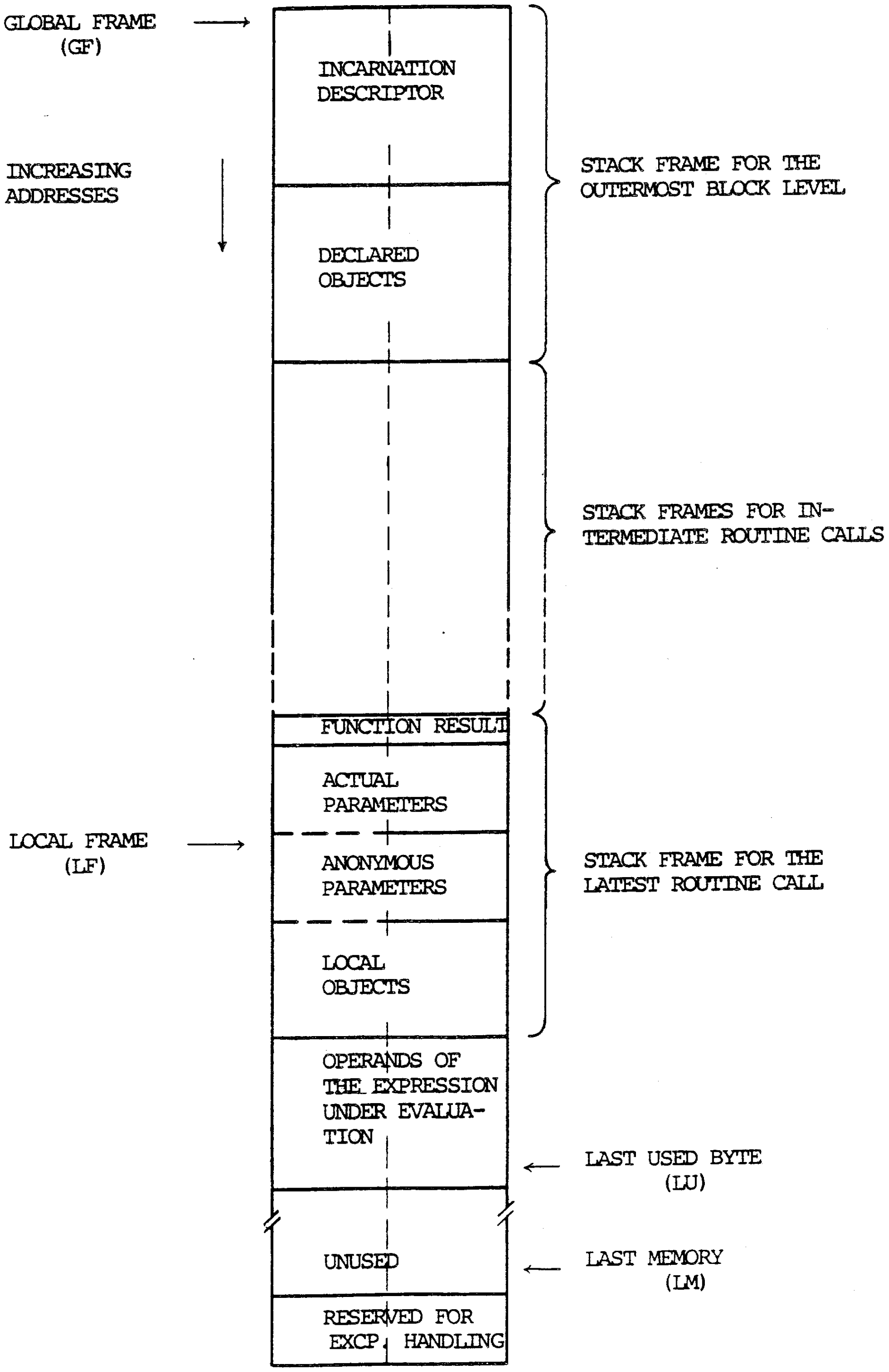


Figure 11: Process Incarnation Stack.

The outermost stackframe of a process incarnation is created when the incarnation is created and lives until the incarnation is removed. The first part of the outermost stackframe is used for the incarnation descriptor, which contains variables used by the microprogram and runtime-system.

Furthermore the outermost stackframe contains the values or the addresses of the actual process incarnation parameters.

Above the stackframe for the latest routine call, the evaluation stack is placed. Operands on the evaluation stack are always word aligned, and occupy always a full 16-bits word. To speed up evaluation stack operations up to eight words of the evaluation stack can be kept in the registerstack. The registerstack is described in section 3.4.

A process incarnation implies the existence of an incarnation descriptor, which is a data structure explained in Real-Time Pascal notation as follows:

```
incarnation_descriptor =
  RECORD
    timer          : integer;
    level          : byte;
    delaychain     : ↑incarnation_descriptor; (* adr *)
    instructioncode: byte;
    exic           : adr;
    exceptionpoint : addr;
    regset         : integer;
    mregset        : integer;
    .
    .
(* additional fields used by runtime system *)
END;
```

This definition implies the following constant declarations:

```
CONST
  timeroffset = 0;
  leveloffset = 2;
```

```
delaychainoffset = 3;
instructioncodeoffset = 6;
exicoffset = 7;
exceptionpointoffset = 10;
regsetoffset = 16;
mregsetoffset = 18;
```

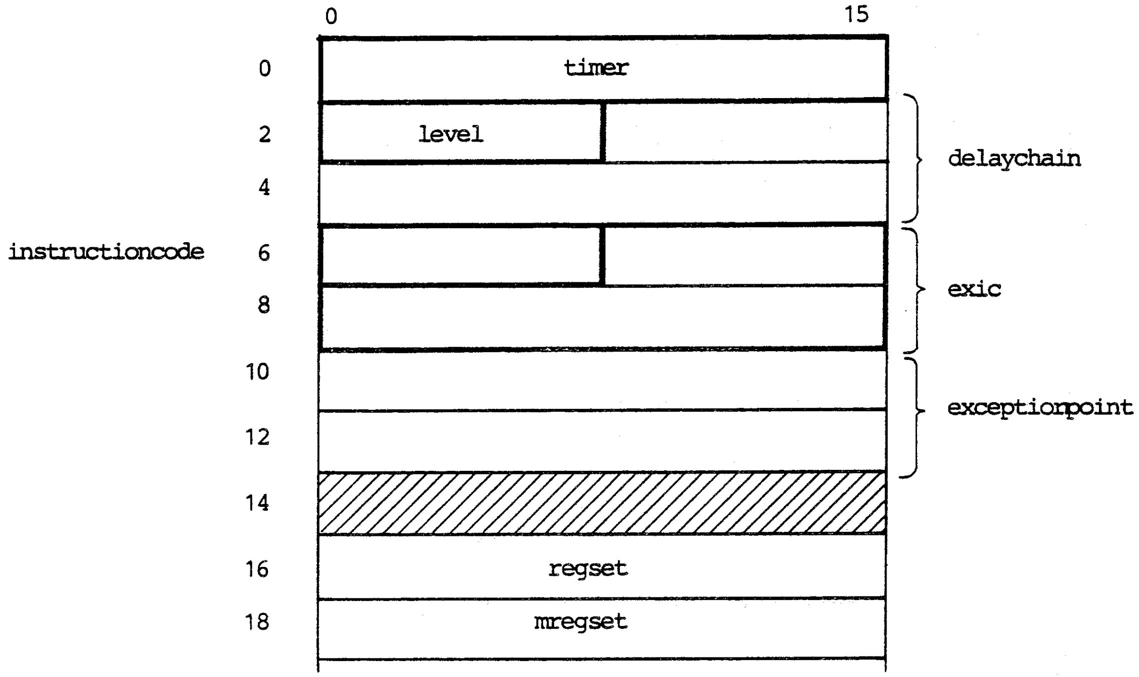


Figure 12: Incarnation Descriptor.

The following denotation is used to access the incarnation stack.

stack(disp: word) is equivalent to mem(a),
when a.base = sb.base and a.disp = disp.

stackbyte(disp: word) is equivalent to membyte(a),
when a.base = sb.base and a.disp = disp.

stackword(disp: word) is equivalent to memword(a),
when a.base = sb.base and
a.disp = disp.

3.3 The Register Set

A process incarnation stack is pointed out by a number of registers from a register set.

sb: Stack Base
The memory module containing the incarnation stack.

gf: Global Frame
The displacement of the incarnation descriptor, and in this way the stackframe of the outermost blocklevel.

lf: Local Frame
The displacement of the anonymous parameters belonging to the latest uncompleted routine call.

lu: Last Used byte
The displacement of the last used byte in the evaluation stack.

lm: Last Memory
The displacement of the last byte, which the process incarnation is allowed to use.

The remaining registers in the register set are used as follows:

ib: Instruction Base
The memory module containing the instruction to be executed. The nilbit in the Instruction Base is used to indicate that an interruptable instruction has been interrupted and that resumption of this process incarnation requires special treatment to finish the interrupted instruction.

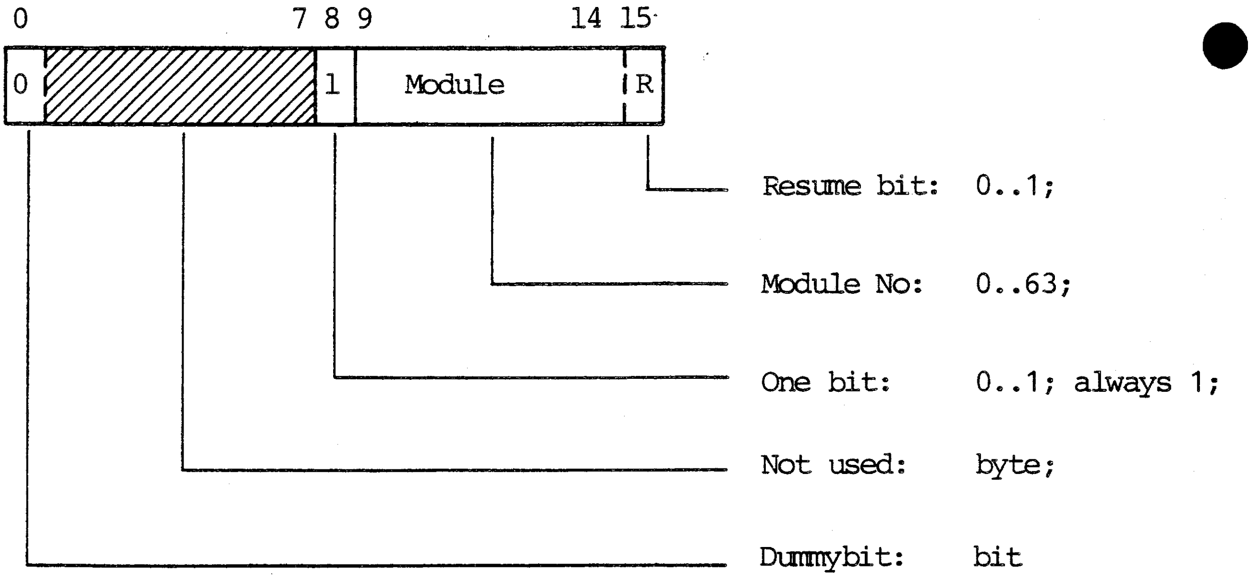


Figure 13: IB register layout.

ic: Instruction Layout

The displacement of the instruction to be executed by the process incarnation.

ps: Process incarnation State

This register has the following layout:

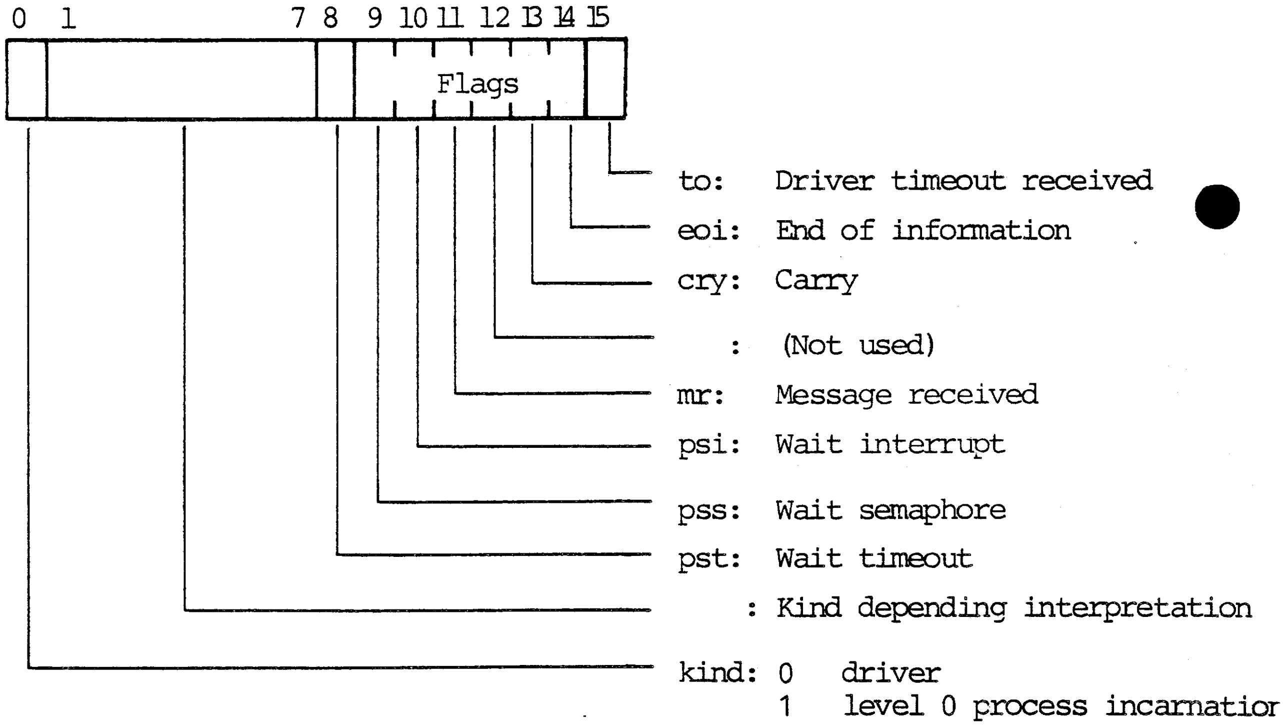


Figure 14: PS register layout.

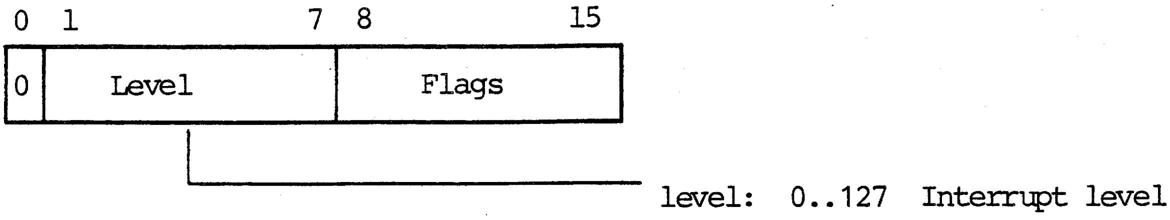


Figure 15: DRIVER PS register layout.

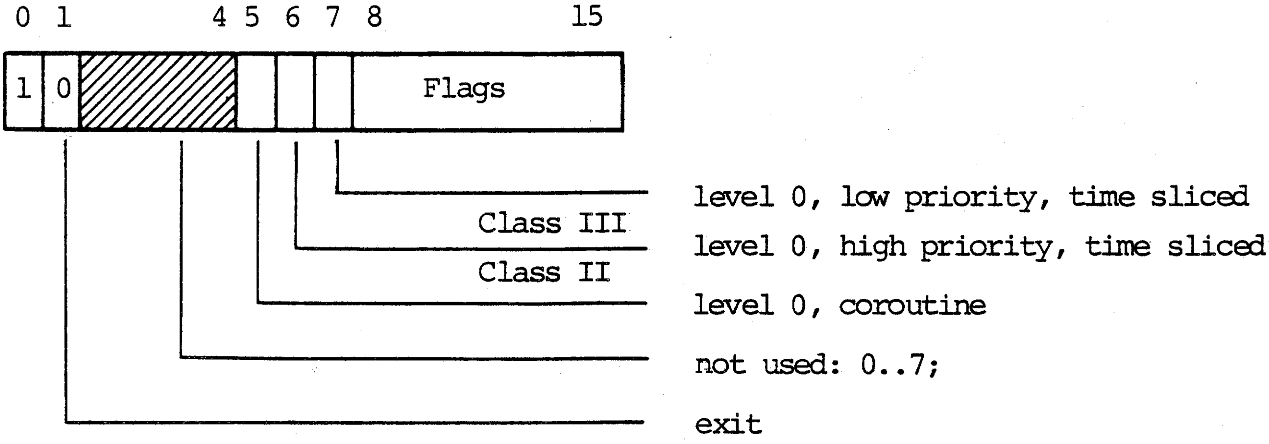


Figure 16: LEVELO PS register layout.

The pascal-types corresponding to a registerset is as follows:

```
ibtype = PACKED RECORD
           dummy: boolean;
           ?,?,?,?,?,?,?: bit;
           onebit: bit;
           module: 0..63;
           resumebit: boolean;
         END

baseword = RECORD
             nullbyte: byte;
             base    : basetype;
           END
```

```
leveltype = PACKED RECORD
              CASE level0: boolean OF
                true: RECORD
                        dummy: boolean;
                        ?,?,?: bit;
                        prio0,
                        priom1,
                        priom2: boolean;
                      END;
                false: (level: 0..127);
              END;
            END;

pstype = PACKED RECORD
           level: leveltype;
           pst: bit;
           pss: bit;
           psi: bit;
           mr:  bit;
           ?:   bit;
           eoi: bit;
           to:  bit;
         END;

regsettype = RECORD
               ps: pstype;
               sb: baseword;
               gf: word;
               lf: word;
               lu: word;
               lm: word;
               ib: ibtype;
               ic: word;
             END;
```

The total registerarray corresponds to the following declarations.

```
com8085type = RECORD
                parityregset: word;
                fifo01: word;
                fifo23: word;
                fifo45: word;
                cow: word;  (* value,disp) *)
                msgerrorcode: word;
                msgbase: word;
                msgdisp: word;
              END;

monitorregtype = RECORD
                   memregsetbase: baseword;
                   memregsetdisp: word;
                   waitqueuelast: word;
                   waitqueuefirst: word;
                   monitorlevel: word;
                   ?,?,?: word;
                 END;

breakpointsettype = RECORD
             breakpointmode: word;
             breakpointbase: word;
             breakpointdisp: word;
             ?,?,?,?,?: word;
           END;

RECORD
  registerset: ARRAY(0..121) OF regsettype;
  multreg:     ARRAY(0..7) OF word;
  masks:       ARRAY(0..15) OF word;
  errorset:    errorsettype;
  monitorreg:  monitorregtype;
  com8085:     com8085type;
END;
```

3.4 Register Stack

A register stack contains up to eight of the topmost words of the evaluation stack. All evaluation stack operations are performed upon the register stack. The compiler secures that an instruction can be executed within the register stack, i.e. all operands needed by the instruction are within the registerstack, and sufficient free registerstack words are present.

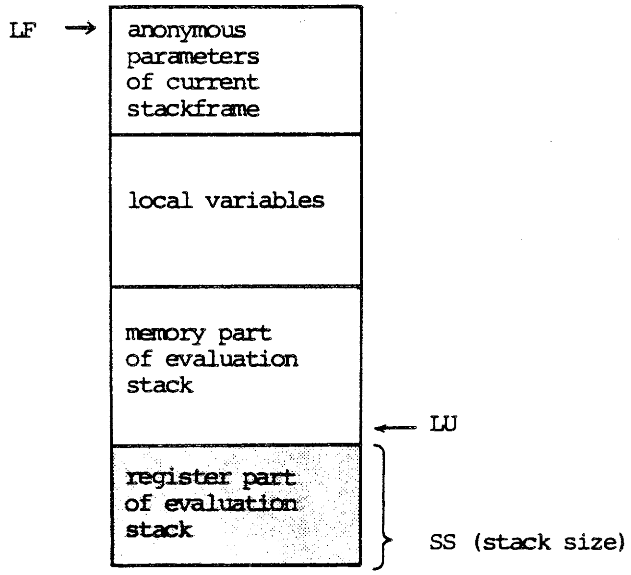

LF
anonymous parameters of current stackframe
local variables
memory part of evaluation stack
LU
register part of evaluation stack
SS (stack size)

Figure 17 : Evaluation stack.

If the number of free words is too small, a special instruction is inserted by the compiler. The parameter of this instruction defines the number of bytes (even number), which shall be present in the register stack. A suitable number of words is then either moved from the registerstack to the memory or from the memory to the registerstack. A register stack has a stack pointer ss, which tells the number of bytes contained in the registerstack. The register lu always points to the last used byte in the memory part of the evaluation stack. Some instructions implicitly empties the register stack, (i.e. a procedure call). Likewise all set expressions are evaluated in the memory part of the evaluation stack, because of the varied size of sets.

If the stackpointer of a register stack is moved outside its legal range, because of an erroneous sequence of instructions an exception will be detected at the next instruction fetch.

Each context contains a register stack.

The following denotation will be used when referring to the registerstacks.

```
TYPE
   regstack = RECORD
                 ss: (underflow,0,2,4,6,8,10,12,14,16, overflow);
                 contents: ARRAY (1..8) of word;
              END

VAR
   registerstacks: ARRAY (0..127) OF regstack;
```

pus:= x denotes the operation

```
   WITH registerstacks (context) DO
   BEGIN
     ss:= ss + 2;
     contents (ss div 2):= x;
   END
```

x:= pop denotes the operation

```
     WITH registerstacks (context) DO
     BEGIN
       x:= contents (ss div 2);
       ss:= ss - 2;
     END
```

x:= ss denotes

```
     registerstacks (context).ss
```

In the descriptions in chapter 5-12 the stack is shown before and after the execution of the instruction. The partition between memory stack is marked by a double line.

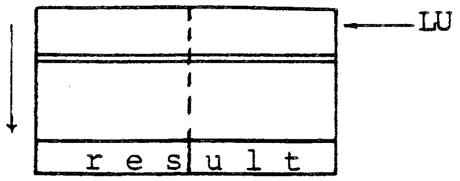


Figure 18: Stack.

The above figure indicates that the memory part is finished at the byte pointed out by LU. A number of words can then be placed in the register stack, but only the topmost element is used by the instruction (result).

3.5 The Prefetch Unit 3.5

The prefetch unit is shared by all process incarnations. The function of the unit is to enable simultaneous execution of an instruction and fetch of succeeding arguments and/or instructions. The prefetch unit has a buffer of 4 bytes, and it always reads a full word, acting as a DMA-device on the backplane bus. The unit is able to deliver both byte and words independent of word boundaries. The prefetch unit contains two registers holding the memory module and the displacement of the instruction counter of the executing process incarnation (the value of the registers ib and ic). The instruction counter always points to the next instruction to be executed. Each time a byte is retrieved the displacement part is increased by 1 and each time a word is retrieved increased by 2. The displacement part can be read to allow relative jumps, and the saving of the instruction counter, when a new process incarnation is scheduled for execution. The module part cannot be read, but the microprogram ensures that each time the instruction counter changes from one module to another, the contents of ib is changed too. Each time a new displacement part is loaded into the prefetch unit, the buffer is emptied and a new contents retrieved.

The word boundary independance of the prefetch unit is used in block moves, where the prefetch unit is loaded with the source address and the words retrieved through the prefetch unit thereby disabling swapping of bytes between even and odd addresses.

Notation:

The load and retrieval of the instruction counter from the prefetch unit is omitted in the description of the instructions, but can be viewed as connected to the ib and ic registers of the current context.

nextbyte

Denotes the retrieval of the next byte from the instruction stream.

nextword

Denotes the retrieval of the next word from the instruction stream.

3.6 Memory Registersets 3.6

To avoid the limitation of the fixed number of physical registersets, a waiting process incarnation can be dumped into a memory registerset placed in a memory module. The administration of memory registersets is managed by the monitor process, but the microprogram takes the placement of the registerset in consideration when operating on registersetchains.

A memory registerset contains two additional fields used to store the address of a received message.

The following constants define offsets in a memory registerset.

```
CONST
  psoffset = 0;
  sboffset = 2;
  gfoffset = 4;
  lfoffset = 6;
  luoffset = 8;
  lmoffset = 10;
  iboffset = 12;
  icoffset = 14;
  res0offset = 16;
  res1offset = 18;
```

All memory registerset must be allocated within 16K bytes of a memory module.

The register monitorreg.memregbase contains the memory module holding the memory registersets. The contents of the register monitorreg.memregdisp is added to the value of the first word of a semaphore (see section 3.10) to obtain the address of the memory registerset.

3.7 Registerset Waitqueue 3.7

A process incarnation, which registerset has been dumped into a memory registerset, is activated through chaining the registerset into a registerset waitqueue and giving an interrupt to the interruption level contained in the register monitorreg.monitorlevel.

The registerset waitqueue is a single linked list of registersets. The lf field is used as the link field and contains the effective address of the next memory registerset. The registers monitorreg.first and monitorreg.last contain the effective address of the first and last registerset in the registerset waitqueue.

3.8 Communication Data Structures 3.8

In the following sections the structures used in communication between process incarnations are described. The description covers the data structures realized by the hardware as well as those realized by the microprogram.

The communication supported by the RC3502 is known as message passing. Messages are communicated through queue semaphores. A detailed description of the supported communication principles may be found in ref. [3] and ref. [4].

3.9 Reference 3.9

A reference is represented by an addr. The nullbyte of the addr is used as a lockcount. If nullbyte <> 0 the reference is locked. Instructions are provided for increasing and decreasing of the nullbyte.

3.10 Semaphore 3.10

A queue semaphore is represented as a queue head for either a queue of messages or a queue of process incarnations.

If the semaphore queue is empty the semaphore is said to be passive and represented by a nil addr.

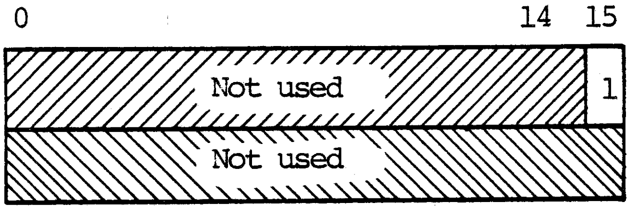


Figure 19: Passive semaphore.

If the semaphore queue is a queue of messages the semaphore is said to be open. The semaphore is represented by an addr referencing the message header of the last message in a single linked circular list of message headers.

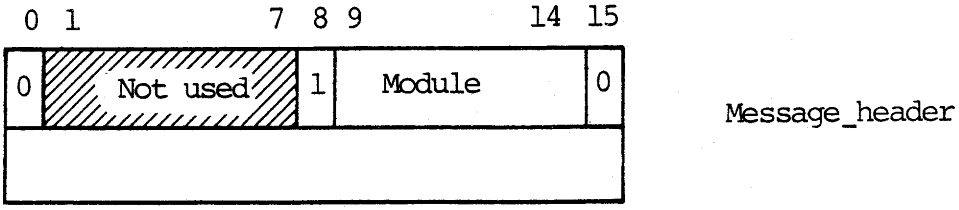


Figure 20: Open semaphore.

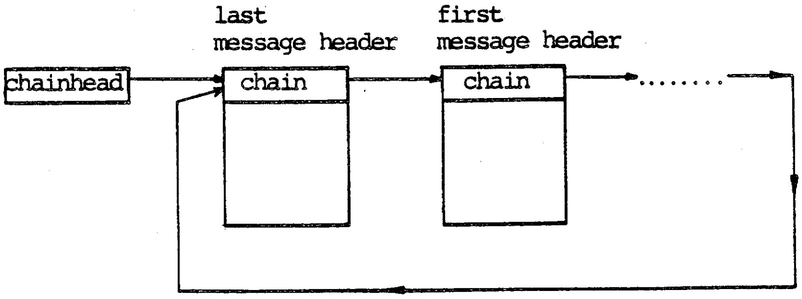


Figure 21: Message header chain.

If the semaphore queue is a queue of waiting process incarnations the semaphore is said to be locked. The semaphore is represented by a registerset number referencing a doubled linked list of registersets. The microprogram is able to handle memory registerset, when operating on the registersetchain.

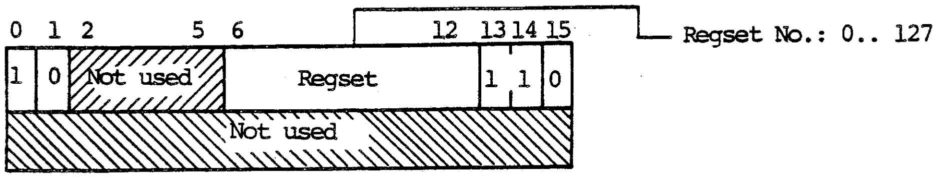


Figure 22: Locked semaphore, first incarnation has a physical registerset.

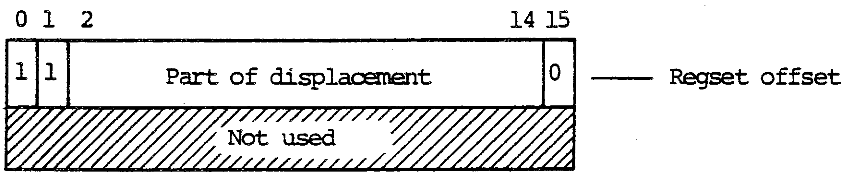


Figure 23: Locked semaphore, first incarnation has a registerset in memory. Bit 0 indicates that the semaphore is locked and bit 1 indicates that the first regset is a physical regset.

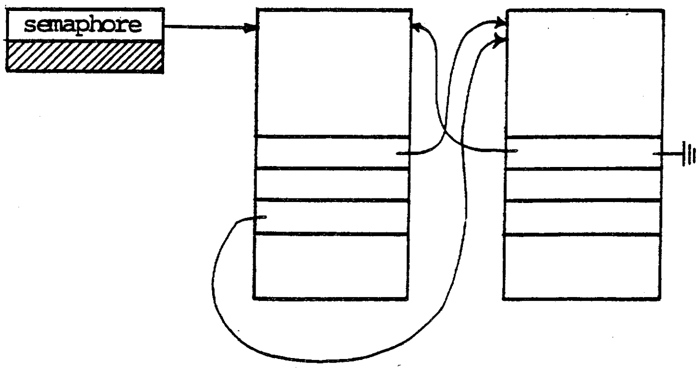


Figure 24: Registerset chain.

The registerset is chained through the registers usually containing LF and LM. The LF and LM are temporarily stored in the registerstack, when the process incarnation is waiting. The LF field is used as the successor reference, the LM field as the predecessor reference. Both fields have the same bitusage as the first word of a locked semaphore. The predecessor of the first registerset points to the last registerset, and the successor of the last registerset has bit 15 set to 1 indicating a nil pointer.

3.11 Messages 3.11

A message consists of a message header and a data part (possibly empty). A number of messages can be stacked in which case the message header of the topmost message references the datapart of the first message with a non-empty datapart (i.e. in fig. 26 a message stack, with three elements of which the topmost message has no datapart, is shown).

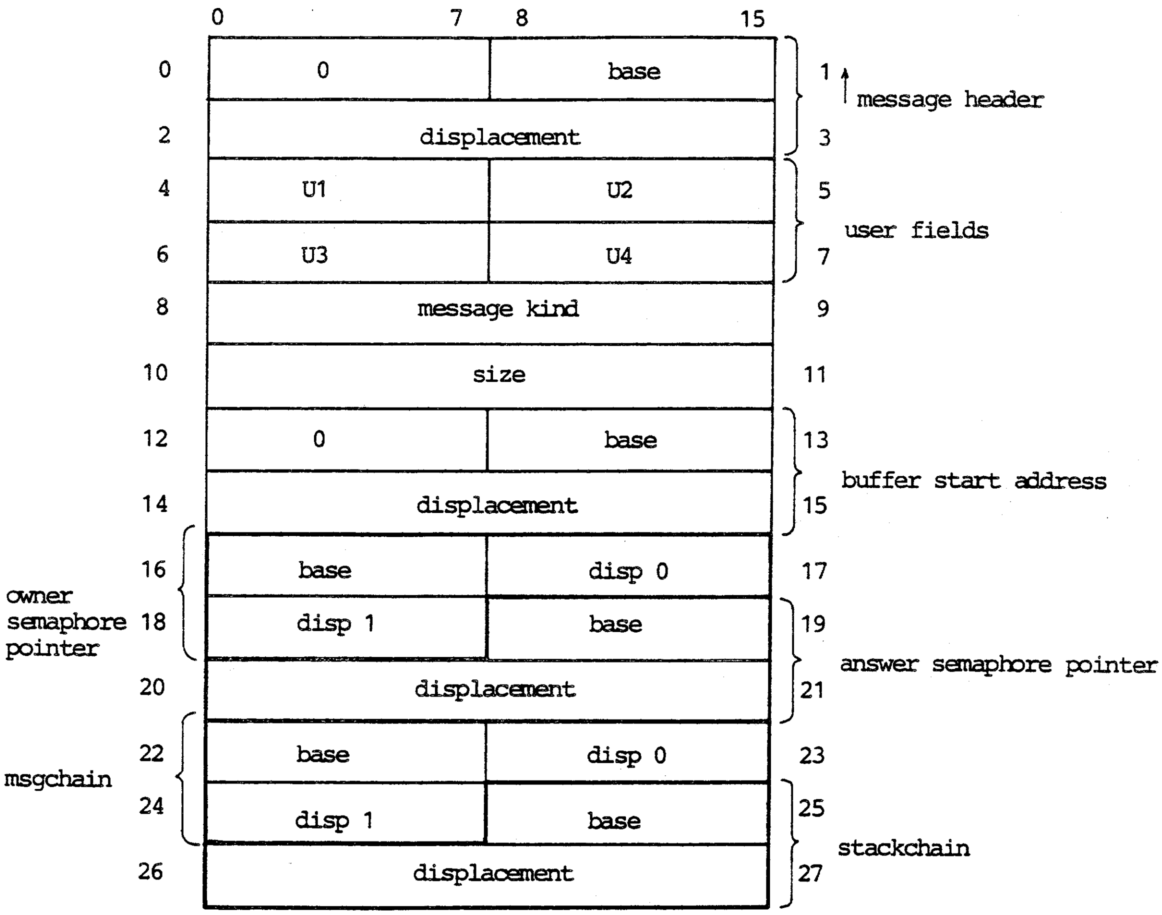


Figure 25: Message header layout.

```
messageheader =
RECORD
  nullbyte        : byte;
  chain           : ↑messageheader;
  u1, u2, u3, u4  : byte;
  messagekind     : integer;
  size            : integer;
  start           : addr;
  owner           : ↑semaphore;
  answer          : ↑semaphore;
  msgchain        : ↑messageheader;
  stackchain      : ↑messageheader;
END;
```

Implying the following constant declarations:

```
CONST
  kindoffset = 8;
  sizeoffset = 10;
  startoffset = 13;
  stackoffset = 25;
```

The fields u1..u4 owner, answer, msgchain are used by the runtime system and for communication purposes.

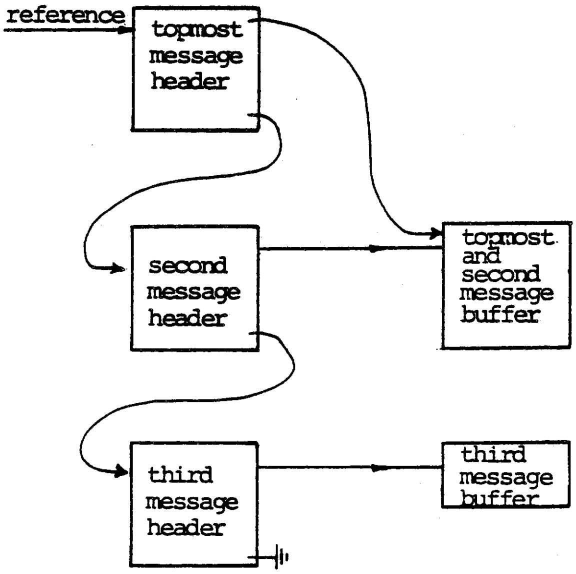


Figure 26: Stacked messages.

3.12 The Executing Process Incarnation 3.12

The processing unit is able to execute the instructions of a single process incarnation only. The process incarnation is within the processing unit represented by a pointer to the context set of this incarnation. This pointer is declared as:

```
VAR
  context: 0..121;
```

(Actually this pointer is a pointer to the last register in the registerarray).

All instruction descriptions in the following chapters must therefore be viewed as prefixed by the statement:

```
   WITH registerset (context) DO
```

Like the registerset pointer, there exists a register containing the interruption level upon which the incarnation is executed.

```
   VAR
     curlevel: 0..127;
```

3.13 Scheduling Structures 3.13

Two different structures are used to support the schedule function of the processing unit, one for incarnations execution at interruption level > 0 and one for those executing at interruption level = 0.

Each interruption level has an interrupt bit. When this bit is set the context associated with the interrupt bit requests execution. The usual pattern for a process incarnation at an interruption level > 0 is that only a few instructions are executed at interruption level > 0, whereafter level 0 is selected for further execution, giving service to process incarnations at a lower level. To avoid moving the contents of the registersets, the interruption level is used as an index in a contextset table to obtain the contextset for the processincarnation belonging to the interruption level. The registerset table maps all of the 128 interruption levels upon one of the 122 contextsets, and is called an intmap. A single registerset (registerset0) is used for all interruption levels not connected to a registerset.

Declaration:

```
interruptbits: ARRAY (0..127) OF bit;
intmap: ARRAY (0..127) OF 0..121;
```

The algorithm performed when selecting a process incarnation at an interruption level > 0 can be described as follows:

```
level:= 127;
WHILE level > curlevel DO
  IF interruptbits (level) = 1
  THEN BEGIN
    curlevel:= level;
    context:= intmap (level);
  END
  ELSE level:= level - 1;
```

The interruption bits are actually distributed upon the various I/O controllers. The level of the highest priority interrupt bit is obtained through the backplane bus. At system initialization a configuration table of the interruption bits present, is build in the debugger working area.

In contrast to levels > 0 several process incarnations can simultaneously be running at level 0. Process incarnations at level 0 can be subdivided into priority 0, -1 and -2. Process incarnations with priority 0 is usually executed as coroutines, meaning that no other priority 0 process incarnation will be executing before the executing process incarnation deschedules by executing some wait instruction.

Scheduling at level 0 is performed by associating three active flags to each contextset, one for each priority. A process incarnation is then activated by setting the flag for the contextset according to the priority of the incarnation (i.e. if the priority -1 incarnation belonging to context set 86 has to be activated, the priority -1 flag of context set 86 is set).

Three independent hardware scanners, one for each priority, select the next contextset to be executed. Each scanner acts in a round robin fashion. As long as the scanner has not met a flag it continues scanning the flags.

When a flag is met it stops further scanning until the contextsetindex found by the scanner has been used to start execution of the associated process incarnation. The coroutine facility of priority 0 is obtained by postponing the restart of the priority 0 scanner until the selected process incarnation performs some wait instruction, thereby removing the active flag.

A new level 0 process incarnation is selected after a given timeslice, indicated by a schedule interrupt to the processing unit. The scheduling interval is approximately 500 µs.

The scheduling structure for level 0 processincarnations can be declared as follows.

```
   activeflags: array (0..127) of 0..7;
                  (* only value 0, 1, 2, 4 is used *)
```

The algorithm performed by each of the three scanners is described by the following declaration:

```
PROCESS scanner (p: prio;
                 VAR activeflags: activeflag;
                 VAR regset: 0..121;
                 VAR stopped: boolean;
                 VAR restart: boolean);
VAR
prioval: integer

BEGIN
  IF prio = 0 THEN prioval:= 4;
  IF prio = -1 THEN prioval:= 2;
  IF prio = -2 THEN prioval:= 1;

  REPEAT
    IF activeflags (regset) = prioval
    THEN BEGIN
      stopped:= true;
      WHILE NOT restart AND
            activeflags (prioval) DO;
      stopped:= false;
      regset:= regset + 1;
      IF regset = 128 THEN regset:= 0;
    END
  UNTIL FALSE;
END;
```

The three incarnations of the scanner process are created with the following parameters.

```
prio0scanner:
   scanner (0, activeflags, prio0regset, prio0stop,
            prio0restart);

prioml scanner:
   scanner (-1, activeflags, prioml regset, prioml stop,
            prioml restart);

priom2scanner:
   scanner (-2, activeflags, priom2regset, priom2regset,
            priom2restart);
```

The total schedule algorithm can now be described by the routine nextset.

The algorithm described by nextset is performed after the occurrence of one of the following three events.

1) An interrupt at higher level.
2) A schedule interrupt indicating a finished timeslice.
3) Execution of certain instructions (i.e. a wait or clear interrupt instruction).

```
PROCEDURE nextset;
BEGIN
  level:= 127;
  WHILE level > curlevel DO
  BEGIN
    IF interruptbits (level) = 1
    THEN BEGIN
      curlevel:= level;
      context:= intmap (level);
    END
    ELSE level:= level - 1;
  END;
  IF level = 0
  THEN BEGIN
    IF prio0stop
    THEN BEGIN
      context:= prio0regset;
    END
    ELSE IF priom1stop
         THEN BEGIN
           context:= priom1regset;
           priom1restart:= true;
         END
         ELSE IF priom2stop
              THEN BEGIN
              context:= priom2regset;
              priom2restart:= true;
         END
         ELSE context:= intmap (0);
  END;
END;
```

3.14 Debug Interface 3.14

The following is a description of the functions performed by the control microprocessor, as seen from a programmer's point of view.

3.14.1 The Variable Array 3.14.1

This array is used as a communication area between the control microprocessor and the microprogram.

The variable array is a RAM memory area in the microprocessor, which can be accessed by the debug console operator using the command Y <yaddr> described in chapter 16. A copy of this RAM area is kept in the RC3502 memory; whenever the microprocessor writes in the area or the microprogram writes in its copy, an update communication takes place between the microprocessor and the microprogram. Therefore special RC3502 instructions are provided for access of the variable array.

RAM addr	
0	RTC level
1	TTO level
2	TTI level
3	timer low]
4	timer high
5	watchdog low]
6	watchdog high
7	TTI input
8	TTO output
9	version number
A	switches 0-7
B	switches 8-F
C	interruption level configuration
1B	
1C	RAM module configuration
1D	
1E	
1F	
20	
21	
22	ROM module configuration
23	
24	
25	
26	
27	
	used by runtime system

Figure 27: Control microprocessor RAM layout.

3.14.2 Real-Time Clock 3.14.2

The value of this timer is placed in RAM addr (4, 3). The step value of the timer is 2.5 milliseconds. When the timer counts down to 0, an interrupt is sent to the level placed in RAM addr 0. The default value of addr 0 is level 1. The default value of addr (4, 3) is (0, 8) corresponding to a timer value of 20 milliseconds.

3.14.3 Console (TTY) Communication 3.14.3

Communication with the Teletype (TTY) compatible debug console occurs on two interruption levels, one for input (RAM addr 2) and one for output (RAM addr 1). RAM addr 7 and 8 are used as data buffers. After power up, RAM addr 1 and 2 are zero.

3.14.4 Watchdog 3.14.4

This timer counts down from the value placed in RAM addr (6, 5), and if the most significant byte (byte 6) decrease to 0, the microprogram is commanded to execute an autoload. The step value is 2.5 milliseconds. The default value of RAM addr (6, 5) is (0, 0) corresponding to a disable of the watchdog function.

3.14.5 Configuration 3.14.5

After power up, the control microprocessor tests the current RC3502 hardware configuration for RAM modules and EPROM modules, and generates a configuration table in its own RAM area. Later on the RC3502 boot program updates the configuration table for interruption levels.

RAM Configuration

The RAM module configuration bit map is placed in microprocessor RAM addr (1C, 21) with the following layout:

Bit:	0	1	2	3	4	5	6	7
1C:	A0	A2	A4	A6	A8	AA	AC	AE
1D:	B0							BE
1E:	C0							CE
1F:	D0							DE
20:	E0							EE
21:	F0							FE

If a module exists, a 1 is placed in the corresponding bit position; otherwise a 0 is written.

EPROM Configuration

The EPROM module configuration bit map is placed in microprocessor RAM addr (22, 27) with a layout like that of the RAM configuration. The existence of an EPROM module can be detected only if the first address (word) in the memory module contains the value AAAA$_{Hex}$.

Interruption Level Configuration

The boot program updates this bitmap. A bit is set for each present interruption level 0..127 placed in microprocessor RAM addr (0C-1B).

3.15 Common Declarations 3.15

3.15.1 Exception 3.15.1

```
CONST
  parityexception        = 1;
  registerstackexception = 2;
  illegalinstruction     = 3;
  oddoperand             = 4;
  stackoverflow          = 5;
  nilpointer             = 6;
  nilreference            = 7;
  refnotnil               = 8;
  reflocked               = 9;
  lockoverflow            = 10;
  overflow                = 11;
  indexexception          = 12;
  fieldexception          = 14;
  wrapexception           = 15;
  rleqr2                  = 16;
  reflstacked             = 17;
  sizeexception           = 18;
  locktype                = 19;
  nodatamessage           = 20;
  nochannel               = 21;
  level0io                = 23;
  setcrexception          = 24;
  truncationexception     = 25;

PROCEDURE exception(cause: word);
BEGIN
  stack(gf + exceptioncodeoffset):= cause;
  stackbyte(gf + instcodeoffset):= inst;
  stackbyte(gf + exicoffset):= byte(ib.base);
  stackword(gf + exicoffset + 1):= ic;
  ib:= ibtype(stackbyte(gf + exceptionpointoffset));
  ic:= stackword(gf + exceptionpointoffset + 1);
  GOTO fetch;
END;
```

3.15.2 Stack Check and Dump 3.15.2

The following routine is used to move the registerstack to memory, to test for stackoverflow and adjustment of LU.

```
PROCEDURE checkanddumpstack(reserve: integer);
BEGIN
  IF lu + stacksize > lm - reserv
   THEN exception(stackoverflow)
  FOR i:=stacksize DIV 2 DOWNTO 1 DO
    MEM(lu+2*i-1):= pop;
  lu:= lu + stacksize + reserv;
END;
```

3.15.3 Offset Generation 3.15.3

These routines generate an adr as an offset from an adr or addr.

```
FUNCTION adroffset(a: adr; offset: word): adr;
BEGIN
  adroffset.base:= a.base;
  adroffset.disp:= a.disp + offset;
END;

FUNCTION addroffset(a: addr; offset: word): adr;
BEGIN
  addroffset.base:= a.base;
  addroffset.disp:= a.disp + offset;
END;
```

3.16 Type Conversion 3.16

Two forms of type conversion, which is not defined in Real-Time Pascal, are used.

3.16.1 The AS Construct 3.16.1

The construct:

```
  WITH a AS t1 DO
```

gives a the type t1 allthough it already has been defined as being of type t2. t1 must occupy the same number of bytes as a type compatible with t2.

3.16.2 Type Conversion Routines 3.16.2

A type identifier can be used as a type conversion routine. I.e. basetype(b) is used to convert the byte b into a basetype value.

4. INSTRUCTION FETCH

This chapter contains a description of the instruction fetch performed by the microprogram.

The following is a short description to the flowchart in fig. 28. The labels refer to the leftmost question in fig. 28.

init:

initialize the micromachine to a proper state. A description of the initial state can be found in chapter 14.

error:

takes care of registerstack or parityerrors found in the previous instruction.

debugrequest:

executes request issued by the 8085 control microprocessor. A special command forces the microprogram to reinitialize itself.

stopmode:

In stopmode each single instruction execution is controlled from the control microprocessor. Usually the microprogram idles with the breakpointmode register equal to zero (for further description is referred to ref. [5]).

interrupt:

Interrupts can be either interrupts caused by the activation of a process on a higher level or interrupts caused by a schedule counter indicating the excess of a timeslice.

resume:

If the resumebit in a registerset is present, the corresponding process incarnation has been interrupted during the execution of a single instruction, i.e. a move-, wait- or set-instruction. An intermediate state is saved in the registerset and the resumebit indicates that the instruction shall be resumed in a special manner.

The resumebit is set, when the instruction execution exits through resumeinstruction. Some interruptable instructions do not need the resumebit to be set since the saved intermediate state allows the instruction to be repeated. This is the case for block-I/O instructions.

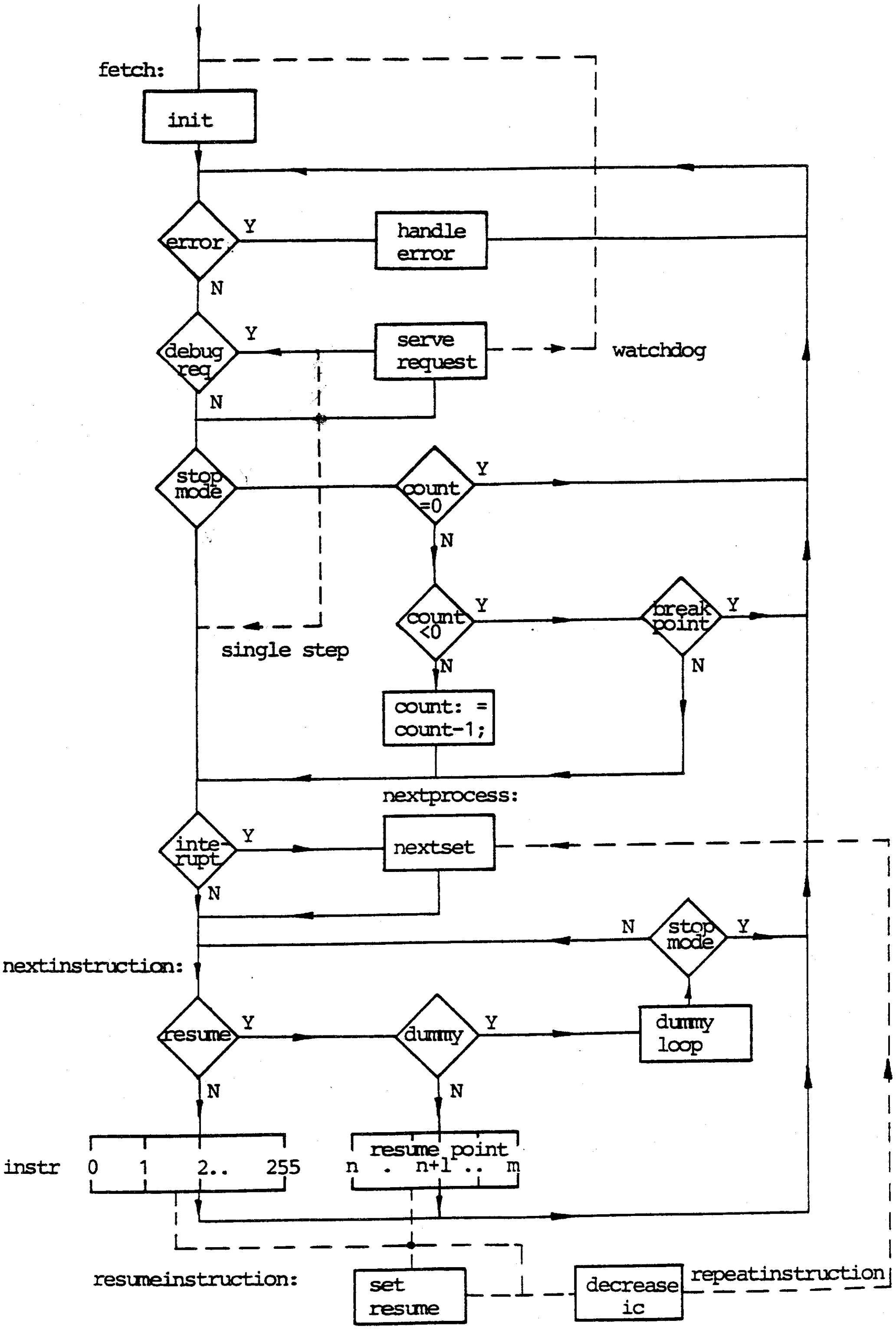


Figure 28: Instruction fetch.

The flowchart reflects the following algorithm. Neither the flowchart nor the algorithm is reflected by a corresponding microprogram sequence, since most of the test is performed in parallel.

```
init:
   initialize;
fetch:
   IF error THEN handleerror;
   IF debugrequest THEN serverequest;
   IF stopmode
   THEN WITH breakpointset DO BEGIN
     IF breakpointmode = 0 THEN GOTO fetch;
     IF breakpointmode > 0
        THEN breakpointmode:= breakpointmode - 1
        ELSE IF (breakpointbase = ib) AND (breakpointdisp = ic)
             THEN GOTO fetch;
   END;
   IF interrupt THEN nextset;

nextinstruction:
   IF ib.resume
   THEN WITH registerset (context) DO
   CASE nxb OF
     (* execute instruction *)
   END
   ELSE BEGIN
     IF not ib.dummy
     THEN BEGIN
       ib.resume:= false;
       WITH registerset (context) DO
       CASE nxb OF
         (* execute resumepart of instruction *)
       END
     END
     ELSE
     REPEAT (* dummy loop *)
       IF curlevel <> 0 THEN device (curlevel).interrupt:= 0;
       count:= pop + 1; pus:= count;
       IF carry THEN ic:= ic + 1; (* dummy counter *)
       IF debugrequest THEN serverequest;
       nextset;
       IF stopmode THEN GOTO fetch;
     UNTIL ib.dummy = false;
     GOTO nextinstruction;
   END;
   GOTO fetch;

resumeinstruction:
   ib.resume:= true;

repeatinstruction:
   ic:= ic - 1;

nextprocess:
   nextset;
   IF stopmode OR debugrequest
   THEN GOTO fetch
   ELSE GOTO nextinstruction;

PROCEDURE handleerror;
(* a parity or stacklimit error has occurred during the
   previous instruction *)
VAR
paritycode: word;
BEGIN
  IF stacklimit
  THEN BEGIN
    pop; (* remove stacklimit *)
    IF stackunderflow THEN pus:= 0;
    exception (registerstackexception);
  END
  ELSE BEGIN (* parity error *)
    paritycode:= 41; (* illegal module *)
    IF leftparity THEN paritycode:= paritycode + 2;
    IF rightparity THEN paritycode:= paritycode + 1;
    com8085.msgerrorcode:= paritycode;
    com8085.msgbase:= pbas; (* baseword of parityaddress *)
    com8085.msgdisp:= padr; (* displacement of parityaddress
                                clears parity condition too *)
    com8085.parityregset:= context;
    set8085interrupt;
    REPEAT
      WHILE NOT debugrequest DO;
      serverequest;
    UNTIL stopmode;
    exception(parityexception);
  END;
END;
```

5. RETRIEVAL OF A VALUE 5.

5.1 Push Nonsense (Reserve Stack Space) 5.1

The operand is retrieved, the register stack moved to memory and a result which occupies <operand> bytes is pushed on the memory stack. The contents of the result are undefined.

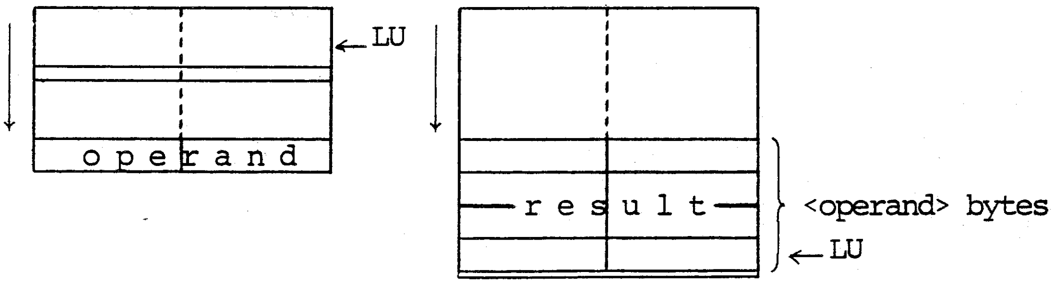
5.1.1 RENPB 5.1.1

Value: $8E_{Hex}$

RÉtrieve Nonsense via P(ush Down List) Bytes

IC → RENPB

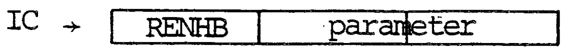
STACK BEFORE: STACK AFTER:

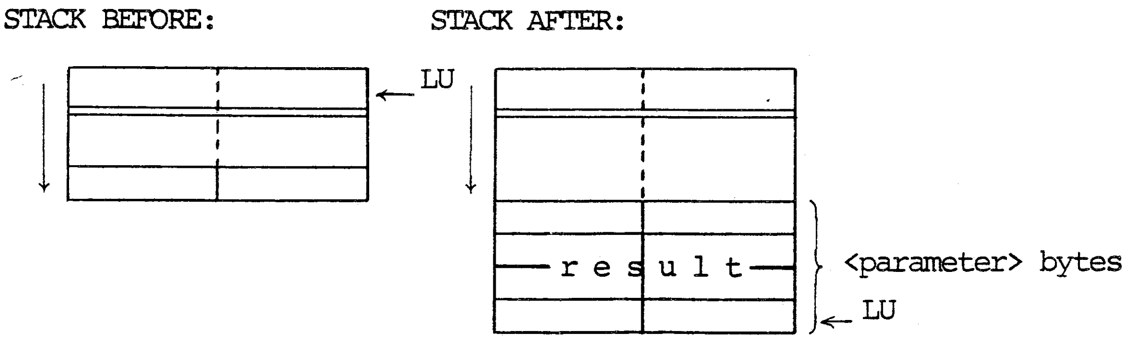


```
VAR
  operand : word;
BEGIN
  operand:= pop;
  IF odd(operand) THEN exception(oddoperand);
  checkanddumpstack(operand);
END;
```

5.1.2 RENHB

REtrieve Nonsense H(ere) Byte   Value: $8F_{Hex}$





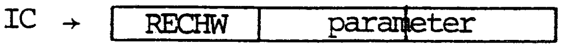
```
(* RENHB *)
VAR
  operand : word;
BEGIN
  operand:= nextword;
  IF odd(operand) THEN exception(oddoperand);
  checkanddumpstack(operand);
END;
```

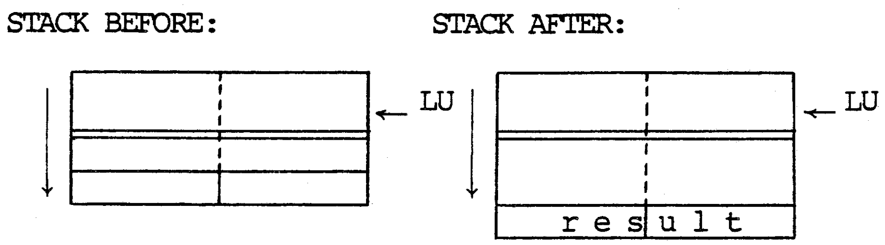
5.2 Push Constant 5.2

The operand is retrieved and pushed on the register stack as the result.

5.2.1 RECHW 5.2.1

REtrieve Constant H(ere) Word Value: A8$_{Hex}$

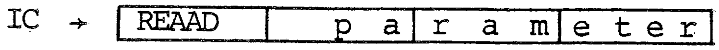




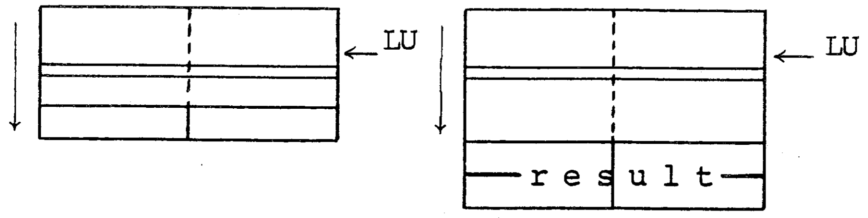
```
BEGIN
  pus:= nextword;
END;
```

5.2.2 REAAD

REtrieve Address A(bsolute) Double   Value: FB_{Hex}



STACK BEFORE: STACK AFTER:



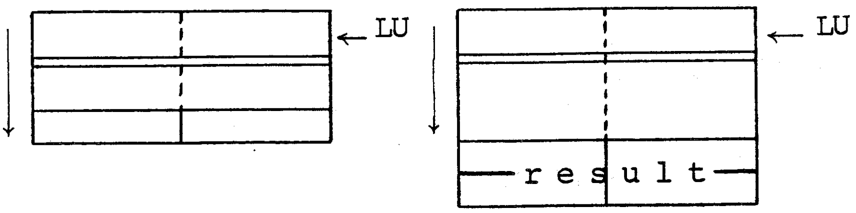
```
BEGIN
  pus:= word(nextbyte);
  pus:= nextword;
END;
```

5.2.3 RECHD

REtrieve Constant H(ere) Double   Value: $E8_{Hex}$

IC → | RECHD | p a | r a m | e t e | r |

STACK BEFORE: STACK AFTER:



```
BEGIN
  pus:=nextword;
  pus:=nextword;
END;
```

5.3 Push Address 5.3

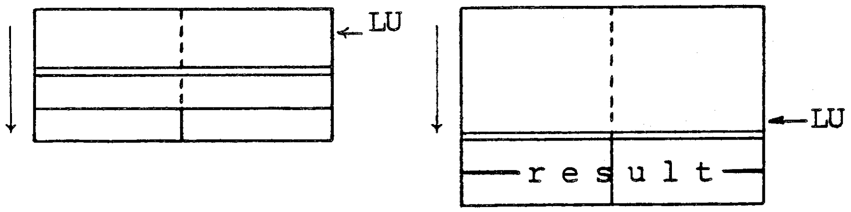
The operand is an address, which is pushed on the register stack as the result.

5.3.1 REAXD 5.3.1

REtrieve Address X (path) Double Value: FD_{Hex}

IC → REAXD

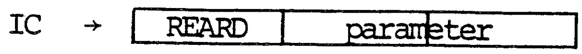
STACK BEFORE: STACK AFTER:

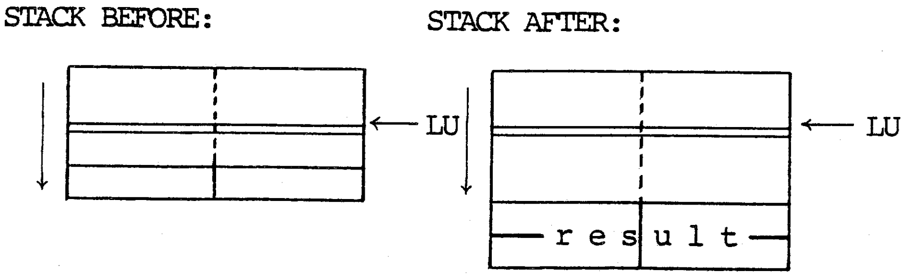


```
BEGIN
  checkanddumpstack(∅);
  pus:= word(sb);
  pus:= lu;
END;
```

5.3.2 REARD

REtrieve Address R(elative) Double   Value: FC_{Hex}



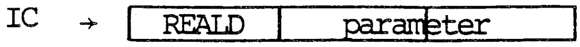


The result is the address denoted by ic + parameter.

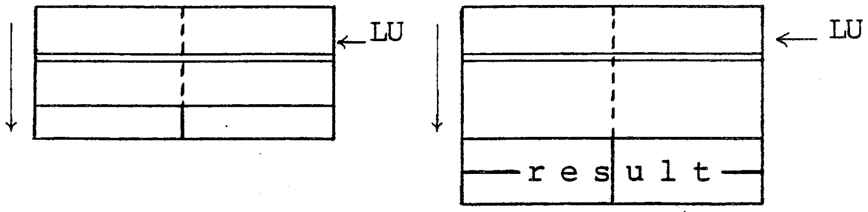
```
VAR
  oldic : word;
BEGIN
  pus:=word(ib);
  oldic:= ic - 1;
  pus:=oldic + nextword;
END;
```

5.3.3 REALD

REtrieve Address L(ocal) Double   Value: $E4_{Hex}$



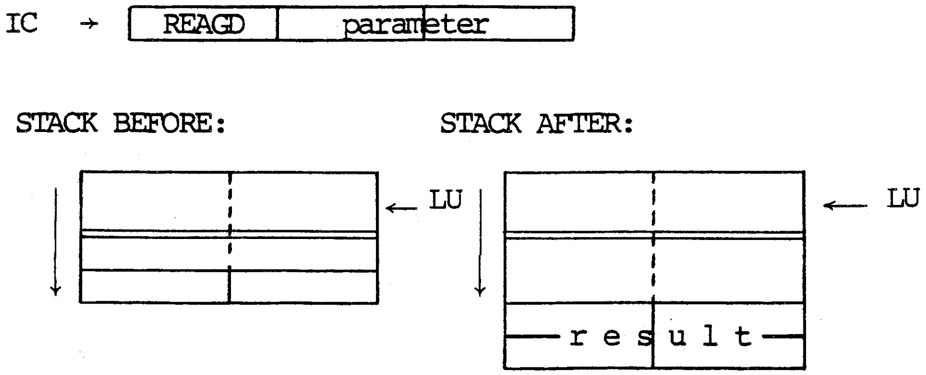
STACK BEFORE: STACK AFTER:



```
BEGIN
  pus:= word(sb);
  pus:= lf + nextword;
END;
```

5.3.4 REAGD

REtrieve Address G(lobal) Double   Value: $B4_{Hex}$



```
BEGIN
  pus:= word(sb);
  pus:= gf + nextword;
END;
```

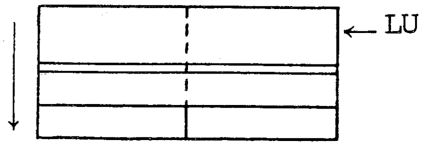
5.3.5 REAID

Value: $E2_{Hex}$

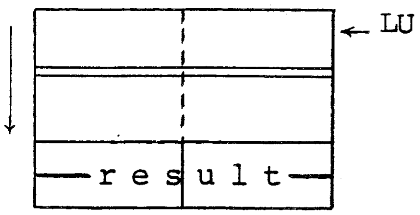
REtrieve Address I(ntermediate) Double

IC → | REAID | param 1 |

STACK BEFORE:



STACK AFTER:



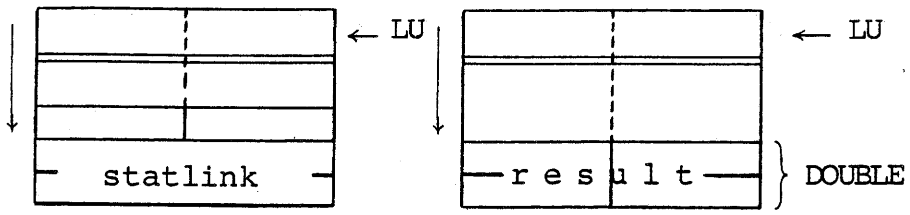
```
VAR
  statlink : adr;
  level : byte;
  i : integer
BEGIN
  pus:= word(sb);
  statlink.base:= sb.base;
  statlink.disp:= lf;
  level:= nextbyte;
  FOR i:=1 TO level DO statlink.disp:= mem(statlink);
  pus:=statlink.disp;
END;
```

5.3.6 REAISD 5.3.6

REtrieve Address I(ntermediate) Stack Double Value: $B5_{Hex}$

IC → | REAISD | param 1 |

STACK BEFORE: STACK AFTER:



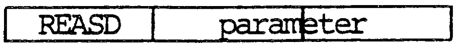
```
VAR
  statlink : adr;
  level : byte;
  i : integer
BEGIN
  statlink.base:= sb.base;
  statlink.disp:= pop;
  level:= nextbyte;
  FOR i:=1 TO level DO statlink.disp:=mem(statlink);
  pus:= statlink.disp;
END;
```

5.3.7 REASD or UADHW 5.3.7

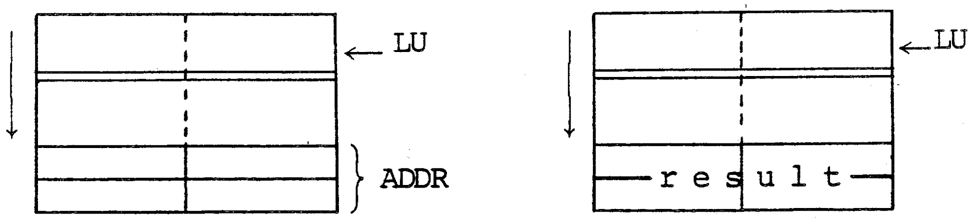
REtrieve Address via S(tack) Double Value: E6$_{\text{Hex}}$

Unsigned ADd H(ere) Word

IC →



STACK BEFORE: STACK AFTER:



```
VAR
  operand : word;
BEGIN
  operand:= pop;
  pus:= operand + nextword;
END;
```

5.4 Push Operand 5.4

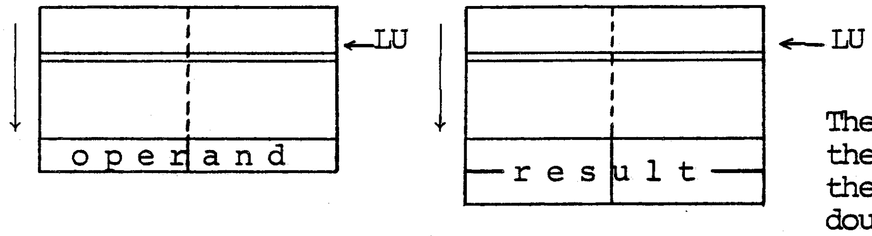
The value of the operand is retrieved and pushed on the register stack as the result.

5.4.1 REVPW 5.4.1

REtrieve Value P(ush Down List) Word   Value: AF$_{\text{Hex}}$

IC → REVPW

STACK BEFORE: STACK AFTER:



The WORD on the top of the stack is doubled.

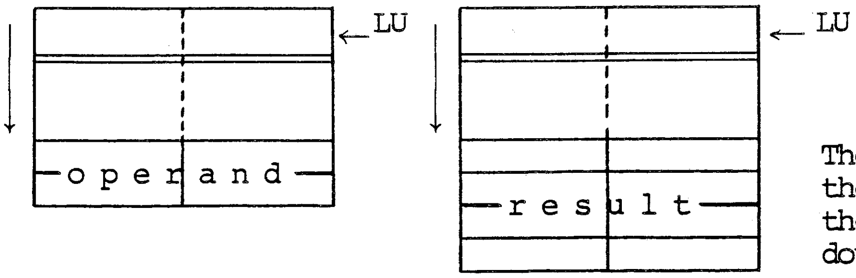
```
VAR
  operand : word;
BEGIN
  operand:= pop;
  pus:= operand;
  pus:= operand;
END;
```

5.4.2 REVPD

REtrieve Value P(ush Down List) Double Value: EF_{Hex}

IC → REVPD

STACK BEFORE: STACK AFTER:

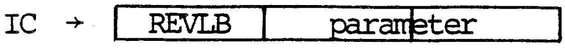


The DOUBLE on the top of the stack is doubled.

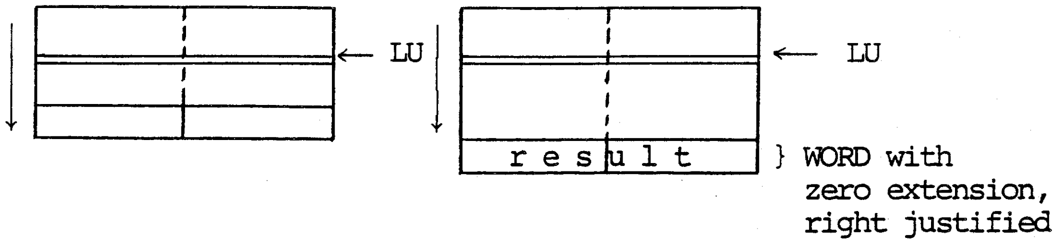
```
VAR
  operand : double;
BEGIN
  operand.w2:= pop;
  operand.wl := pop;
  pus:= operand.wl ;
  pus:= operand.w2;
  pus:= operand.wl ;
  pus:= operand.w2;
END;
```

5.4.3 REVLB

REtrieve Value L(ocal) Byte Value: 97_{Hex}



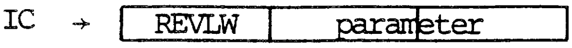
STACK BEFORE: STACK AFTER:

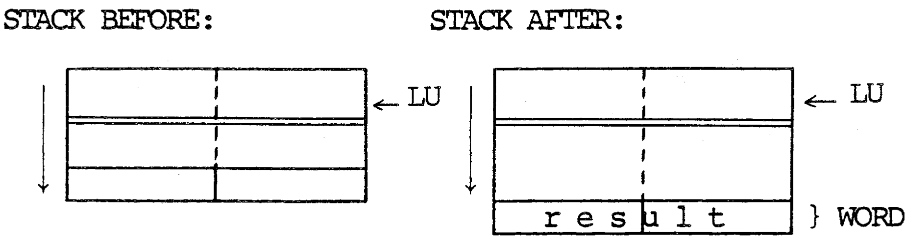


```
BEGIN
  pus:= word(stackbyte(lf + nextword));
END;
```

5.4.4 REVLW

REtrieve Value L(ocal) Word   Value: B7$_{\text{Hex}}$

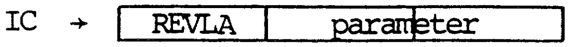




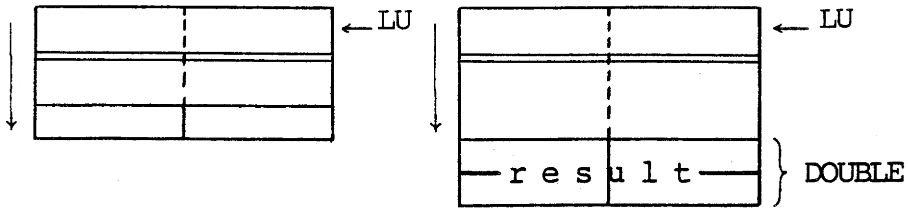
```
BEGIN
  pus:= stackword(lf + nextword);
END;
```

5.4.5 REVLA

REtrieve Value L(ocal) Address Value: $D7_{Hex}$



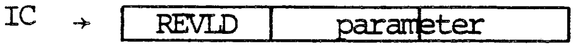
STACK BEFORE: STACK AFTER:

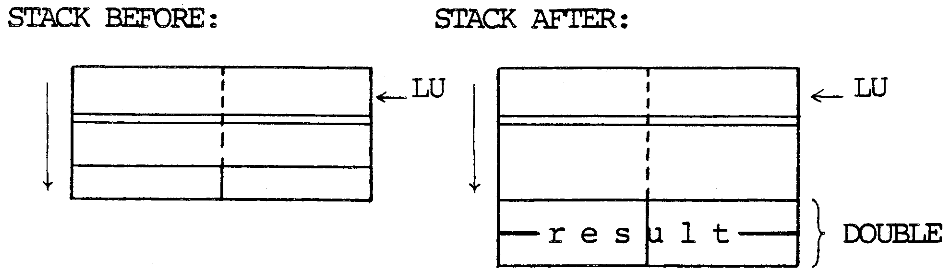


```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  pus:= word(stackbyte(lf + offset));
  pus:= stackword(lf + offset + 1);
END;
```

5.4.6 REVLD

REtrieve Value L(ocal) Double   Value: $F7_{Hex}$

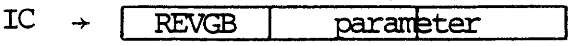


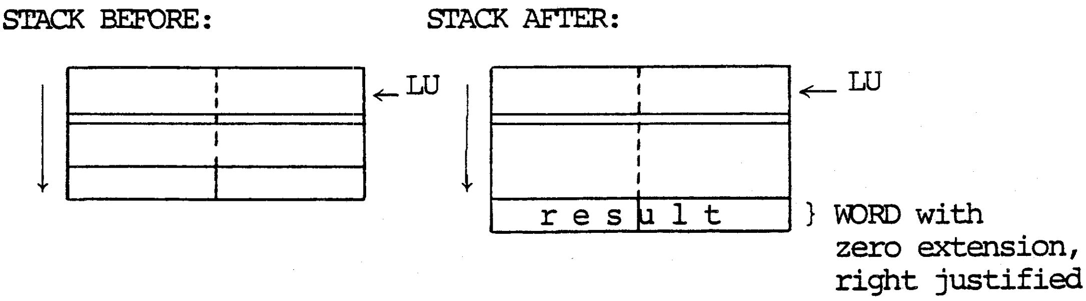


```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  pus:= stackword(lf + offset);
  pus:= stackword(lf + offset + 2);
END;
```

5.4.7 REVGB

REtrieve Value G(lobal) Byte   Value: 93_{Hex}

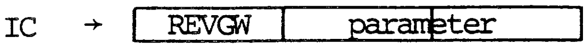




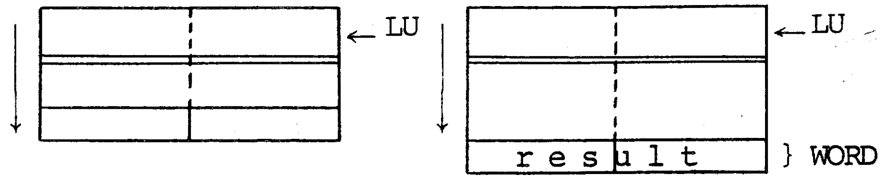
```
BEGIN
  pus:= word(stackbyte(gf + nextword));
END;
```

5.4.8 REVGW

REtrieve Value G(lobal) Word   Value: B3$_{\text{Hex}}$



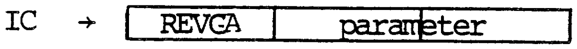
STACK BEFORE: STACK AFTER:

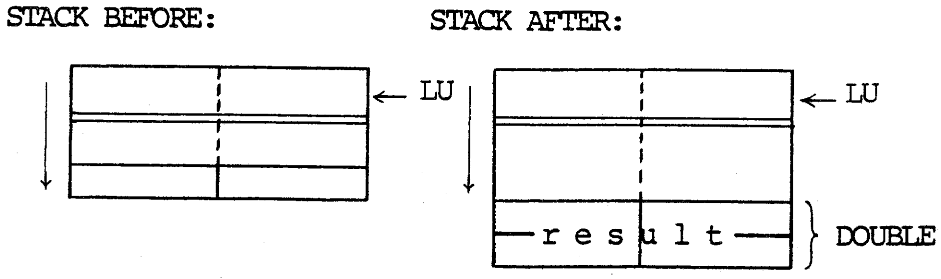


```
BEGIN
  pus:= stackword(gf + nextword);
END;
```

5.4.9 REVGA

REtrieve Value G(lobal) Address   Value: D3$_{Hex}$





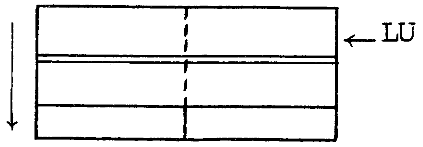
```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  pus:= word(stackbyte(gf + offset));
  pus:= stackword(gf + offset + 1);
END;
```

5.4.10 REVGD

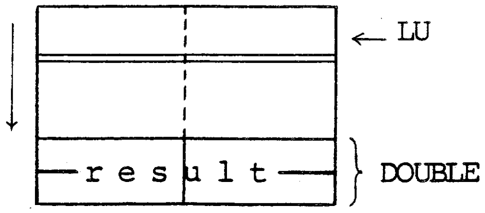
REtrieve Value G(lobal) Address Value: $F9_{Hex}$

IC → | REVGD | parameter |

STACK BEFORE:



STACK AFTER:

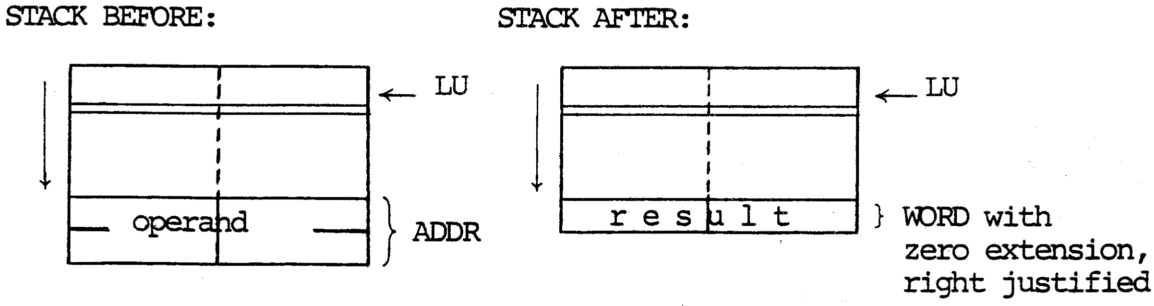


```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  pus:= stackword(gf + offset);
  pus:= stackword(gf + offset + 2);
END;
```

5.4.11 REVSB

REtrieve Value S(tack) Byte   Value: $B8_{Hex}$

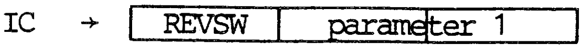
IP → | REVSB | parameter 1 |



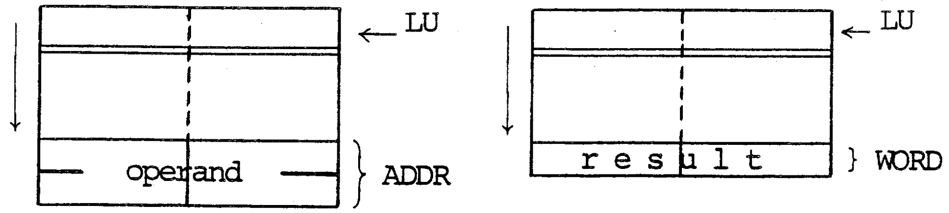
```
VAR
  adress : adr;
BEGIN
  adress.disp:=pop;
  adress.base:=basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  pus:= word(membyte(address + nextword));
END;
```

5.4.12 REVSW

REtrieve Value S(tack) Word   Value: BA_{Hex}



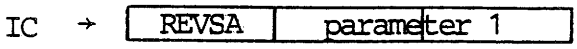
STACK BEFORE: STACK AFTER:

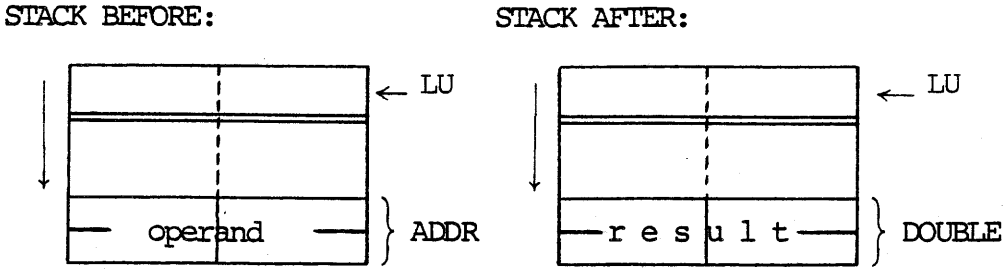


```
VAR
  adress : adr;
BEGIN
  adress.disp:= pop;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  pus:= memword(adress + nextword);
END;
```

5.4.13 REVSA

REtrieve Value S(tack) Address   Value: BC_{Hex}





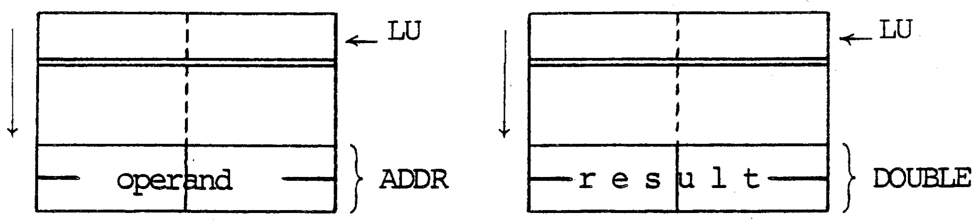
```
VAR
  adress : adr;
BEGIN
  adress.disp:= pop + nextword;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  pus:= word(membyte(adress));
  pus:= memword(adress + 1));
END;
```

5.4.14 REVSD

REtrieve Value S(tack) Double   Value: BE_{Hex}

IC → | REVSD | parameter 1 |

STACK BEFORE: STACK AFTER:



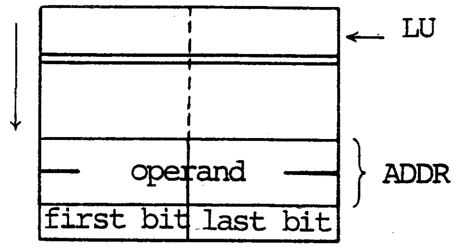
```
VAR
  adress : adr;
BEGIN
  adress.disp:= pop + nextword;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  pus:= memword(adress);
  pus:= memword(adress + 2));
END;
```

5.4.15 REVSF

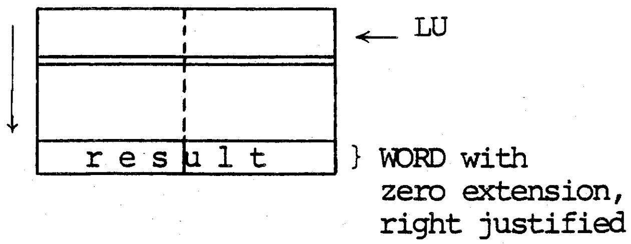
REtrieve Value S(tack) Field   Value: $D9_{Hex}$

IC → REVSF

STACK BEFORE:



STACK AFTER:

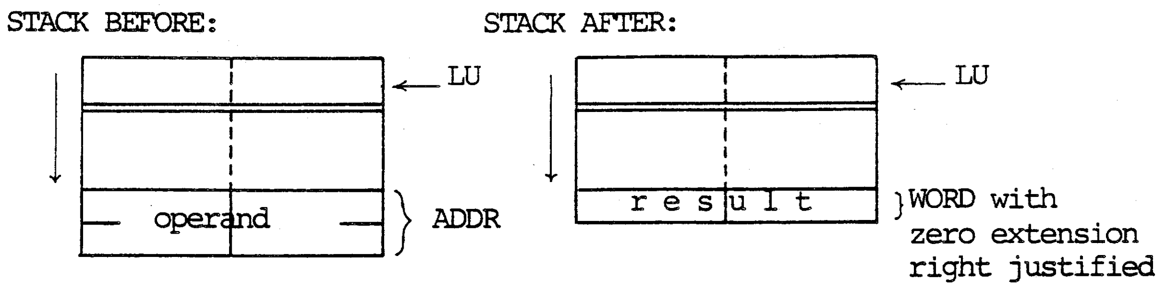


```
VAR
  field : RECORD
            firstbit : 0..15;
            lastbit : 0..15;
          END;
  adress : adr
  mask, result : word;
BEGIN
  WITH field AS word DO
  field:= pop;
  adress.disp:= pop;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1
    THEN exception(nilpointer);
  WITH field DO
  BEGIN
    IF firstbit > lastbit THEN exception(field);
    mask:= masks(lastbit-firstbit);
    result:= memword(adress);
    pus:= (result shift (lastbit-15)) AND mask;
  END;
END;
```

5.4.16 READB

READ Byte   Value: 90_{Hex}

IC → READB



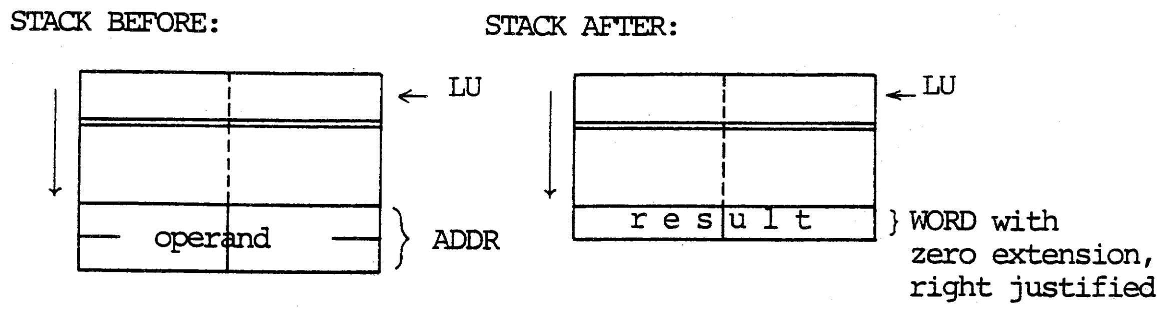
If a parity error occurs, the standard parity error procedure is suppressed. The actual result of the reading is delivered, besides the address is stored in the com8085 register set. The processing unit continues in run mode.

```
VAR
  adress : adr;
BEGIN
  adress.disp:= pop;
  adress.base:= basetype(pop);
  pus:= membyte(adress);
  IF parityerror
  THEN WITH com8085 DO BEGIN
    msgbase.base:= adress.base;
    msgdisp:= adress.disp;
    parityerror:= false;
  END;
END;
```

5.4.17 READW

READ Word Value: $B0_{Hex}$

IC → READW



If a parity error occurs, the standard parity error procedure is suppressed. The actual result of the reading is delivered, besides the address is stored in the com8085 register set. The processing unit continues in run mode.

```
VAR
  adress : adr;
BEGIN
  adress.disp:= pop;
  adress.base:= basetype(pop);
  pus:= mem(adress);
  IF parityerror
  THEN WITH com8085 DO BEGIN
    msgbase.base:= adress.base;
    msgdisp.disp:= adress.disp;
    parityerror:= false;
  END;
END;
```

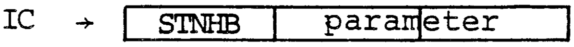
6. STORAGE OF A VALUE 6.

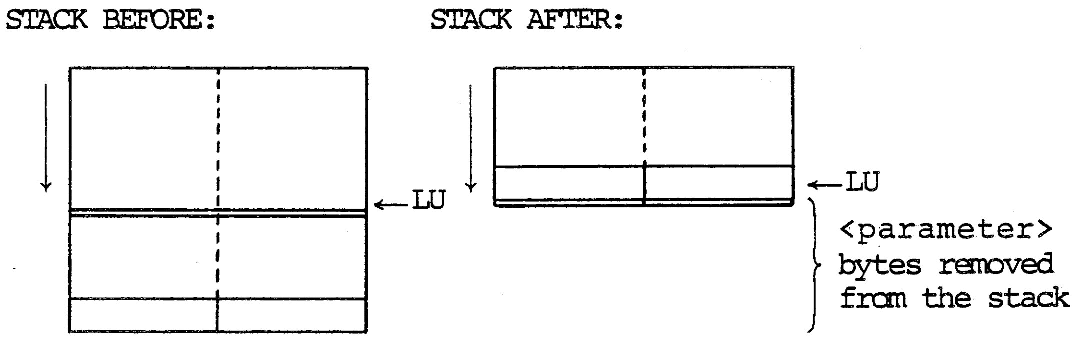
6.1 Pop Garbage 6.1

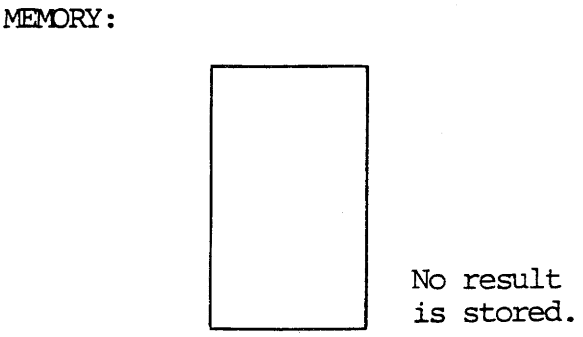
The operand is retrieved, the register stack moved to memory and <operand> bytes are removed from the stack. No result is stored.

6.1.1 STNHB 6.1.1

STore Nonsense H(ere) Byte Value: $8C_{Hex}$







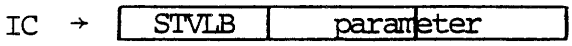
```
(* STNHB *)
VAR
  param : word;
BEGIN
  param:= nextword;
  IF odd(param) THEN exception(oddoperand);
  checkanddumpstack(0);
  lu:= lu - param ;
END;
```

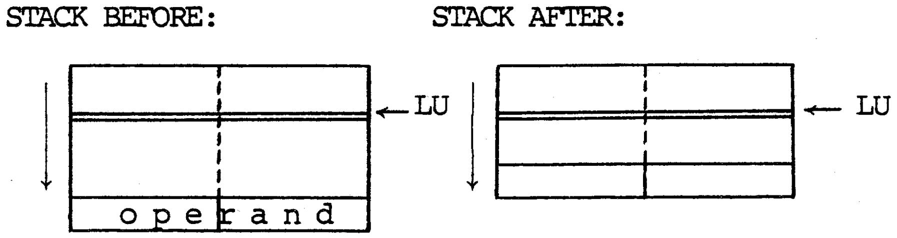
6.2 Pop Result 6.2

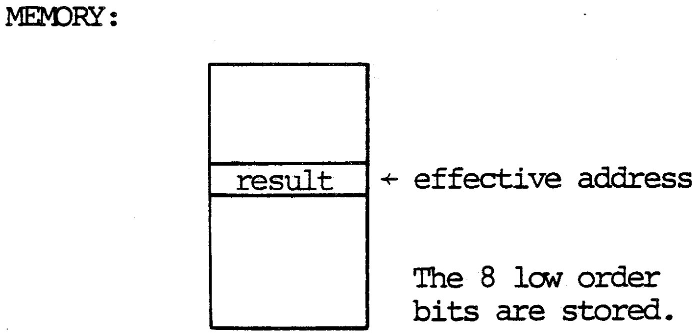
The operand is removed from the stack and stored as the result in the memory location defined by the effective address.

6.2.1 STVLB 6.2.1

STore Value L(ocal) Byte Value: 96_{Hex}



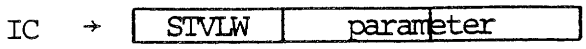


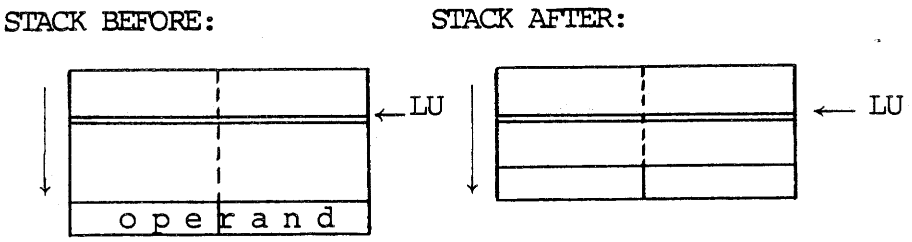


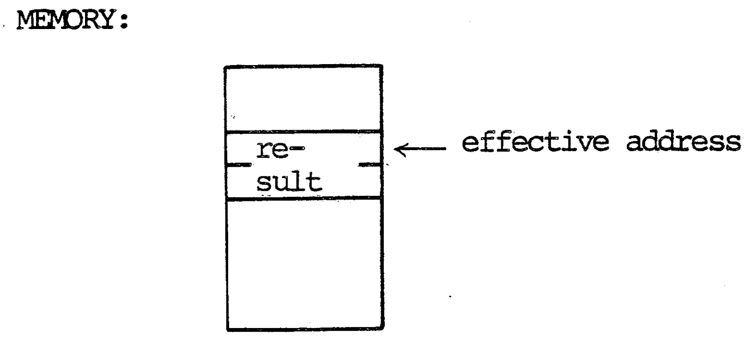
```
(* STVLB *)
BEGIN
  stackbyte(lf + nextword):= byte(pop)
END;
```

6.2.2 STVLW

STore Value L(ocal) Word   Value: $B6_{Hex}$



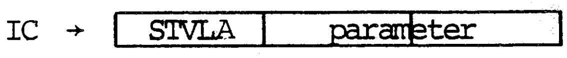




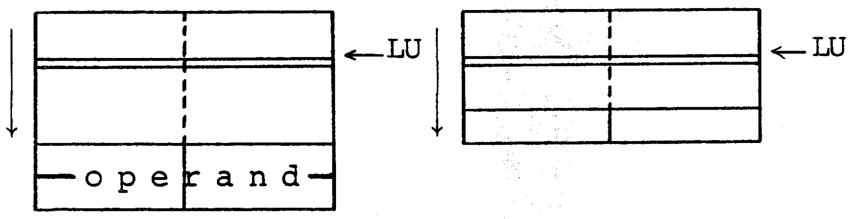
```
BEGIN
  stackword(lf + nextword):= pop;
END;
```

6.2.3 STVLA

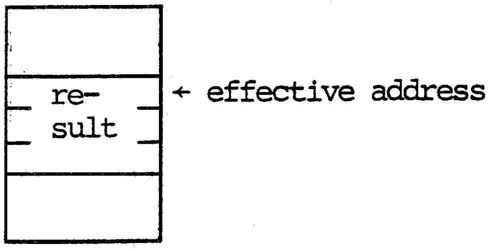
STore Value L(ocal) Address Value: $D6_{Hex}$



STACK BEFORE: STACK AFTER:



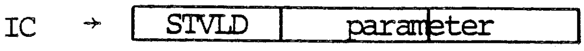
MEMORY:



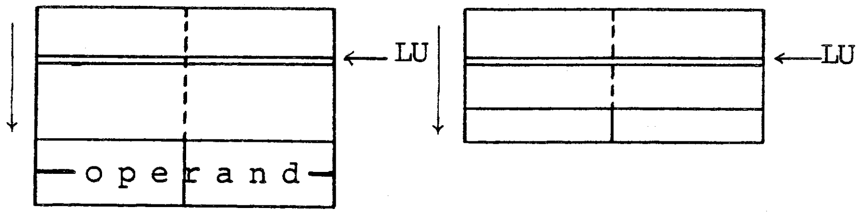
```
(* STVLA *)
VAR
  offset : word;
BEGIN
  offset:= nextword;
  stackword(lf + offset + 1):= pop;
  stackbyte(lf + offset):= byte(pop);
END;
```

6.2.4 STVLD 6.2.4

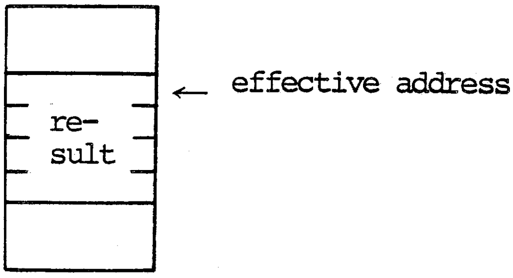
STore Value L(ocal) Double Value: F8$_{Hex}$



STACK BEFORE: STACK AFTER:



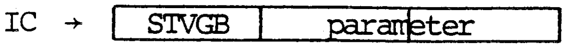
MEMORY:



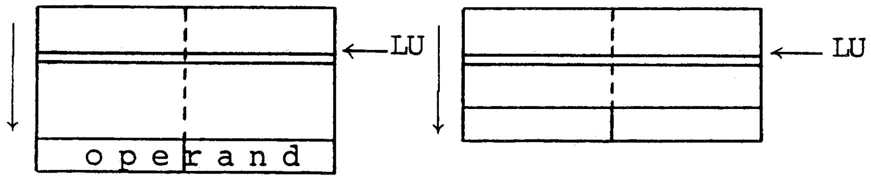
```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  stackword(lf + offset + 2):= pop;
  stackword(lf + offset):= pop;
END;
```

6.2.5 STVGB

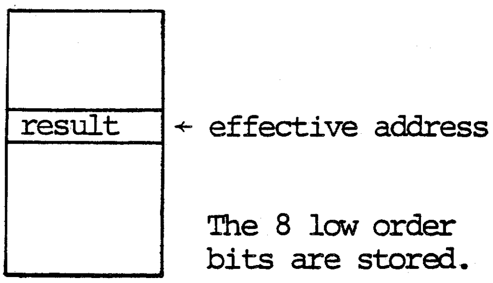
STore Value G(lobal) Byte Value: 92_{Hex}



STACK BEFORE: STACK AFTER:



MEMORY:

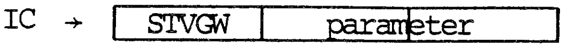


The 8 low order bits are stored.

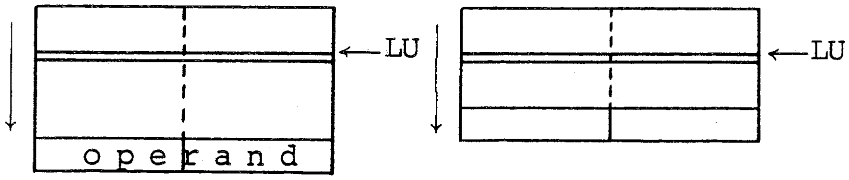
```
BEGIN
  stackbyte(gf + nextword):= byte(pop);
END;
```

6.2.6 STVGW

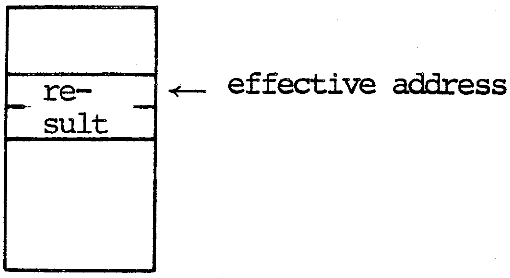
STore Value G(lobal) Word   Value: $B2_{Hex}$



STACK BEFORE: STACK AFTER:



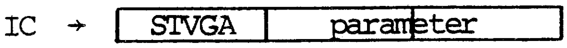
MEMORY:



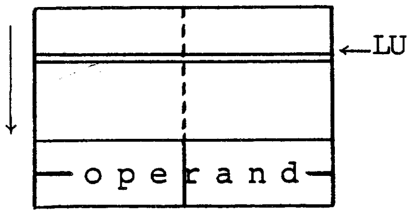
```
BEGIN
  stackword(gf + nextword):= pop;
END;
```

6.2.7 STVGA

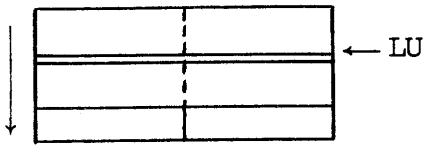
STore Value G(lobal) Address Value: D2$_{Hex}$



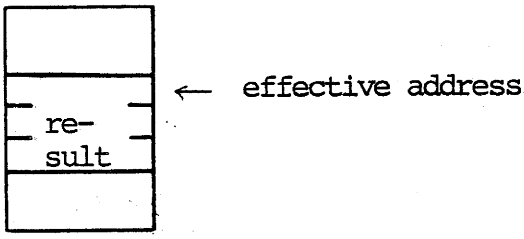
STACK BEFORE:



STACK AFTER:



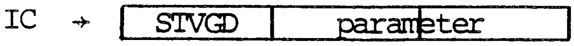
MEMORY:



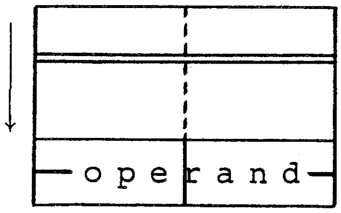
```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  stackword(lf + offset + 1):= pop;
  stackbyte(lf + offset):= byte(pop);
END;
```

6.2.8 STVGD 6.2.8

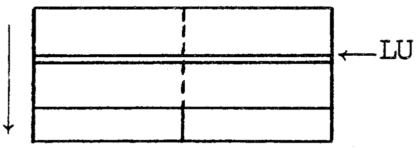
STore Value G(lobal) Double Value: FA_{Hex}



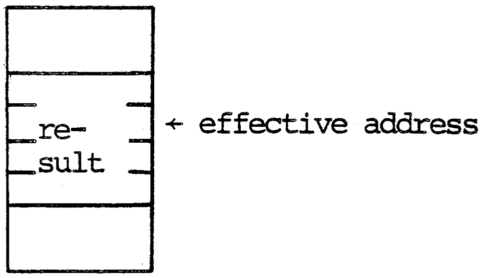
STACK BEFORE:



STACK AFTER:



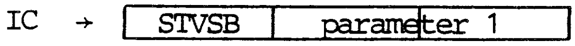
MEMORY:



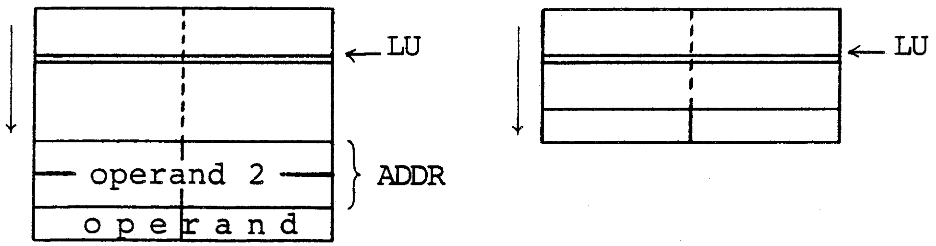
```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  stackword(lf + offset + 2):= pop;
  stackword(lf + offset):= pop;
END;
```

6.2.9 STVSB 6.2.9

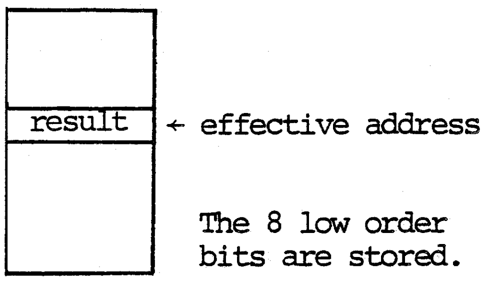
STore Value S(tack) Byte Value: 98_{Hex}



STACK BEFORE: STACK AFTER:



MEMORY:

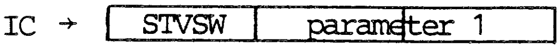


The 8 low order bits are stored.

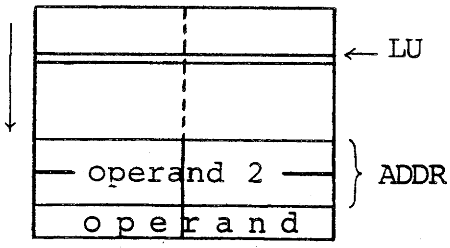
```
VAR
  arg : byte
  adress : adr;
BEGIN
  arg:= pop;
  adress.disp:= pop + nextword;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=l THEN exception(nilpointer);
  membyte(adress):= byte(arg);
END;
```

6.2.10 STVSW

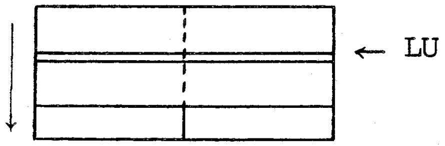
STore Value S(tack) Word   Value: $9A_{Hex}$



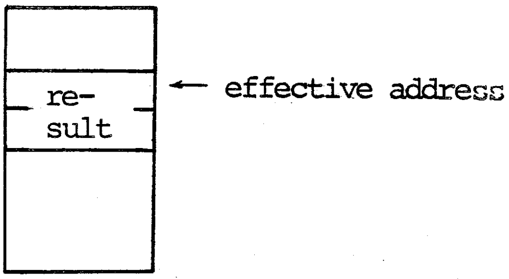
STACK BEFORE:



STACK AFTER:



MEMORY:



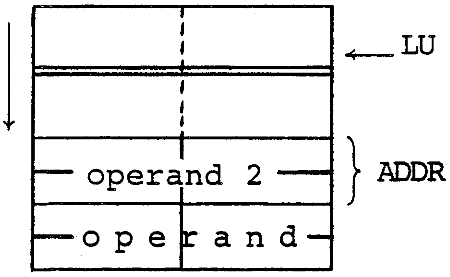
```
VAR
  arg : word;
  adress : adr;
BEGIN
  arg:= pop;
  adress.disp:= pop + nextword;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  memword(adress):= arg;
END;
```

6.2.11 STVSA

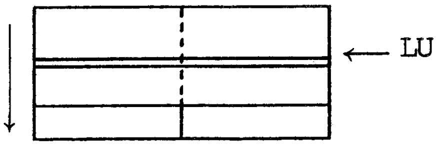
STore Value S(tack) Address Value: $9C_{Hex}$

IC → | STVSA | parameter 1 |

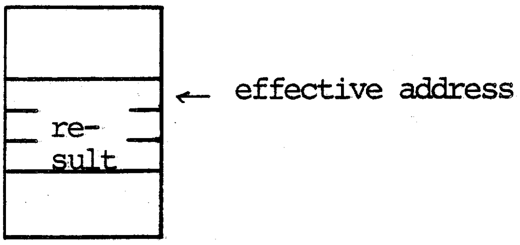
STACK BEFORE:



STACK AFTER:



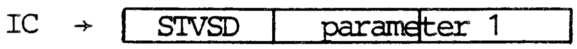
MEMORY:



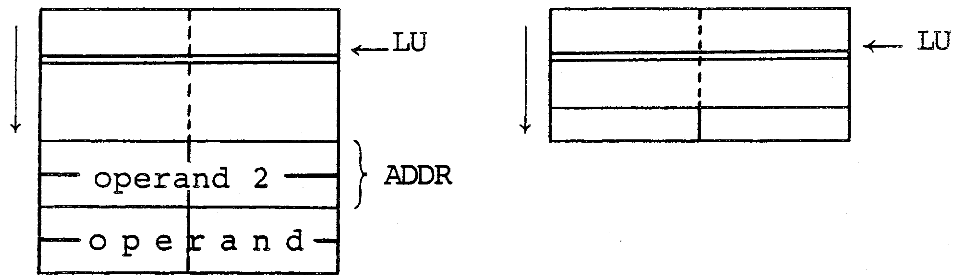
```
VAR
  operand1, operand2 : word;
  adress : adr;
BEGIN
  operand2:= pop;
  operand1:= pop;
  adress.disp:= pop + nextword;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  memword(adroffset(adress,1):= operand2;
  membyte(adress):= byte(operand1);
END;
```

6.2.12 STVSD

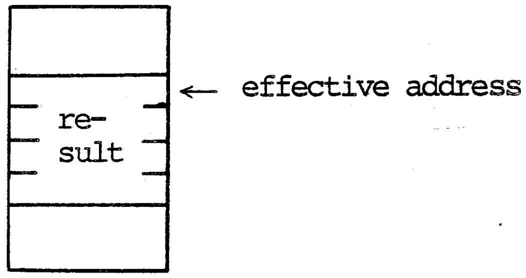
STore Value S(tack) Double Value: 9E$_{Hex}$



STACK BEFORE: STACK AFTER:



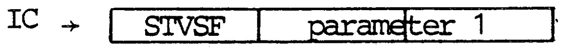
MEMORY:

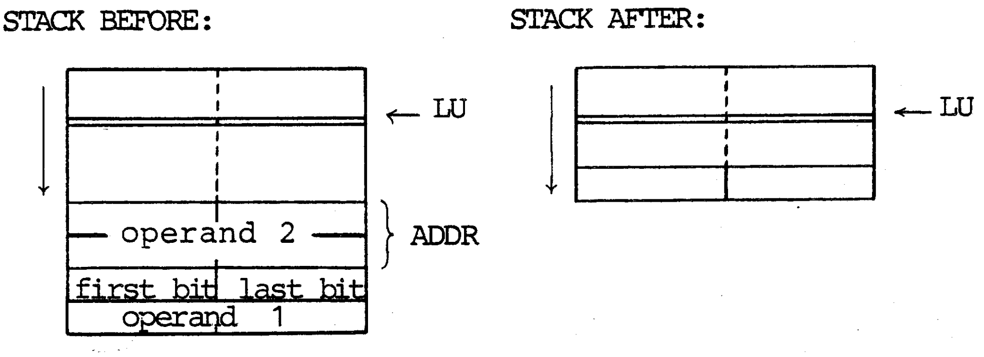


```
VAR
  operand1, operand2 : word;
  adress : adr;
BEGIN
  operand2:= pop;
  operand1:= pop;
  adress.disp:= pop + nextword;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  memword(adroffset(adress,2):= operand2;
  memword(adress):= operand1;
END;
```

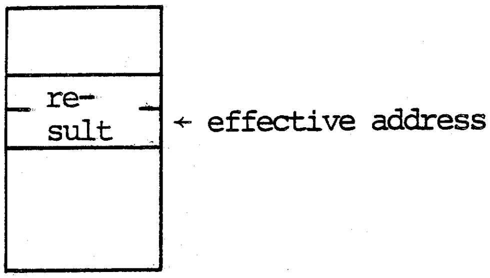
6.2.13 STVSF

STore Value S(tack) Field   Value: $D8_{Hex}$





MEMORY:



```
VAR
  operand : word
  field : record
            firstbit, lastbit : 0..15;
          end
  adress : adr;
  mask, result : word;
BEGIN
  operand:= pop;
  WITH field AS word DO field:= pop;
  adress.disp:= pop;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  WITH field DO
  BEGIN
    IF firstbit>lastbit THEN exception(fieldexception);
    result:= memword(adress);
    mask:= masks(lastbit-firstbit);
    IF operand-mask > 0 THEN exception(fieldexception);
    result:= result AND NOT (mask shift (15-lastbit));
    result:= result OR (operand shift (15-lastbit));
    memword(adress):= result;
  END;
END;
```

6.3 Manipulation of Storage Areas

6.3.1 Move a Storage Area

Parameter 1 is retrieved, and parameter 1 bytes from the operand (which is assumed to be a storage area of at least parameter 1 bytes) are moved to the result (which is assumed to be a storage area of at least parameter 1 bytes).

If parameter 1 is a large value, there will be one or more pauses in the execution of the instruction to permit interrupts.

Usually the move is performed through retrieving a number of bytes, and afterwards storing the retrieved bytes. To ensure integrity of the operation it is tested whether the destination area overlaps the source area or not. If it overlaps the move is performed one byte at a time. The overlap test is performed by the following routine:

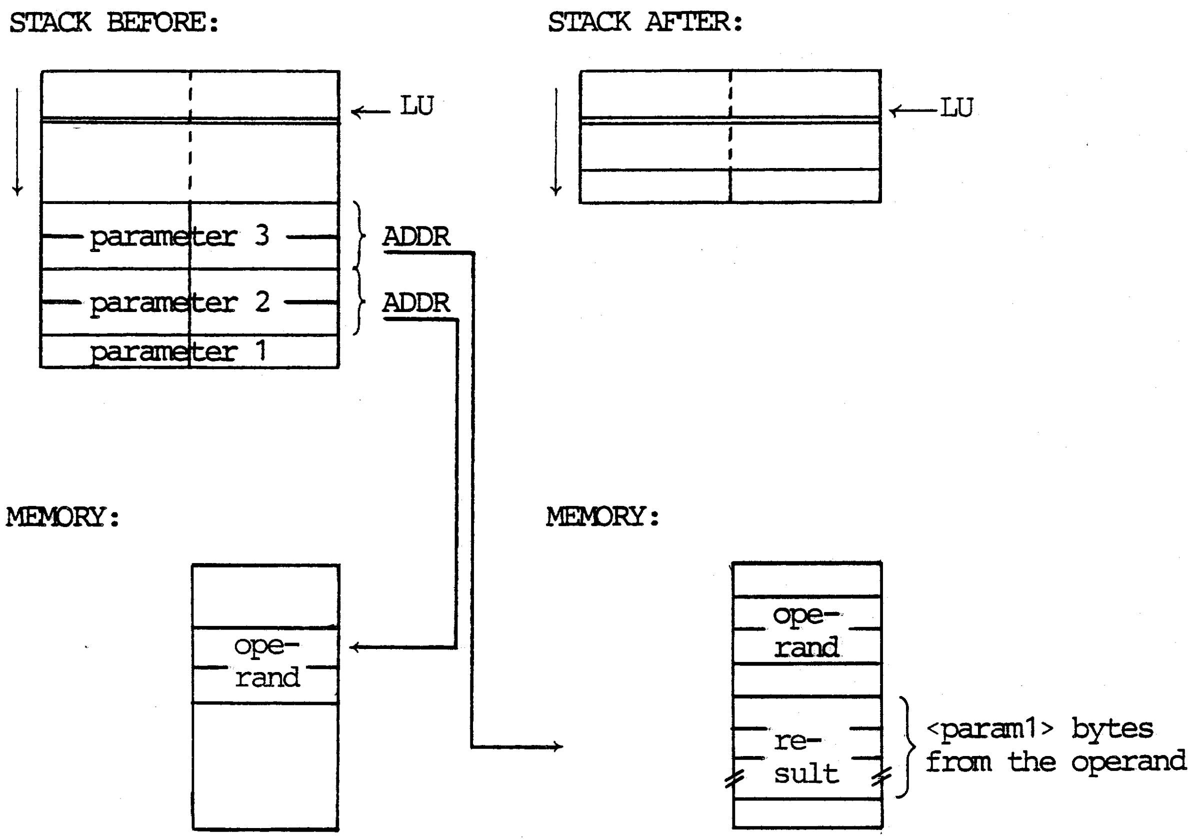
```
function overlapping(fadr,tadr: adr; length: word): boolean;
begin
  if fadr.base<>tadr.base then overlapping:= false
  else begin
    if fadr.disp>=tadr.disp then overlapping:= false
    else overlapping:= fadr.disp+length > tadr.disp;
  end;
end;
```

6.3.1.1 MOVEG

MOVE General

Value: AE_{Hex}

IC → MOVEG



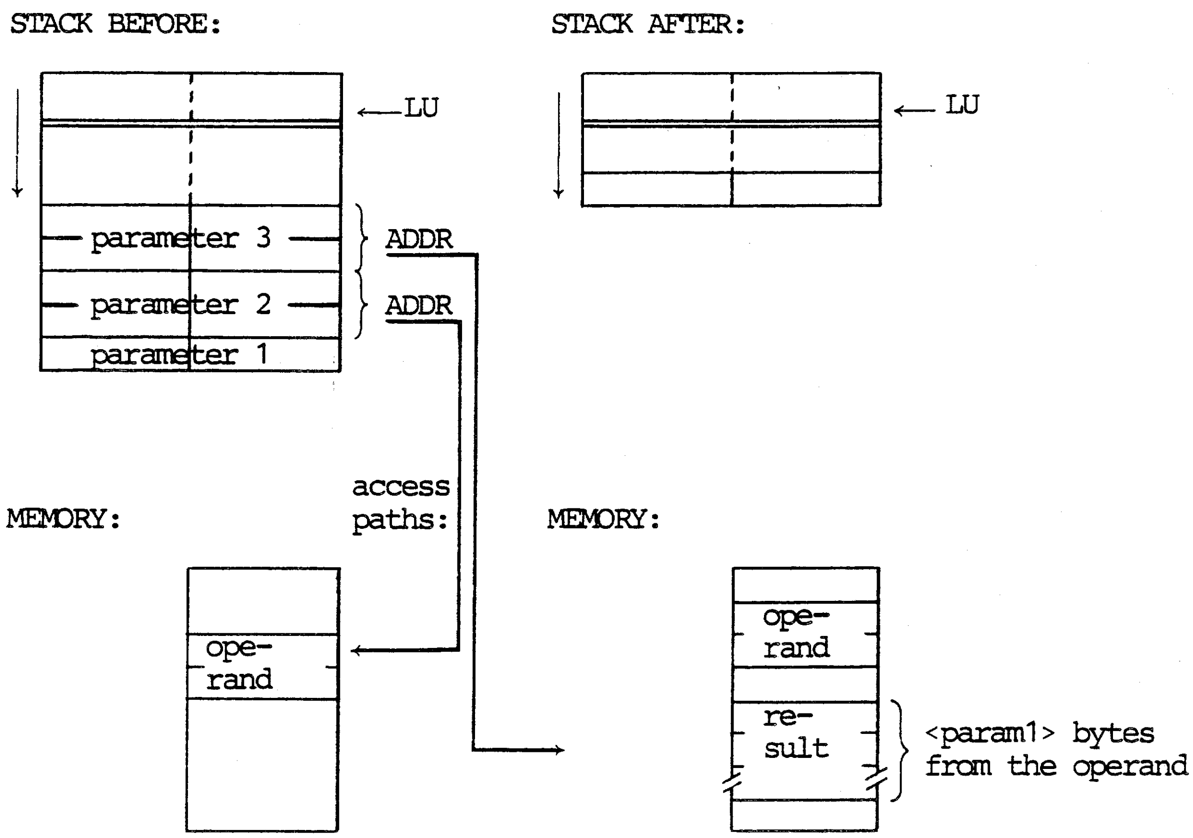
```
CONST
  wordblock = 8;
VAR
  length,i: word;
  fadress, tadress: adr;
  overlap: boolean;
BEGIN
  length:= pop;
  fadress.disp:= pop;
  fadress.base:= basetype(pop);
  tadress.disp:= pop;
  tadress.base:= basetype(pop);
  IF (fadress.base.nilbit=1) OR (tadress.base.nilbit=1)
    THEN exception(nilpointer);
  IF fadress.disp + length >= 64k THEN exception(wrapexception);
  IF tadress.disp + length >= 64k THEN exception(wrapexception);
  overlap:= overlapping(fadress,tadress,length);
  IF odd(tadress.disp)
  THEN BEGIN
    membyte(tadress):= membyte(fadress);
    fadress.disp:= fadress.disp + 1;
    tadress.disp:= tadress.disp + 1;
    length:= length - 1;
  END;
  WHILE (length > 1) AND NOT interrupt DO
  BEGIN
    IF length >= wordblock AND NOT overlap
    THEN BEGIN
      FOR i:=0 TO wordblock-1 DO
        mem(adroffset(tadress,2*i)):= memword(adroffset(fadress,2*i));
      fadress.disp:= fadress.disp + wordblock;
      tadress.disp:= tadress.disp + wordblock;
      length:= length - wordblock;
    END
    ELSE BEGIN
      IF overlap
      THEN BEGIN
        membyte(tadress):= membyte(fadress);
        fadress.disp:= fadress.disp + 1;
        tadress.disp:= tadress.disp + 1;
        length:= length - 1;
      END
      ELSE BEGIN
        WHILE length > 1 DO
        BEGIN
          mem(tadress):= memword(fadress);
          fadress.disp:= fadress.disp + 2;
          tadress.disp:= tadress.disp + 2;
          length:= length - 2;
        END;
      END;
    END;
    IF interrupt AND (length > 1)
    THEN BEGIN
      pus:= word(tadress.base);
      pus:= tadress.disp;
      pus:= word(fadress.base);
      pus:= fadress.disp;
      pus:= length;
      GOTO resumeinstruction;
    END
    ELSE BEGIN
      IF length = 1 THEN membyte(tadress):= membyte(fadress);
    END;
  END;
END;
```

6.3.1.2 MOVEB

MOVE Bytes

Value: AC_{Hex}

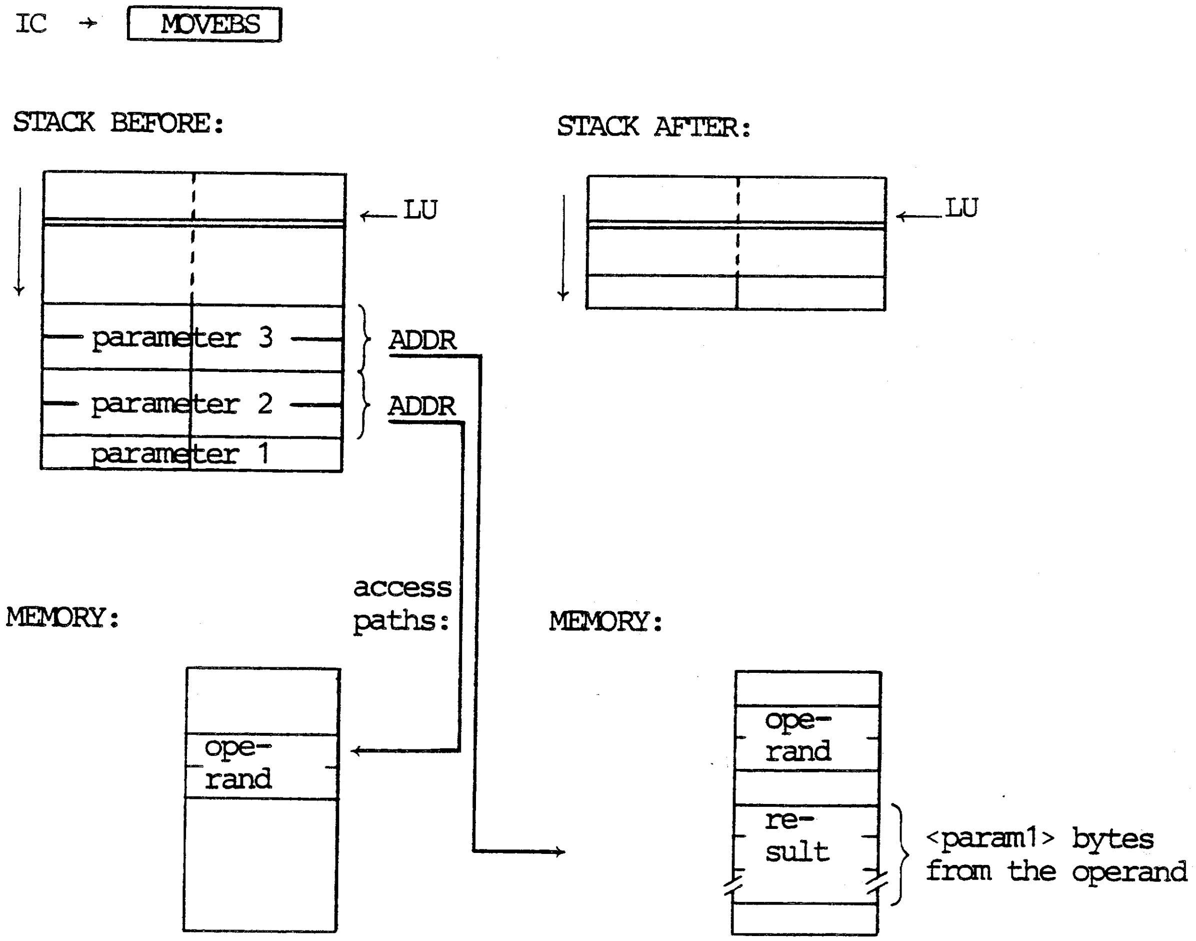
IC → MOVEB



```
CONST
  byteblock = 4;
VAR
  length,i: word;
  fadress, tadress: adr;
  overlap: boolean;
  copyreg: array(0..byteblock-1) of byte;
BEGIN
  length:= pop;
  fadress.disp:= pop;
  fadress.base:= basetype(pop);
  tadress.disp:= pop;
  tadress.base:= basetype(pop);
  IF (fadress.base.nilbit=1) OR (tadress.base.nilbit=1)
    THEN exception(nilpointer);
  IF fadress.disp + length > 64k THEN exception(wrapexception);
  IF tadress.disp + length > 64k THEN exception(wrapexception);
  overlap:= overlapping(fadress,tadress,length);
  WHILE length > 0 AND NOT interrupt DO
  BEGIN
    IF length >= byteblock AND NOT overlap
    THEN BEGIN
      FOR i:=0 TO byteblock-1 DO copyreg(i):= membyte(adroffset(fadress,i));
      FOR i:=0 TO byteblock-1 DO membyte(adroffset(tadress,i)):= copyreg(i);
      fadress.disp:= fadress.disp + byteblock;
      tadress.disp:= tadress.disp + byteblock;
      length:= length - byteblock;
    END
    ELSE BEGIN
      IF overlap
      THEN BEGIN
        membyte(tadress):= membyte(fadress);
        fadress.disp:= fadress.disp + 1;
        tadress.disp:= fadress.disp + 1;
        length:= length - 1;
      END
      ELSE BEGIN
        WHILE length > 0 DO
        BEGIN
          membyte(tadress):= membyte(fadress);
          fadress.disp:= fadress.disp + 1;
          tadress.disp:= tadress.disp + 1;
          length:= length - 1;
        END;
      END;
    END;
    IF interrupt AND (length > 0) DO
    THEN BEGIN
      pus:= word(tadress.base);
      pus:= tadress.disp;
      pus:= word(fadress.base);
      pus:= fadress.disp;
      pus:= length;
      GOTO resumeinstruction;
    END
  END;
END;
```

6.3.1.3 MOVEBS

MOVE Bytes Single   Value: 12_{Hex}



```
VAR
  length,i: word;
  fadress, tadress: adr;
  overlap: boolean;
BEGIN
  length:= pop;
  fadress.disp:= pop;
  fadress.base:= basetype(pop);
  tadress.disp:= pop;
  tadress.base:= basetype(pop);
  IF (fadress.base.nilbit=1) OR (tadress.base.nilbit=1)
    THEN exception(nilpointer);
  IF fadress.disp + length > 64k THEN exception(wrapexception);
  IF tadress.disp + length > 64k THEN exception(wrapexception);
  overlap:= overlapping(fadress,tadress,length);
  WHILE length > 0 AND NOT interrupt DO
  BEGIN
    membyte(tadress):= membyte(fadress);
    fadress.disp:= fadress.disp + 1;
    tadress.disp:= tadress.disp + 1;
    length:= length - 1;
  END;
    IF interrupt AND (length > 0) DO
    THEN BEGIN
      pus:= word(tadress.base);
      pus:= tadress.disp;
      pus:= word(fadress.base);
      pus:= fadress.disp;
      pus:= length;
      GOTO resumeinstruction;
    END
  END;
END;
```

6.3.2 Compare Two Storage Areas

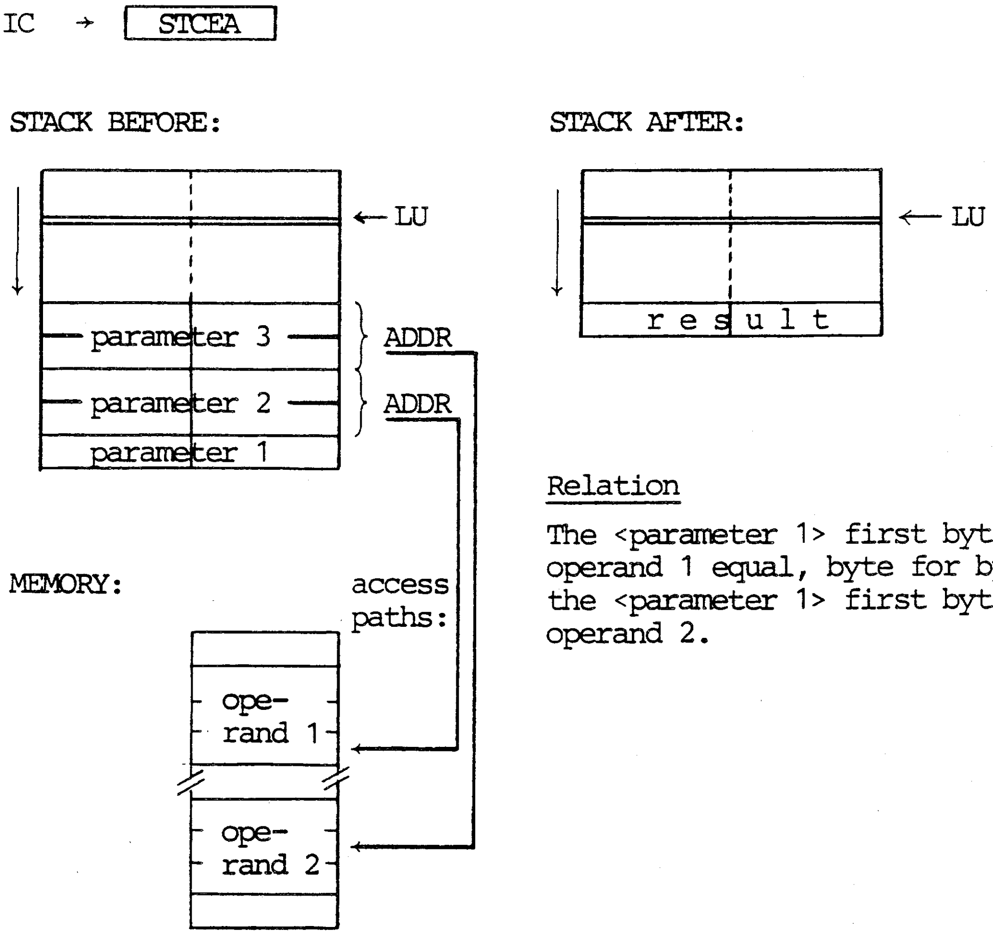
Parameter 1 is retrieved, and parameter 1 bytes from operand 1 (which is assumed to be a storage area of at least parameter 1 bytes) are compared with parameter 1 bytes from operand 2 (which is assumed to be a storage area of at least parameter 1 bytes).

If parameter 1 is a large value, there will be one or more pauses in the execution of the instruction to permit interrupts.

The operands are compared byte for byte. The result is the number of remaining bytes when the first difference between operand1 and operand2 is found. I.e. if operand1 equals operand2 the result is zero.
The carrybit is set according to the relation operand1 <= operand2.

6.3.2.1 STCEA

STorage Compare Equal Area   Value: EE_{Hex}



Relation

The <parameter 1> first bytes of operand 1 equal, byte for byte, the <parameter 1> first bytes of operand 2.

```
VAR
  length: word;
  fadress, tadress: adr;
BEGIN
  length:= pop;
  fadress.disp:= pop;
  fadress.base:= basetype(pop);
  tadress.disp:= pop;
  tadress.base:= basetype(pop);
  IF (tadress.base.nilbit=1) OR (fadress.base.nilbit=1)
    THEN exception(nilpointer);
  WHILE (length > 0) AND
         membyte(fadress) = membyte(tadress) AND
         NOT interrupt DO
  BEGIN
    fadress.disp:= fadress.disp + 1;
    tadress.disp:= tadress.disp + 1;
    length:= length - 1;
  END
  IF (length > 0) AND interrupt AND(membyte(fadress)=membyte(tadress))
  THEN BEGIN
    pus:= tadress.base;
    pus:= tadress.disp;
    pus:= fadress.base;
    pus:= fadress.disp;
    pus:= length;
    GOTO resumeinstruction;
  END
  ELSE BEGIN
    pus:= length;
    ps.carry:= (membyte(fadress)=membyte(tadress));
  END;
END;
```

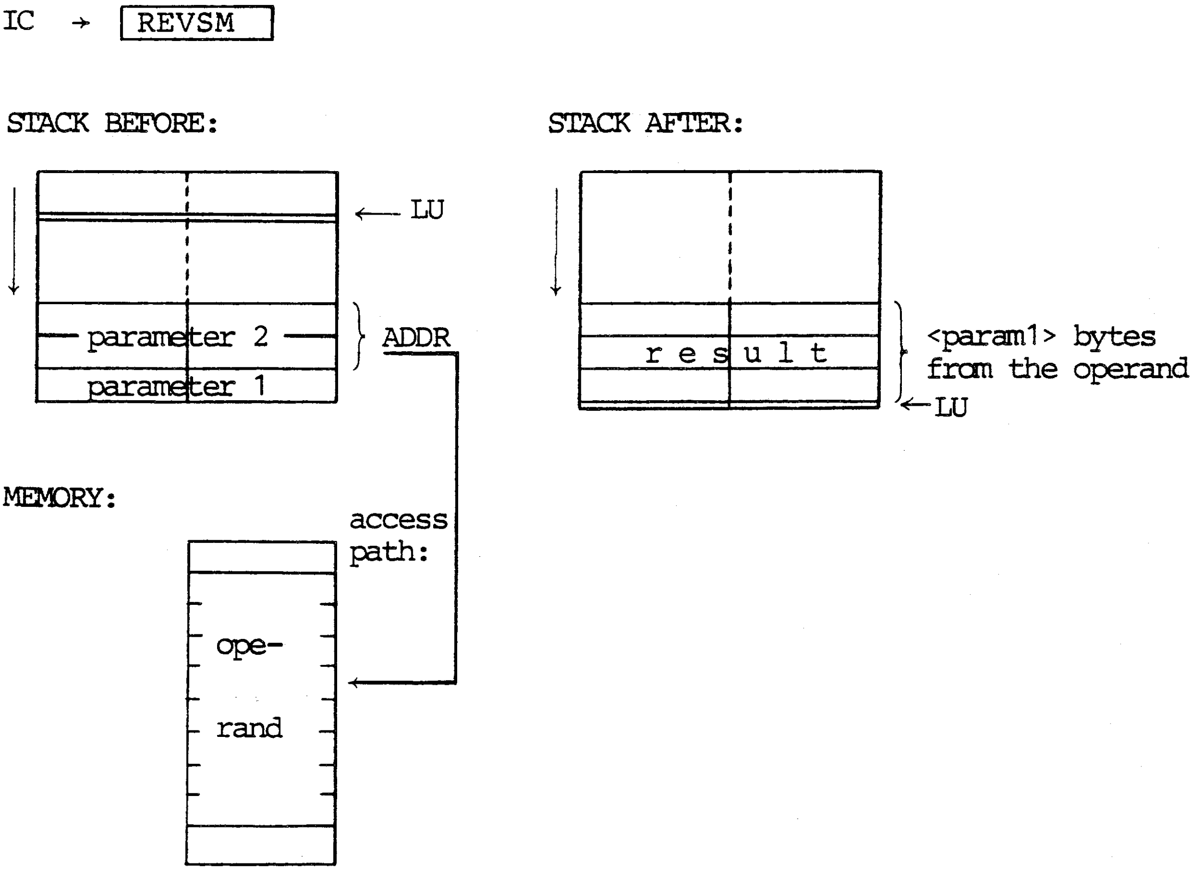
6.3.3 Push a Storage Area

Parameter 1 is retrieved, the register stack moved to memory and parameter 1 bytes from the operand (which is assumed to be a storage area of at least parameter 1 bytes) are pushed on the stack.

If parameter 1 is a large value, there will be one or more pauses in the execution of the instruction to permit interrupts.

6.3.3.1 REVSM

REtrieve Value Stack Multiple Value: CC_{Hex}

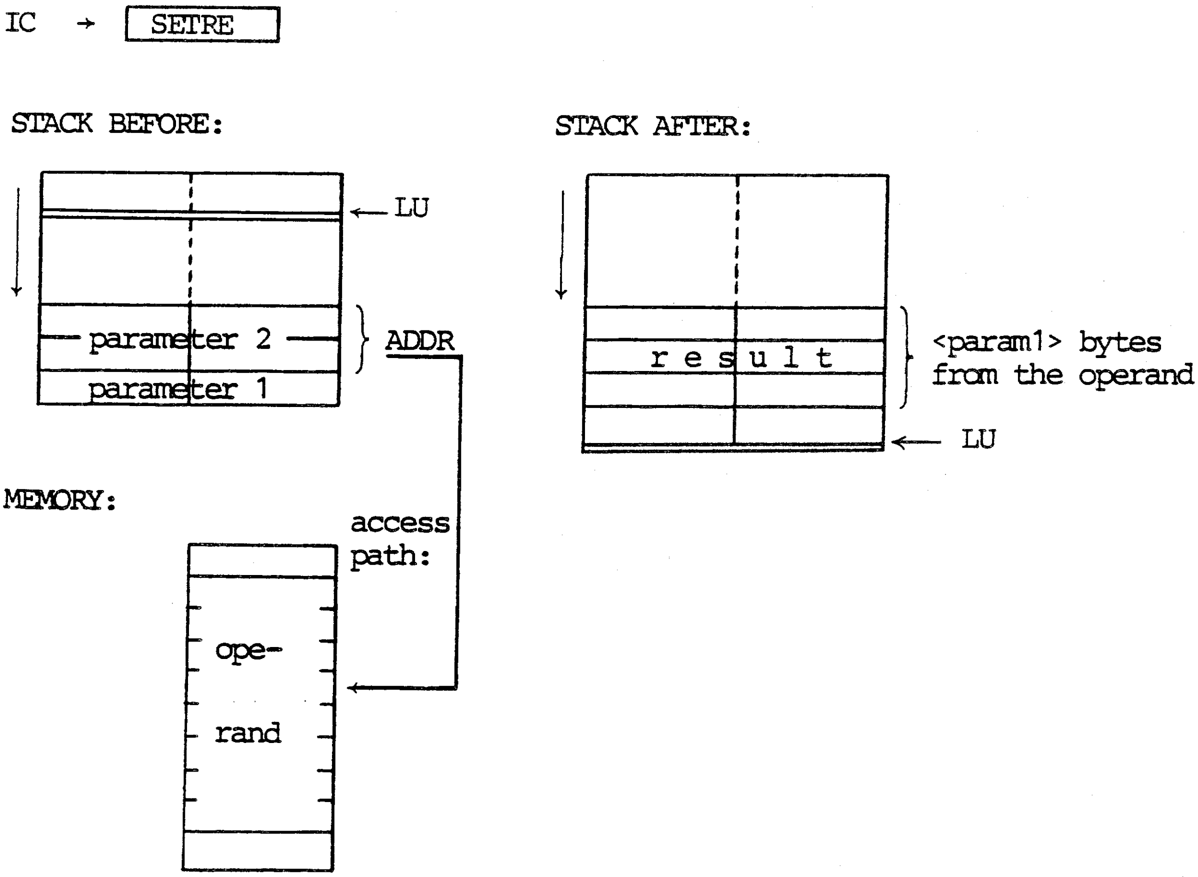


```
CONST
  wordblock = 8;
VAR
  length: word;
  tadress, fadress: adr;
BEGIN
  length:= pop;
  fadress.disp:= pop;
  fadress.base:= basetype(pop);
  IF (fadress.base.nilbit=1) THEN exception(nilpointer);
  checkanddumpstack(length);
  tadress.base:= sb.base;
  tadress.disp:= lu - length - 1;
  WHILE length > 1 AND NOT interrupt DO
  BEGIN
    IF (length >= wordblock) AND NOT overlap
    THEN BEGIN
      FOR i:= 0 TO wordblock-1 DO
        mem(adroffset(tadress,2*i)):= memword(adroffset(fadress,2*i));
      fadress.disp:= fadress.disp + wordblock;
      tadress.disp:= tadress.disp + wordblock;
      length:= length - wordblock;
    END
    ELSE BEGIN
      WHILE length > 1 DO
      BEGIN
        mem(tadress):= memword(fadress);
        fadress.disp:= fadress.disp + 2;
        tadress.disp:= tadress.disp + 2;
        length:= length - 2;
      END;
    END;
    IF interrupt AND (length > 1)
    THEN BEGIN
      (* revsm is resumed as moveg *)
      pus:= word(tadress.base);
      pus:= tadress.disp;
      pus:= word(fadress.base);
      pus:= fadress.disp;
      pus:= length;
      GOTO resumeinstruction;
    END
    ELSE BEGIN
      IF length = 1 THEN membyte(tadress):= membyte(fadress);
    END;
  END;
END;
```

6.3.3.2 SETRE

SET REtrieve

Value: 94_{Hex}



```
CONST
  wordblock = 8;
VAR
  length: word;
  fadress, tadress: adr;
BEGIN
  length:= pop;
  fadress.disp:= pop;
  fadress.base:= basetype(pop);
  IF (fadress.base.nilbit=1) THEN exception(nilpointer);
  checkanddumpstack(length+2);
  tadress.base:= sb.base;
  tadress.disp:= lu - length - 3;
  IF odd(length) THEN exception(oddoperand);
  mem(lu-1):= length;
  WHILE length > 1 AND NOT interrupt DO
  BEGIN
    IF length >= wordblock AND NOT overlap
    THEN BEGIN
      FOR i:=0 TO wordblock-1 DO
        mem(adroffset(tadress,2*i)):= memword(adroffset(fadress,2*i));
      fadress.disp:= fadress.disp + wordblock;
      tadress.disp:= tadress.disp + wordblock;
      length:= length - wordblock;
    END
    ELSE BEGIN
      WHILE length > 1 DO
      BEGIN
        mem(tadress):= memword(fadress);
        fadress.disp:= fadress.disp + 2;
        tadress.disp:= tadress.disp + 2;
        length:= length - 2;
      END;
    END;
    IF interrupt AND (length > 1)
    THEN BEGIN
      (* resumed as moveg *)
      pus:= word(tadress.base);
      pus:= tadress.disp;
      pus:= word(fadress.base);
      pus:= fadress.disp;
      pus:= length;
      GOTO resumeinstruction;
    END
    ELSE BEGIN
      IF length = 1 THEN membyte(tadress):= membyte(fadress);
    END;
  END;
END;
```

6.3.4 Pop a Storage Area

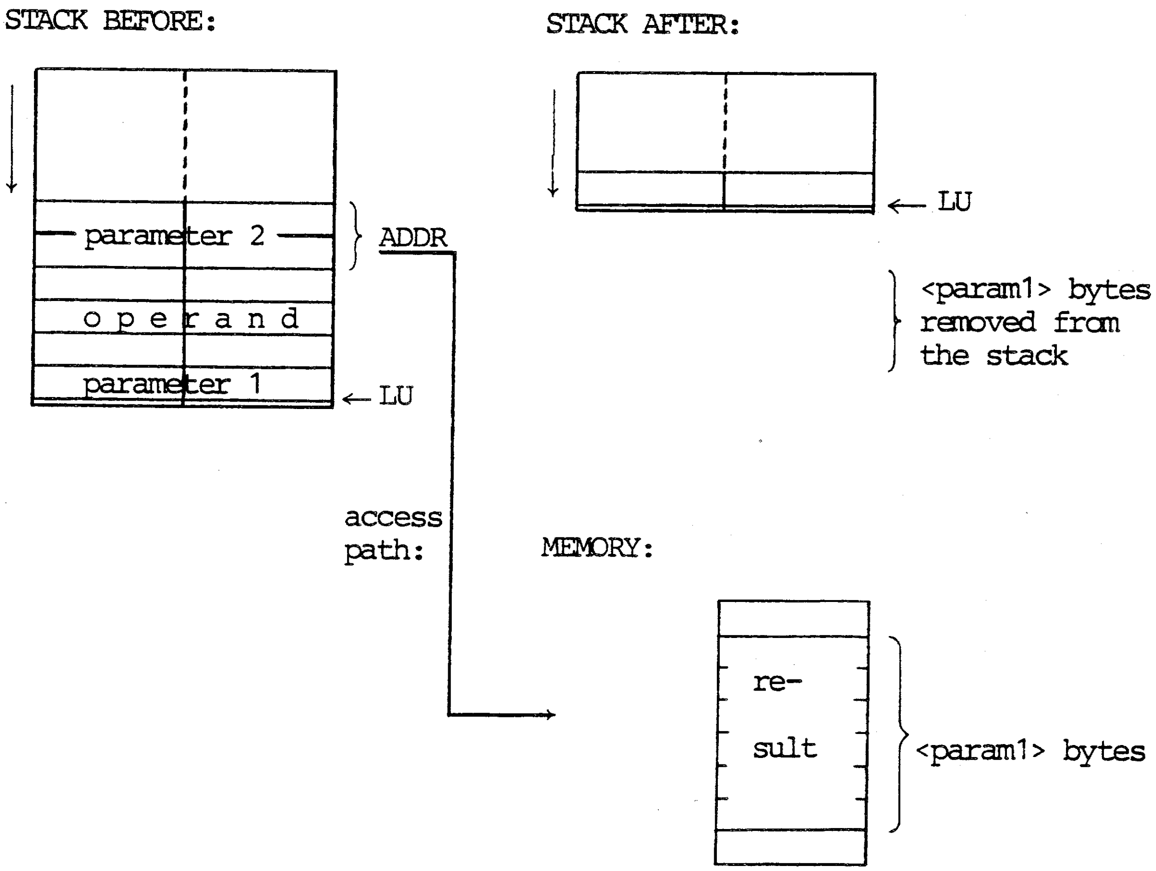
Parameter 1 is retrieved, and parameter 1 bytes are removed from the stack and stored as the result (which is assumed to be a storage area of at least parameter 1 bytes).

If parameter 1 is a large value, there will be one or more pauses in the execution of the instruction to permit interrupts.

6.3.4.1 SETST

SET STore Value: EC_{Hex}

IC → SETST



```
CONST
  wordblock = 8;
VAR
  length: word;
  fadress, tadress: adr;
BEGIN
  length:= mem(lu-1);
  fadress.base:= sb.base;
  fadress.disp:= lu - length - 1;
  tadress:= memadr(adroffset(fadress,-3););
  lu:= lu - length - 6;
  IF odd(length) THEN exception(oddoperand);
  IF (tadress.base.nilbit=1) THEN exception(nilpointer);
  WHILE length > 1 AND NOT interrupt DO
  BEGIN
    IF length >= wordblock AND NOT overlap
    THEN BEGIN
      FOR i:= 0 TO wordblock-1 DO
        mem(adroffset(tadress,2*i)):= memword(adroffset(fadress,2*i));
      fadress.disp:= fadress.disp + wordblock;
      tadress.disp:= tadress.disp + wordblock;
      length:= length - wordblock;
    END
    ELSE BEGIN
      WHILE length > 1 DO
      BEGIN
        mem(tadress):= memword(fadress);
        fadress.disp:= fadress.disp + 2;
        tadress.disp:= tadress.disp + 2;
        length:= length - 2;
      END;
    END;
    IF interrupt AND (length > 1)
    THEN BEGIN
      (* resumed as moveg *)
      pus:= word(tadress.base);
      pus:= tadress.disp;
      pus:= word(fadress.base);
      pus:= fadress.disp;
      pus:= length;
      GOTO resumeinstruction;
    END
    ELSE BEGIN
      IF length = 1 THEN membyte(tadress):= membyte(fadress);
    END;
  END;
END;
```

7. PROCEDURE CALL AND EXIT 7.

7.1 Enter a Routine 7.1

The execution of these instructions initiate a routine call. It assumes that the actual parameters have been calculated on the top of the stack.

(tegning 7.1)

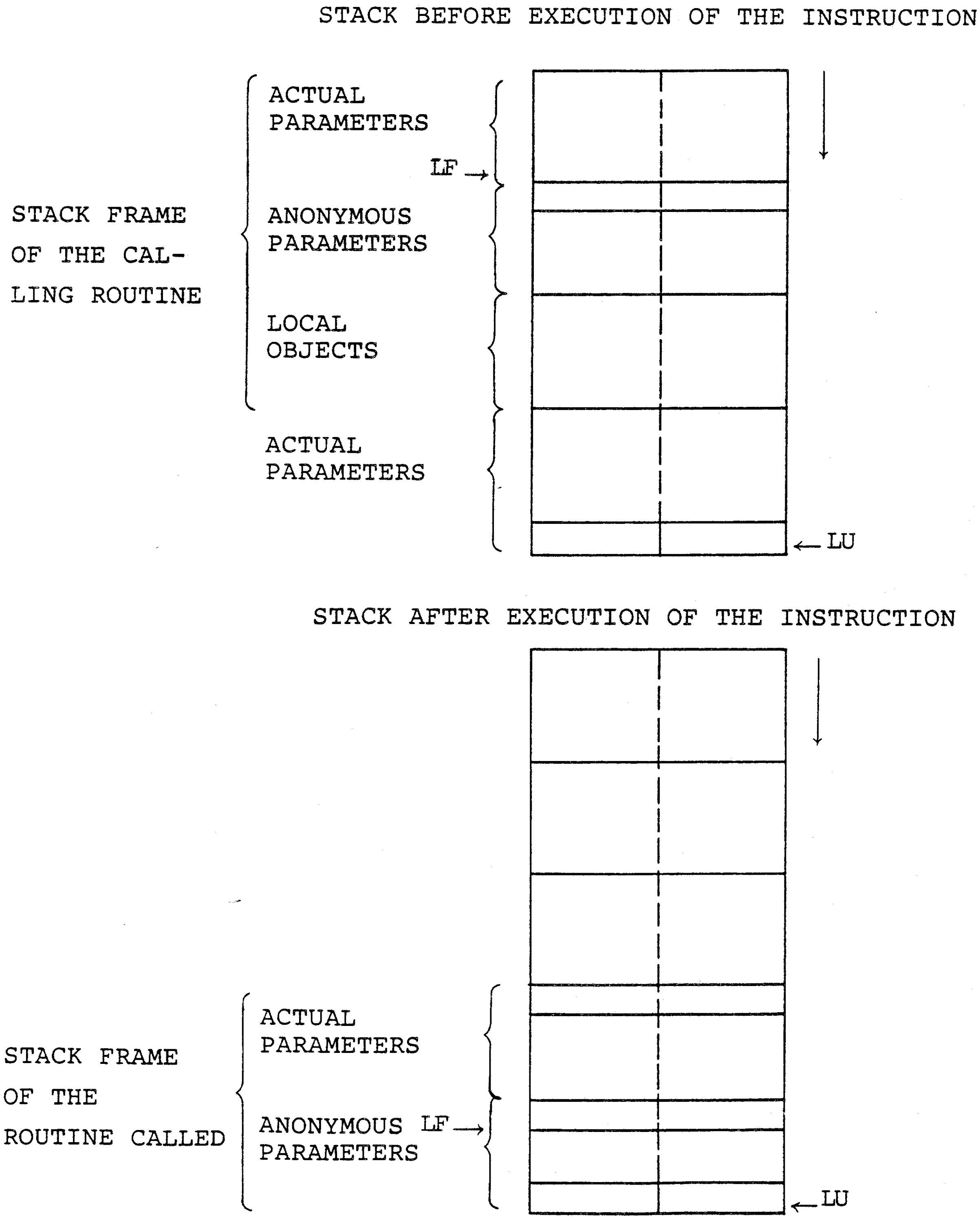

STACK BEFORE EXECUTION OF THE INSTRUCTION
ACTUAL PARAMETERS
LF
STACK FRAME OF THE CALLING ROUTINE
ANONYMOUS PARAMETERS
LOCAL OBJECTS
ACTUAL PARAMETERS
LU
STACK AFTER EXECUTION OF THE INSTRUCTION
STACK FRAME OF THE ROUTINE CALLED
ACTUAL PARAMETERS
ANONYMOUS PARAMETERS
LF
LU

Figure 29: Stack Frame.

The instructions have one parameter which specifies the entry point in the routine to be called.

The execution of the instructions have the following effect:

- The register stack is moved to memory.

- An area for the anonymous parameters is reserved and the contents are defined.

- The local frame pointer, LF, is set to address the first byte of the first anonymous parameter.

- The value of the parameter (an ADR) is assigned to the instruction pointer, (IB, IC).

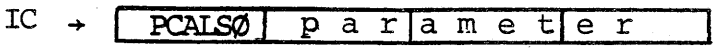
The anonymous parameters are:

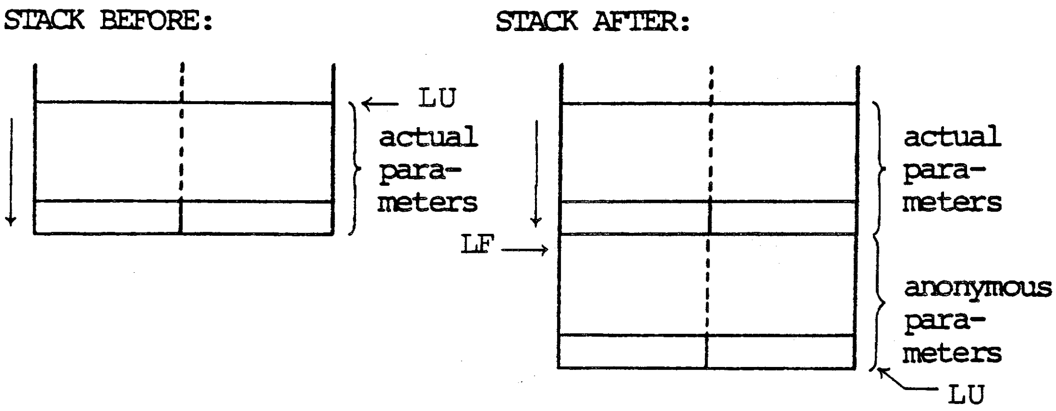
Static link pointer (displacement only)
Dynamic link pointer, i.e. old LF (displacement only)
Return point (base, displacement)

LF ->	static link
	dynamic link
	returnpoint base
	returnpoint disp

7.1.1 PCALS0

Procedure CALl Static level0 Value: 78_{Hex}



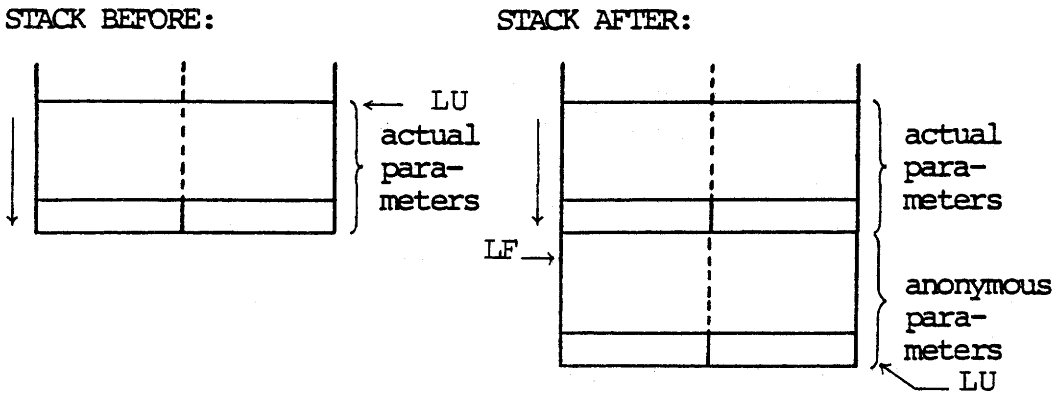


```
BEGIN
  checkanddumpstack(8);
  stack(lu - 7):= lf;
  stack(lu - 5):= lf;
  stack(lu - 3):= ib;
  stack(lu - 1):= ic;
  lf:= lu - 7;
  ib:= ibtype(nextbyte);
  ic:= nextword;
END;
```

7.1.2 PCALS1

Procedure CALl Static level1   Value: 79_{Hex}

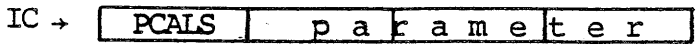
IC →	PCALS1	p a	r a m e	t e r



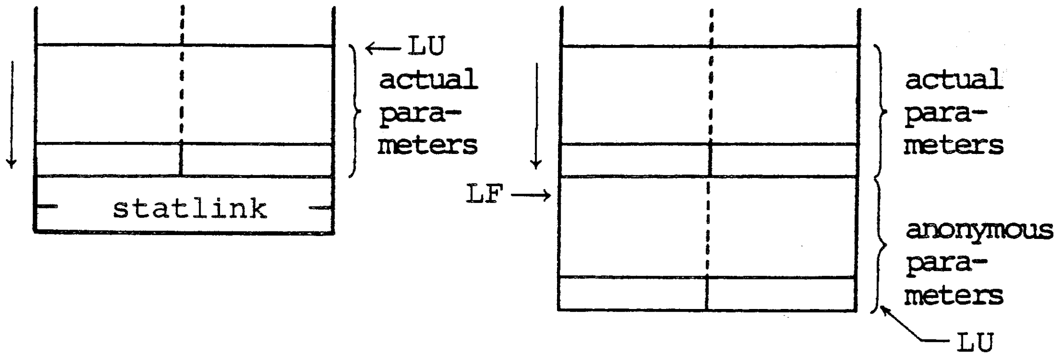
```
BEGIN
  checkanddumpstack(8);
  stack(lu - 7):= stack(lf);
  stack(lu - 5):= lf;
  stack(lu - 3):= ib;
  stack(lu - 1):= ic;
  lf:= lu - 7;
  ib:= ibtype(nextbyte);
  ic:= nextword;
END;
```

7.1.3 PCALS

Procedure CALl Static Value: $7A_{Hex}$



STACK BEFORE: STACK AFTER:



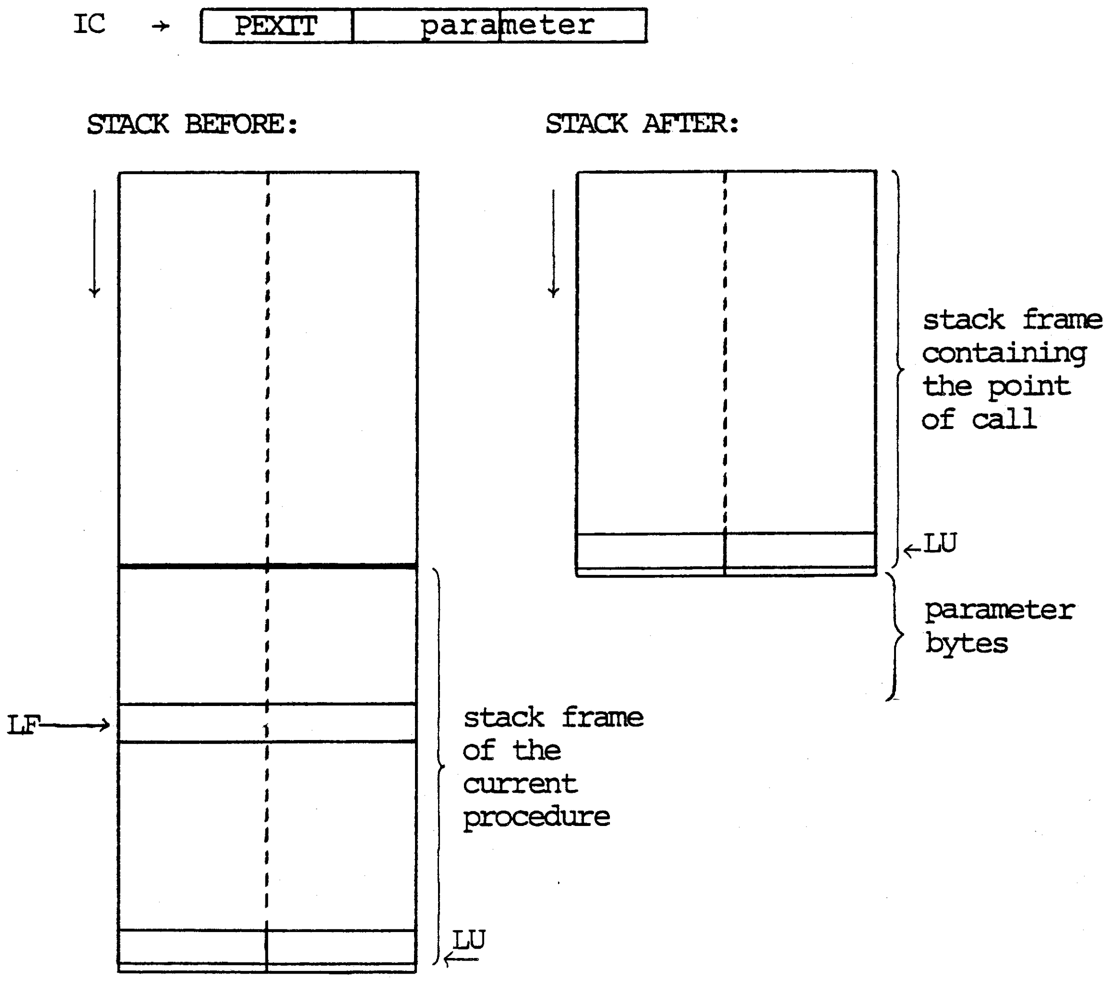
```
VAR
  statlink : adr;
BEGIN
  statlink.disp:= pop;
  pop; (* remove statlink.base *)
  pus:= statlink.disp;
  checkanddumpstack(6);
  stack(lu - 5):= lf;
  stack(lu - 3):= ib;
  stack(lu - 1):= ic;
  lf:= lu - 7;
  ib:= ibtype(nextbyte);
  ic:= nextword;
END;
```

7.2 Exit from a Routine 7.2

The execution of this instruction terminates the current routine call by returning to the point of call. The stack frame at the top of the stack is removed.

7.2.1 PEXIT 7.2.1

Procedure EXIT   Value: $7B_{Hex}$


IC →
PEXIT
parameter
STACK BEFORE:
STACK AFTER:
LF
LU
stack frame of the current procedure
stack frame containing the point of call
LU
parameter bytes

```
(* PEXIT *)
BEGIN
  lu:=lf-nextword;
  ib:= stack(ibtype(lf + 4));
  ic:= stack(lf + 6);
  lf:= stack(lf + 2);
END;
```

8. JUMPS 8.

8.1 Unconditional Jumps 8.1

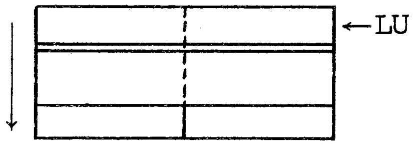
The operand is retrieved, and a result, which is interpreted as an ADR, is calculated and assigned to the instruction pointer, (IB, IC).

8.1.1 JMPHC 8.1.1

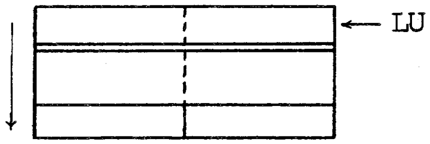
JuMP H (path) Constant Value: 69_{Hex}

IC →	JMPHC	para	meter

STACK BEFORE:



STACK AFTER:

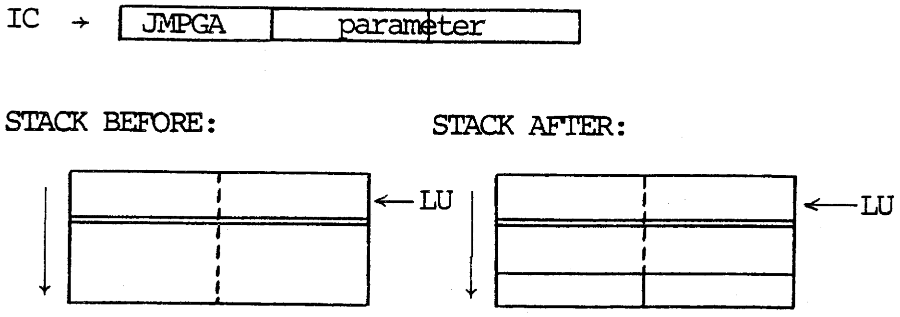


```
BEGIN
  ib:=ibtype(nextbyte);
  ic:=nextword;
END;
```

8.1.2 JMPGA

JuMP Global Address

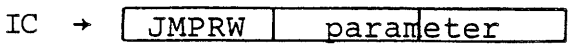
Value: $6A_{Hex}$

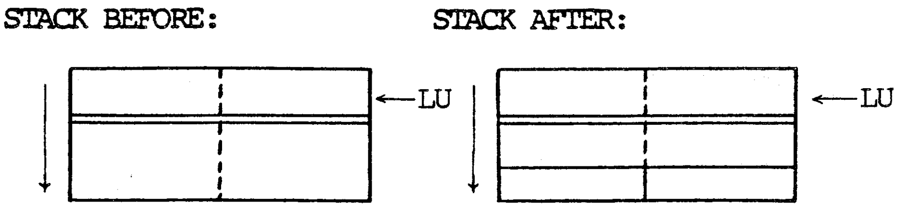


```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  ib:= ibtype(stackbyte(lf + offset));
  ic:= stackword(lf + offset + 1);
END;
```

8.1.3 JMPRW

JuMP Relative Word   Value: 68_{Hex}





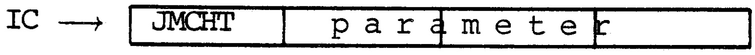
```
VAR
  offset : word;
BEGIN
  offset:= nextword;
  ic:= ic + offset;
END;
```

8.2 Case Jump 8.2

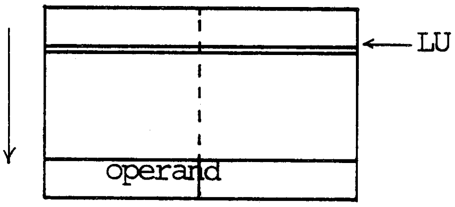
The parameter is interpreted as the start address (ADR) of a table consisting of a range descriptor and program points (ADR's). The operand is interpreted as an index to this table. The program point selected by the index is assigned to the instruction pointer, (IB, IC).

8.2.1 JMCHT 8.2.1

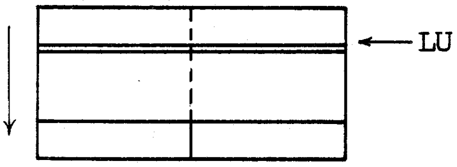
JuMp Case H (path) Table Value: $6B_{Hex}$

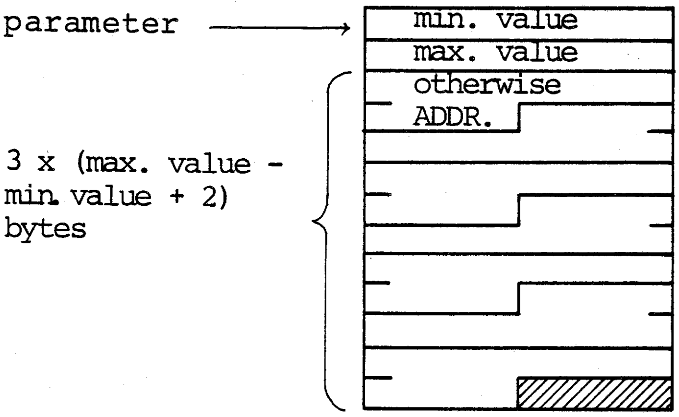


STACK BEFORE:



STACK AFTER:





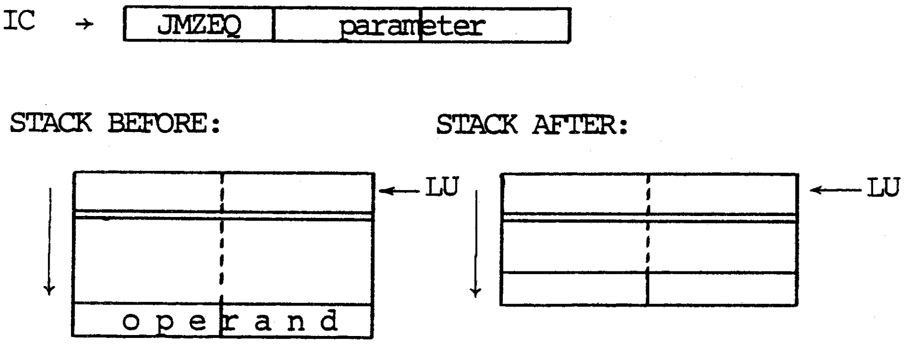
```
VAR
  dopeadress0, jmpadress : adr;
  operand : integer;
  lower, upper : integer;
BEGIN
  dopeadress.base:= basetype(nextbyte);
  dopeadress.disp:= nextword;
  operand:= pop;
  lower:= mem(dopeadress);
  upper:= mem(adroffset(dopeadress,2));
  IF (operand < lower) OR (operand>upper)
  THEN jmpadress:= memadr(adroffset(dopeadress,4));
  ELSE BEGIN
    operand:= operand - lower;
    jmpadress:= memadr(adroffset(dopeadress,7 + operand*3));
  END
  ib.base:= jmpadress.base;
  ic:= jmpadress.disp;
END;
```

8.3 Conditional Jumps with One Operand 8.3

The value of the operand is tested according to a relation (e.g. operand = 0). If the relation holds, a result is calculated and assigned to the instruction pointer, (IB, IC).

8.3.1 JMZEQ 8.3.1

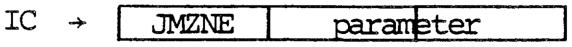
JuMp Zero EQual Value: 62_{Hex}



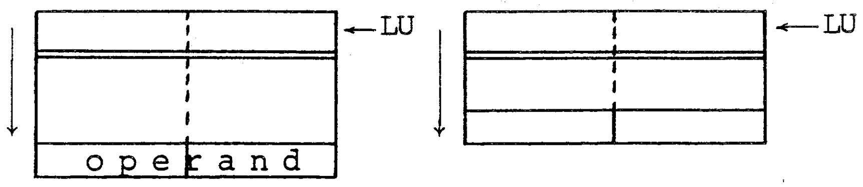
```
VAR
  offset : word;
  operand : integer;
BEGIN
  offset:= nextword;
  operand:= pop;
  IF operand = 0 THEN ic:= ic + offset
END;
```

8.3.2 JMZNE

JuMp Zero Not Equal   Value: 63_{Hex}



STACK BEFORE: STACK AFTER:

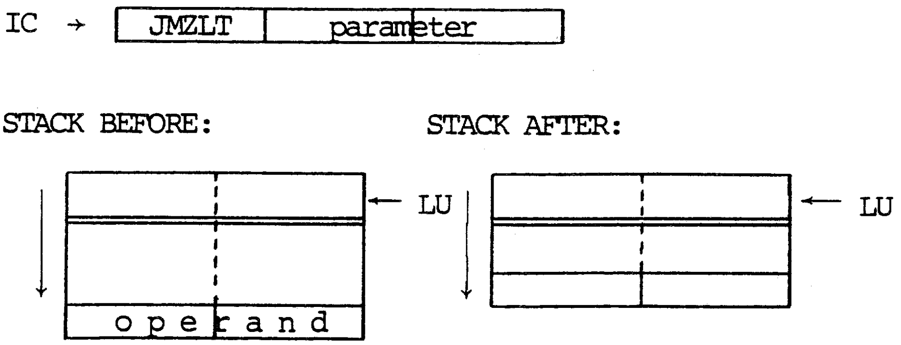


```
VAR
  offset : word;
  operand : integer;
BEGIN
  offset:= nextword;
  operand:= pop;
  IF operand <> 0 THEN ic:= ic + offset;
END;
```

8.3.3 JMZLT

JuMp Zero Less Than

Value: 64_{Hex}

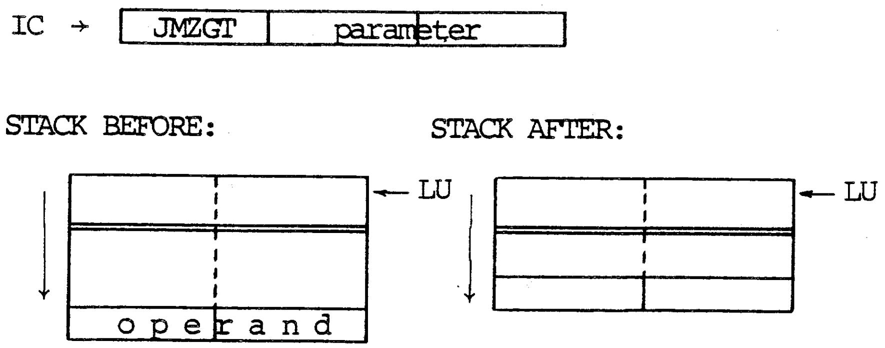


```
VAR
  offset : word;
  operand : integer;
BEGIN
  offset:= nextword;
  operand:= pop;
  IF operand < 0 THEN ic:= ic + offset;
END;
```

8.3.4 JMZGT

JuMp Zero Greater Than

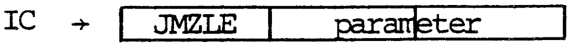
Value: 65_{Hex}



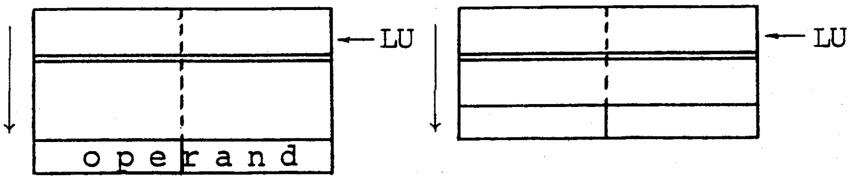
```
VAR
  offset : word;
  operand : integer;
BEGIN
  offset:= nextword;
  operand:= pop;
  IF operand > 0 THEN ic:= ic + offset;
END;
```

8.3.5 JMZLE

JuMp Zero Less Than or Equal   Value: 66_{Hex}



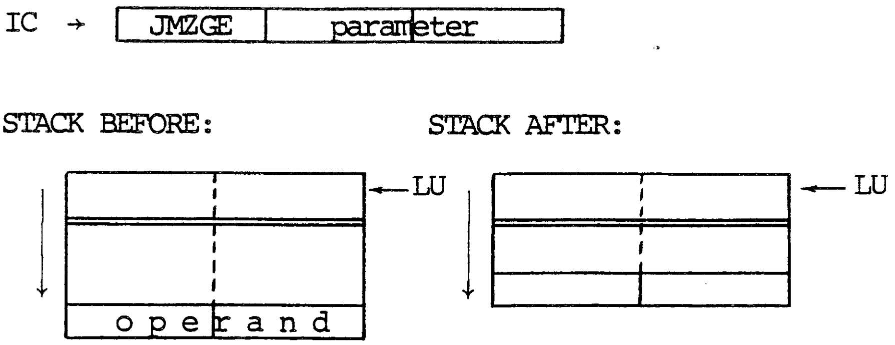
STACK BEFORE: STACK AFTER:



```
VAR
  offset : word;
  operand : integer;
BEGIN
  offset:= nextword;
  operand:= pop;
  IF operand <= 0 THEN ic:= ic + offset;
END;
```

8.3.6 JMZGE

JuMp Zero Greater Than or Equal   Value: 67_{Hex}



```
VAR
  offset : word;
  operand : integer;
BEGIN
  offset:= nextword;
  operand:= pop;
  IF operand >= 0 THEN ic:= ic + offset;
END;
```

9. OPERATORS 9.

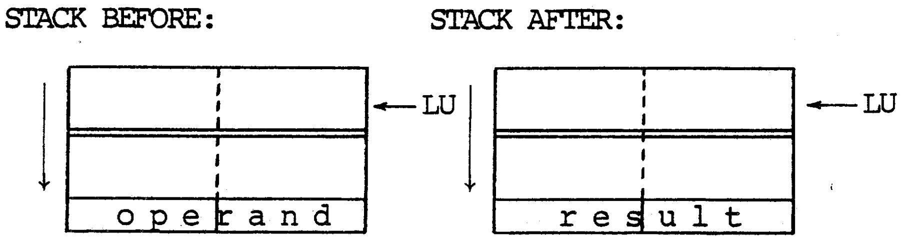
9.1 Monadic Operators 9.1

A single operand is retrieved, and a result is produced from this operand in accordance with the operator.

9.1.1 NEG 9.1.1

NEGate   Value: 50_{Hex}

IC → NEG


STACK BEFORE:

STACK AFTER:

operator: negative (monadic minus)

The operand is interpreted as a signed integer.

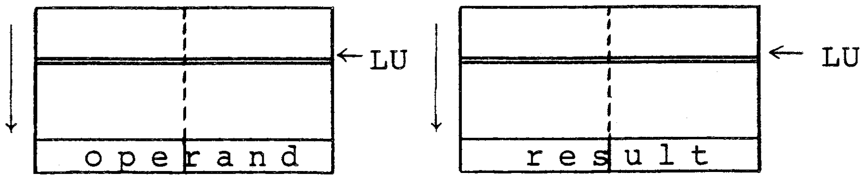
The result is the twos complement of the operand.

```
VAR
  result : integer;
BEGIN
  result:=-pop;
  IF overflow THEN exception(overflow);
  pus:=result;
END;
```

9.1.2 NOT

NOT   Value: 55_{Hex}

STACK BEFORE: STACK AFTER:



operator: not

The operand is interpreted as a boolean value.

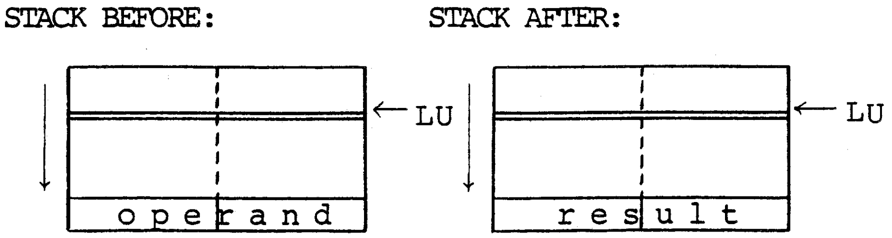
The result is true (if the operand is false) or false (if the operand is true).

```
BEGIN
  IF (pop AND 1) = 1
  THEN pus:= false
  ELSE pus:= true;
END;
```

9.1.3 ABS

ABSsolute Value   Value: 51_{Hex}

IC → $\boxed{\text{ABS}}$



operator: absolute value

The operand is interpreted as a signed integer.

The result is the absolute value of the operand.

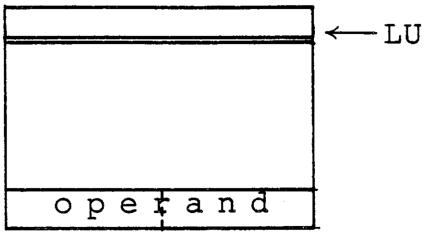
```
VAR
  operand : integer;
BEGIN
  operand:= pop;
  IF operand < 0
  THEN BEGIN
    operand:=-operand;
    IF overflow THEN exception(overflow);
  END;
  pus:=operand;
END;
```

9.1.4 SHC8

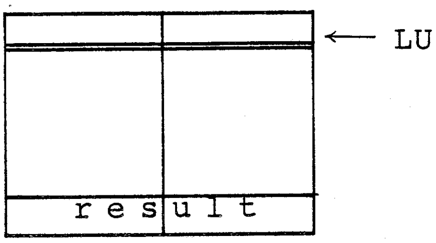
SHift Cyclic 8 Value: 54_{Hex}

IC ⟶ SHC 8

STACK BEFORE:



STACK AFTER:



```
VAR
  val : word
  byt : byte;
BEGIN
  val:= pop;
  WITH val AS array(0..1) of byte DO
  BEGIN
    byt:= val(0); val(0):= val(1);
    val(1):= byt;
  END;
END;
```

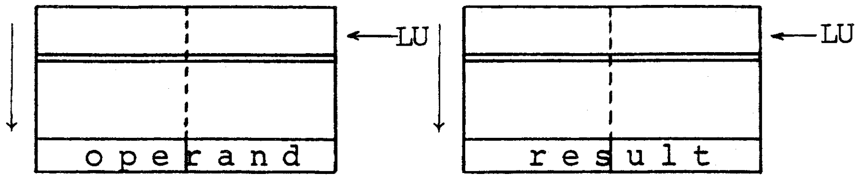
9.1.5 COMPL

COMPLement

Value: 52_{Hex}

IC → COMPL

STACK BEFORE: STACK AFTER:



operator: ones complement

The operand is interpreted as a signed integer.

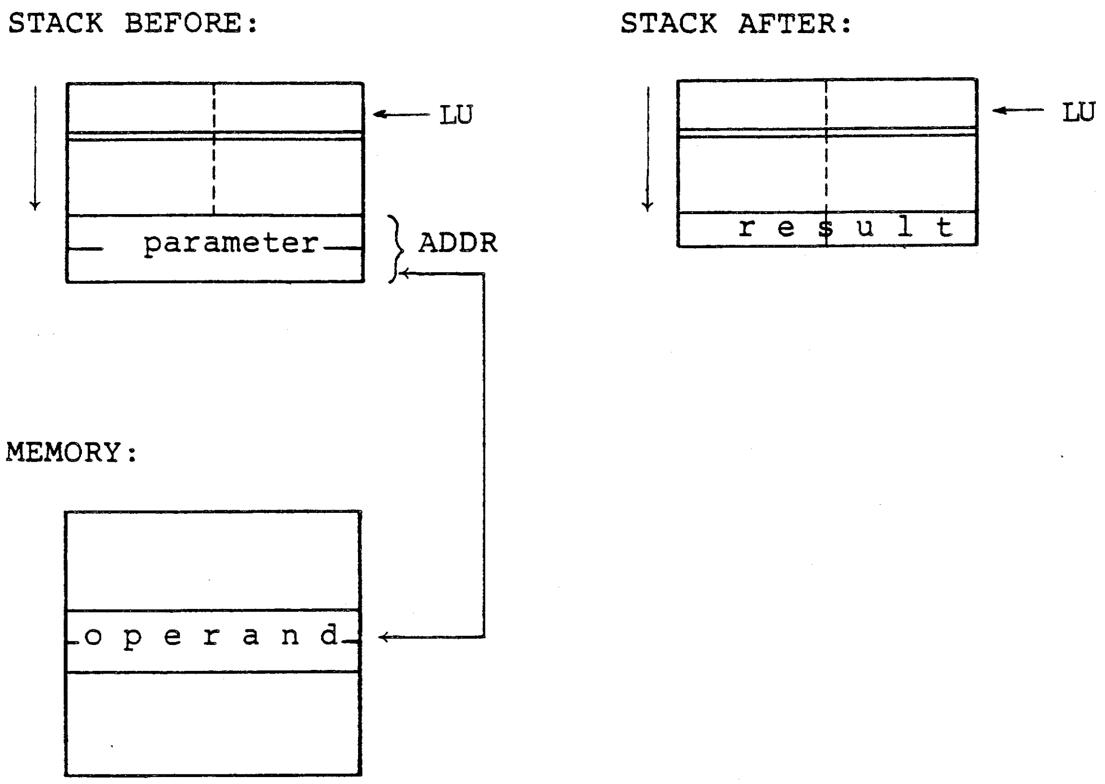
The result is the ones complement of the operand.

```
VAR
  i : integer;
  operand : word;
BEGIN
  operand:= pop;
  WITH operand AS bitword DO
    FOR i:= 0 TO 15 DO operand(i):=1 - operand(i);
  pus:=operand;
END;    (* COMPL *)
```

9.1.6 TNILL

Test NILL address   Value: $3D_{Hex}$

IC → TNILL



The operand is retrieved, and the result true (= 1) is delivered if the nilbit is set; otherwise false (= 0).

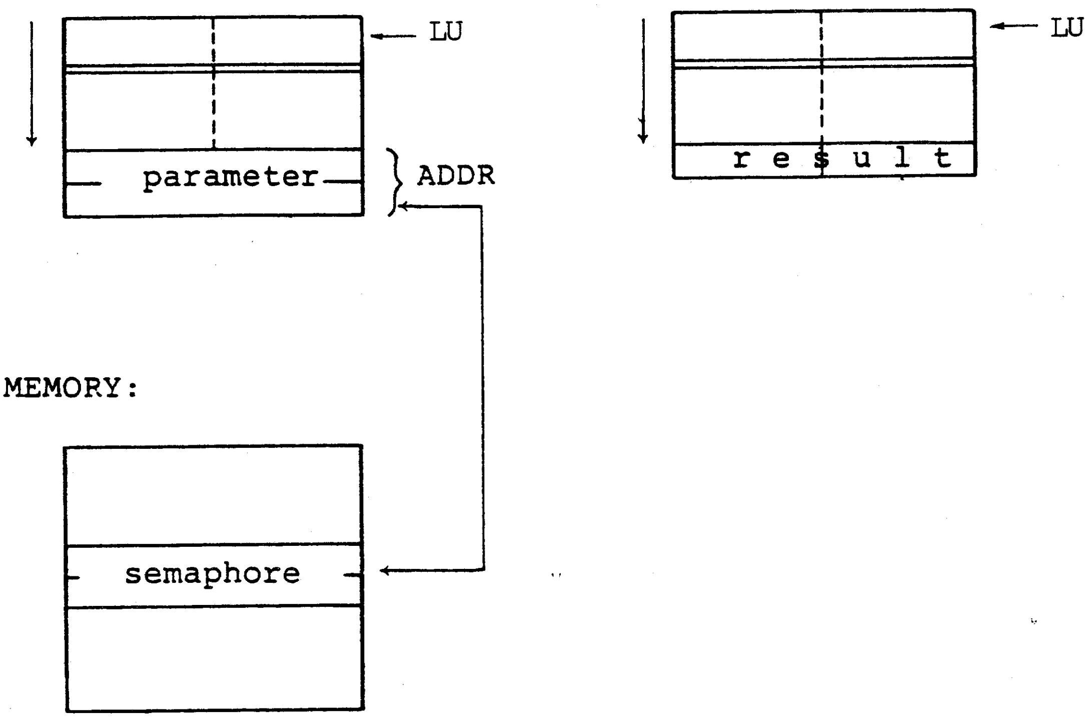
```
VAR
  adress, operand : adr;
  result : word;
BEGIN
  adress.disp:=pop;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  operand.base:= basetype(membyte(adress));
  result:=operand.base.nilbit;
  pus:=result;
END;
```

9.1.7 TOPEN

Test OPEN Semaphore Value: $3A_{Hex}$

IC → TOPEN

STACK BEFORE: STACK AFTER:



The operand must start on a word boundary.

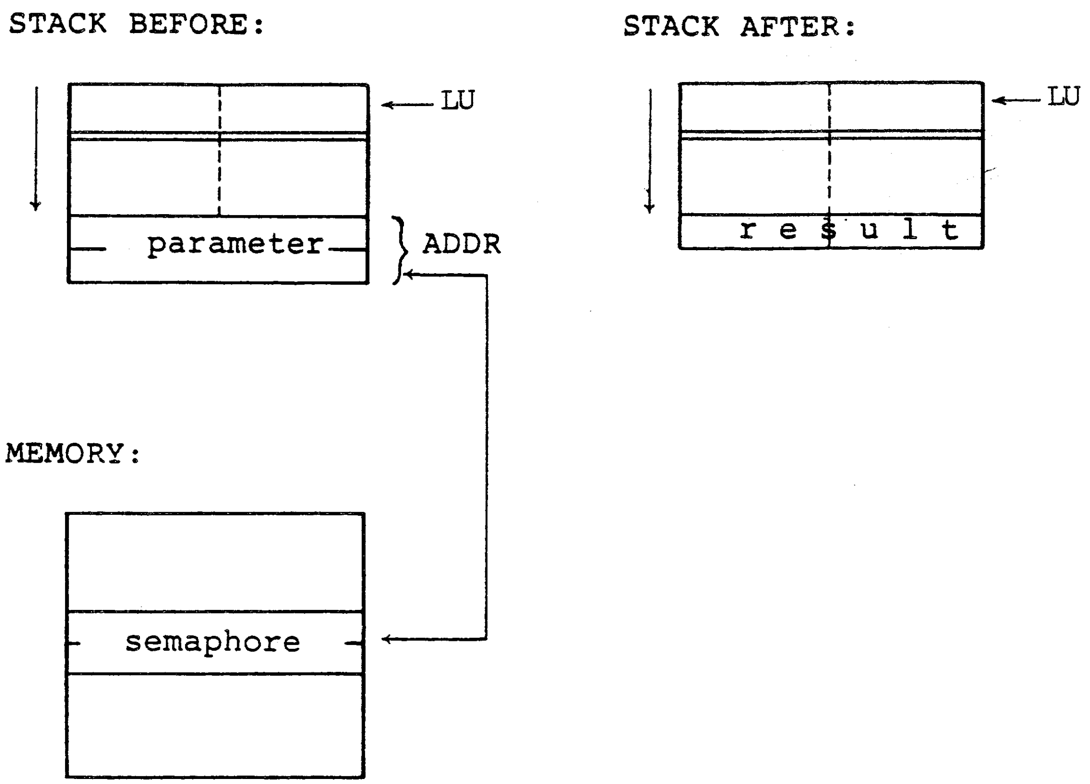
The operand, which is interpreted as a semaphore, is retrieved, and the result true (= 1) is delivered if the semaphore is open; otherwise false (= 0).

```
VAR
  adress : adr;
  semword : integer;
  semadr : adr;
BEGIN
  adress.disp:=pop;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  semword:=mem(adress);
  semadr.base:= basetype(semword);
  IF semadr.base.nilbit=1 THEN result:= false
  ELSE result:= (semword >= 0);
  pus:=result;
END;
```

9.1.8 TLOCK 9.1.8

Test LOCKed Semaphore Value: $3B_{Hex}$

IC → $\boxed{\text{TLOCK}}$



The operand must start on a word boundary.

The operand, which is interpreted as a semaphore, is retrieved, and the result true (= 1) is delivered if the semaphore is locked; otherwise false (= 0).

```
VAR
  adress : adr;
  semword : integer;
  semadr : adr;
BEGIN
  adress.disp:=pop;
  adress.base:= basetype(pop);
  IF adress.base.nilbit=1 THEN exception(nilpointer);
  semword:=mem(adress);
  semadr.base:= basetype(semword);
  IF semadr.base.nilbit=1 THEN result:= false
  ELSE result:= (semword < 0);
  pus:=result;
END;
```

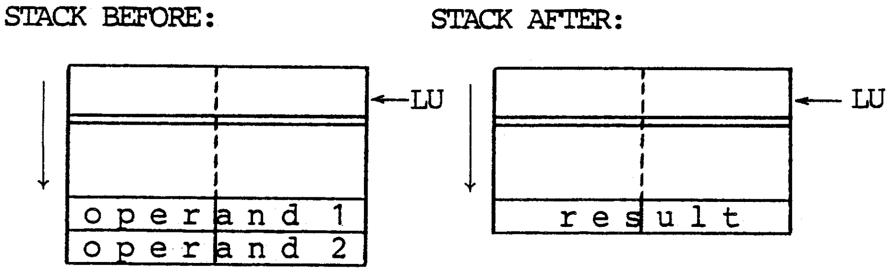
9.2 Dyadic Operators 9.2

Operands 1 and 2 are retrieved, and a result is produced from these operands in accordance with the operator.

9.2.1 ADD 9.2.1

ADD Value: 44_{Hex}

IC → ADD



operator: add

Operands 1 and 2 are interpreted as signed integers.

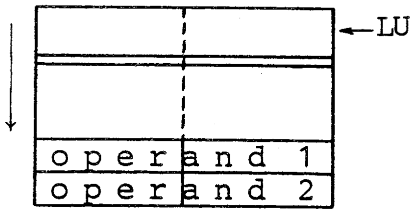
```
VAR
  operand1, operand2, result : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 + operand2;
  IF overflow THEN exception(overflow);
  pus:= result;
END;
```

9.2.2 SUB

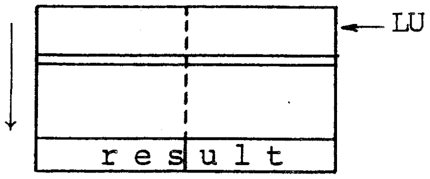
SUBtract Value: 45_{Hex}

IC → $\boxed{\text{SUB}}$

STACK BEFORE:



STACK AFTER:



operator: subtract

Operands 1 and 2 are interpreted as signed integers.

Operand 2 is subtracted from operand 1.

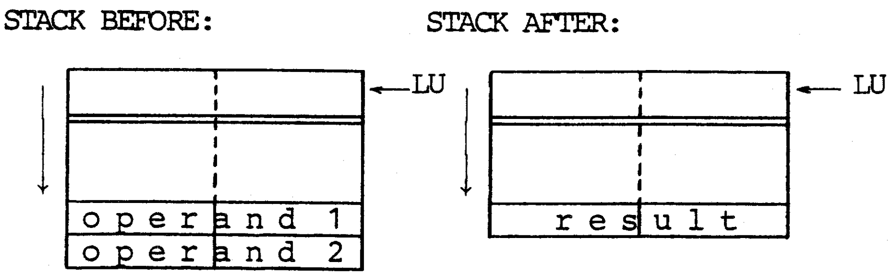
```
VAR
  operand1, operand2, result : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 - operand2;
  IF overflow THEN exception(overflow);
  pus:= result;
END;
```

9.2.3 MUL

MULtiply

Value: 49_{Hex}

IC → MUL



operator: multiply

Operands 1 and 2 are interpreted as signed integers.

The result contains the 16 least significant bits of the product.

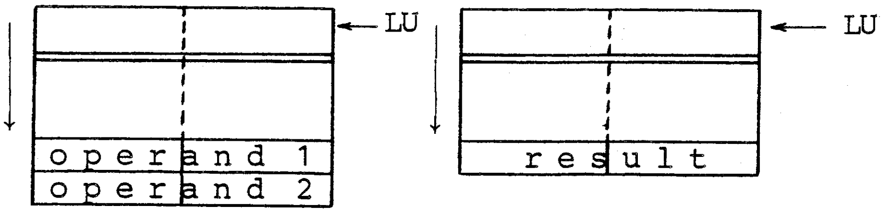
```
VAR
  operand1, operand2, result : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 * operand2;
  IF overflow THEN exception(overflow);
  pus:= result;
END;
```

9.2.4 DIV

DIVide   Value: $4A_{Hex}$

IC → DIV

STACK BEFORE: STACK AFTER:



operator: divide

Operands 1 and 2 are interpreted as signed integers.

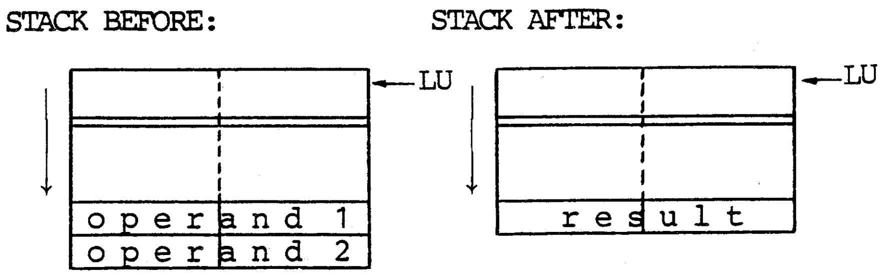
Operand 1 is the dividend and operand 2 the divisor.

```
VAR
  operand1, operand2, result : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  IF operand2 = 0 THEN exception(overflow);
  result:= operand1 DIV operand2;
  IF overflow THEN exception(overflow);
  pus:= result;
END;
```

9.2.5 MOD

MODulus Value: $4B_{Hex}$

IC → MOD



operator: modulus

Operands 1 and 2 are interpreted as signed integers.

The result is the remainder from the operator divide (see above).

The result has the sign of the first operand.

```
VAR
  operand1, operand2, result : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  IF operand2 = 0 THEN exception(overflow);
  result:= operand1 - ((operand1 div operand2) * operand2);
  IF overflow THEN exception(overflow);
  pus:= result;
END;
```

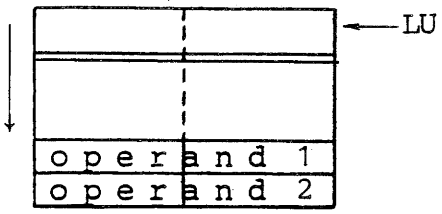
9.2.6 UADD

Unsigned ADD

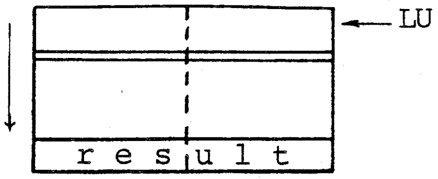
Value: 42_{Hex}

IC → UADD

STACK BEFORE:



STACK AFTER:



operator: unsigned add

Operands 1 and 2 are interpreted as unsigned integers (i.e. numbers in the range 0..65535).

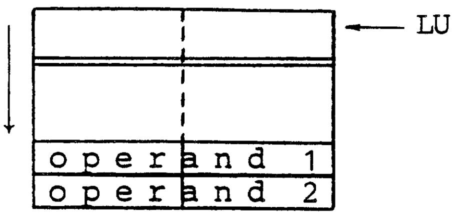
```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 + operand2;
  IF carry THEN exception(overflow);
  pus:= result;
END;
```

9.2.7 USUB

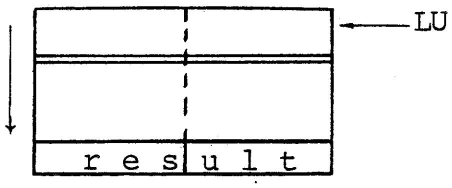
Unsigned SUBtract Value: 43_{Hex}

IC → USUB

STACK BEFORE:



STACK AFTER:



operator: unsigned subtract

Operands 1 and 2 are interpreted as unsigned integers (i.e. numbers in the range 0..65535).

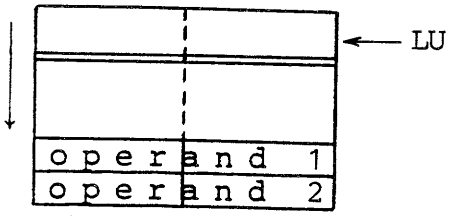
```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 - operand2;
  IF NOT carry THEN exception(overflow);
  pus:= result;
END;
```

9.2.8 UMUL

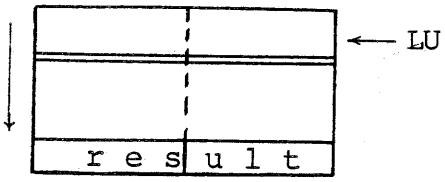
Unsigned MULtiply Value: 46_{Hex}

IC → UMUL

STACK BEFORE:



STACK AFTER:



operator: unsigned multiply

Operands 1 and 2 are interpreted as unsigned integers (i.e. numbers in the range 0..65535).

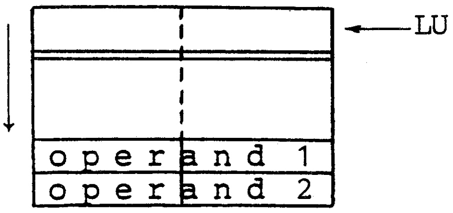
```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 * operand2;
  IF carry THEN exception(overflow);
  pus:= result;
END;
```

9.2.9 UDIV

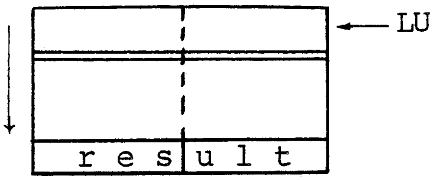
Unsigned DIVide   Value: 47_{Hex}

IC → $\boxed{\text{UDIV}}$

STACK BEFORE:



STACK AFTER:



operator: unsigned divide

Operands 1 and 2 are interpreted as unsigned integers (i.e. numbers in the range 0..65535).

```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  IF operand2 = 0 THEN exception(overflow);
  result:= operand1 DIV operand2;
  pus:= result;
END;
```

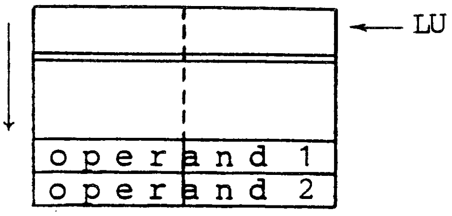
9.2.10 UMOD

Unsigned MODulus

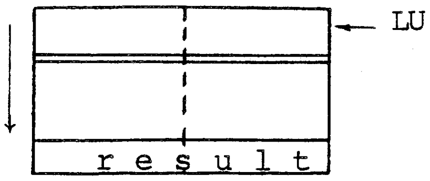
Value: 48_{Hex}

IC → $\boxed{\text{UMOD}}$

STACK BEFORE:



STACK AFTER:



operator: unsigned modulus

Operands 1 and 2 are interpreted as unsigned integers (i.e. numbers in the range 0..65535).

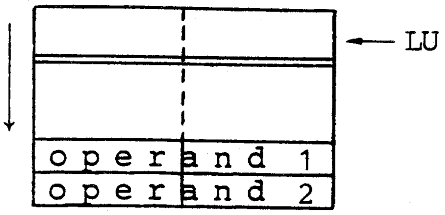
```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  IF operand2 = 0 THEN exception(overflow);
  result:= operand1 - (operand1 div operand2) * operand2;
  pus:= result;
END;
```

9.2.11 MADD

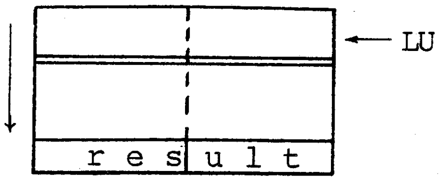
Modulo ADD   Value: $3F_{Hex}$

IC → MADD

STACK BEFORE:



STACK AFTER:



operator: addition modulo 64K

The operands are interpreted as unsigned integers (i.e. numbers in the range 0..65535) and the result is modulo 64K, i.e. overflow will not occur.

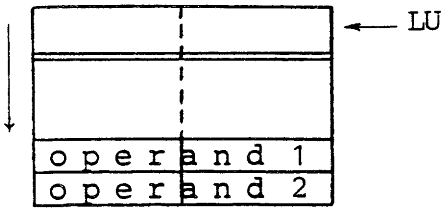
```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 + operand2;
  ps.carry:= ord(carry);
  pus:= result;
END;
```

9.2.12 MSUB

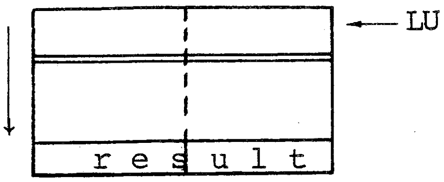
Modulo SUBtract   Value: 41_{Hex}

IC → MSUB

STACK BEFORE:



STACK AFTER:



operator: subtraction modulo 64K

The operands are interpreted as unsigned integers (i.e. numbers in the range 0..65535) and the result is modulo 64K, i.e. overflow will not occur.

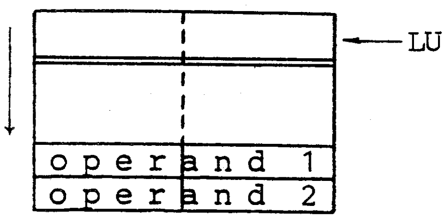
```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 - operand2;
  ps.carry:= not ord(carry);
  pus:= result;
END;
```

9.2.13 MMUL

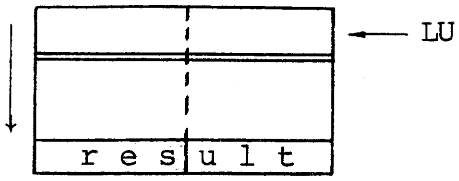
Modulo MULtiply Value: $3E_{Hex}$

IC → MMUL

STACK BEFORE:



STACK AFTER:



operator: subtraction modulo 64K

The operands are interpreted as unsigned integers (i.e. numbers in the range 0..65535) and the result is modulo 64K, i.e. overflow will not occur.

```
VAR
  operand1, operand2, result : word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  result:= operand1 * operand2;
  ps.carry:= ord(carry);
  pus:= result;
END;
```

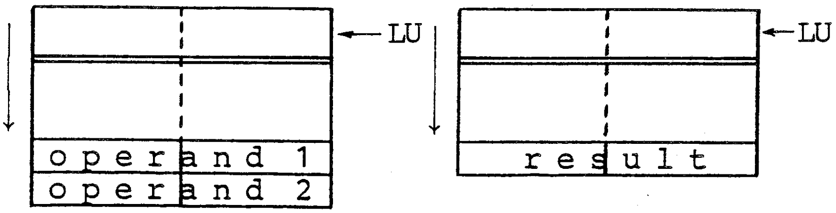
9.2.14 EQ

EQual

Value: 32_{Hex}

IC → EQ

STACK BEFORE: STACK AFTER:



operator: operand 1 = operand 2

Operands 1 and 2 are compared according to the relation EQual. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

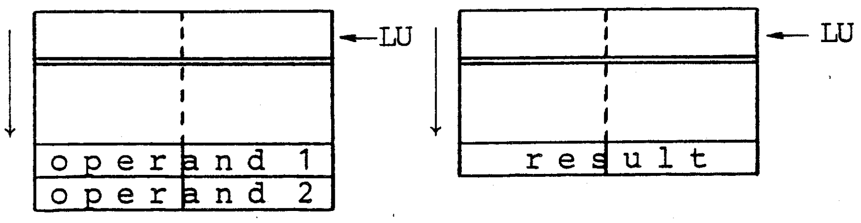
```
VAR
  operand1, operand2: integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 = operand2);
END;
```

9.2.15 NE

Not Equal Value: 33_{Hex}

IC → NE

STACK BEFORE: STACK AFTER:



operator: operand 1 <> operand 2

Operands 1 and 2 are compared according to the relation Not Equal. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

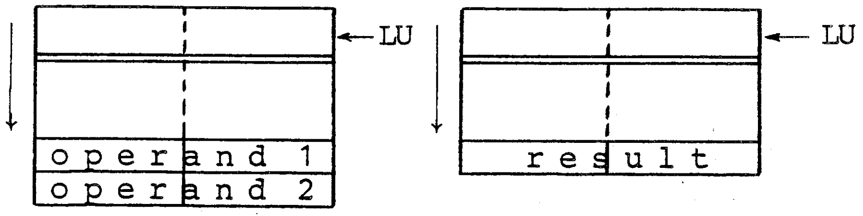
```
VAR
  operand1, operand2: integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 <> operand2); (* signed *)
END;
```

9.2.16 LT

Less Than   Value: 34_{Hex}

IC → LT

STACK BEFORE: STACK AFTER:



operator: operand 1 < operand 2

Operands 1 and 2 are compared according to the relation Less Than. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

Operands 1 and 2 are interpreted as signed integers.

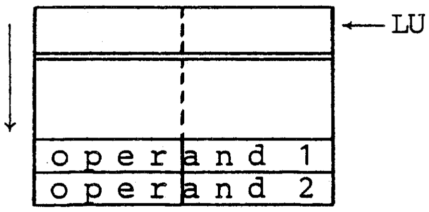
```
VAR
  operand1, operand2: integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 < operand2); (* signed *)
END;
```

9.2.17 GT

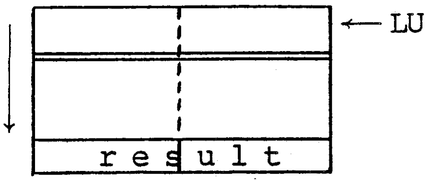
Greater Than Value: 35_{Hex}

IC → GT

STACK BEFORE:



STACK AFTER:



operator: operand 1 > operand 2

Operands 1 and 2 are compared according to the relation Greater Than. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

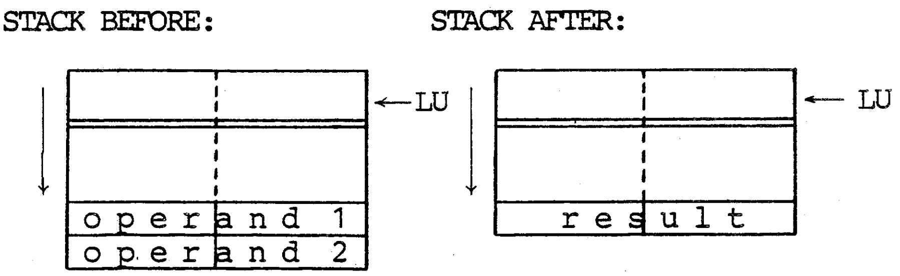
Operands 1 and 2 are interpreted as signed integers.

```
VAR
  operand1, operand2: integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 > operand2);
END;
```

9.2.18 LE

Less Than or Equal   Value: 36_{Hex}

IC → $\boxed{\text{LE}}$



operator: operand 1 <= operand 2

Operands 1 and 2 are compared according to the relation Less Than or Equal. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

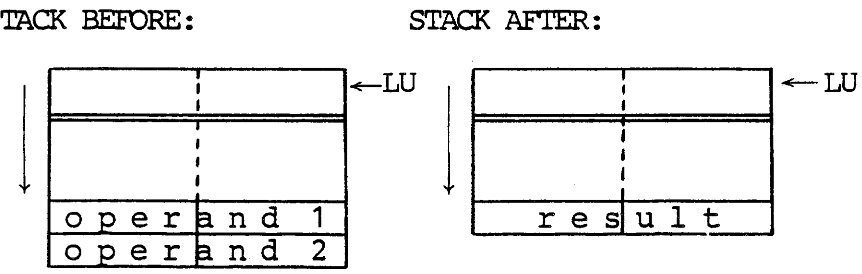
Operands 1 and 2 are interpreted as signed integers.

```
VAR
  operand1, operand2: integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 <= operand2); (* signed *)
END;
```

9.2.19 GE

Greater Than or Equal   Value: 37_{Hex}

IC → GE



operator: operand 1 >= operand 2

Operands 1 and 2 are compared according to the relation Greater Than or Equal. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

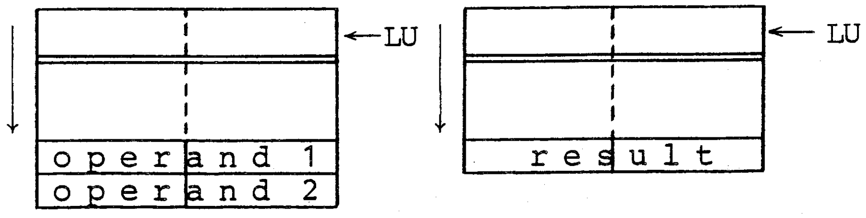
Operands 1 and 2 are interpreted as signed integers.

```
VAR
  operand1, operand2: integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 >= operand2); (* signed *)
END;
```

9.2.20 ULT

Unsigned Less Than   Value: 31_{Hex}

IC → ULT



STACK BEFORE:

STACK AFTER:

operator: operand 1 < operand 2

Operands 1 and 2 are compared according to the relation Less THAN. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

Operand 1 and 2 are interpreted as unsigned integers (i.e. numbers in the range 0..65535).

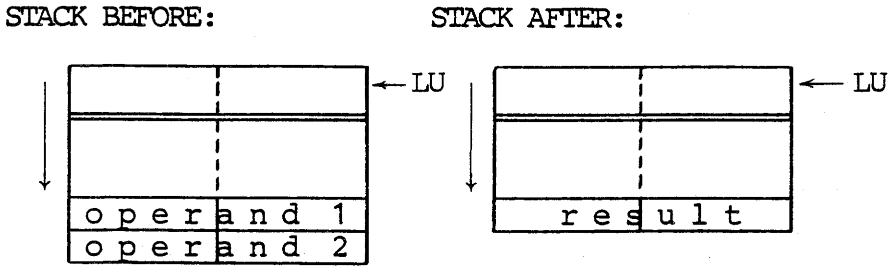
```
VAR
  operand1, operand2: word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  pus:= word(operand1 < operand2); (* unsigned *)
END;
```

9.2.21 AND

AND

Value: $4C_{Hex}$

IC → AND


STACK BEFORE:

STACK AFTER:

operator: AND

Operands 1 and 2 are interpreted as ordered sets of 16 logical values, true (=1) or false (= 0).

The logical operator AND operates bit for bit on the operands to produce the 16 bits of the result.

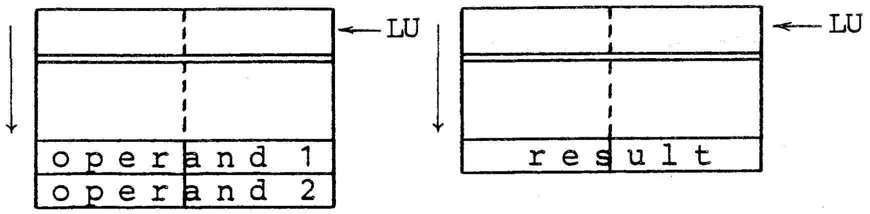
```
VAR
  operand1, operand2, result : word;
  i : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  FOR i:=0 TO 15 DO WITH operand1, operand2, result AS bitword DO
    CASE operand1(i) + operand2(i) OF
      0: result(i):=0;
      1: result(i):=0;
      2: result(i):=1;
    END;
  pus:= result;
END;
```

9.2.22 OR

OR   Value: $4D_{Hex}$

IC → $\boxed{\text{OR}}$

STACK BEFORE: STACK AFTER:



operator: OR

Operands 1 and 2 are interpreted as ordered sets of 16 logical values, true (=1) or false (= 0).

The logical operator OR operates bit for bit on the operands to produce the 16 bits of the result.

```
VAR
  operand1, operand2, result : word;
  i : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  FOR i:=0 TO 15 DO WITH operand1, operand2, result AS bitword DO
    CASE operand1(i) + operand2(i) OF
      0: result(i):=0;
      1: result(i):=1;
      2: result(i):=1;
    END;
  pus:= result;
END;
```

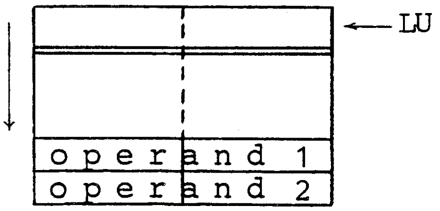
9.2.23 XOR

EXclusive OR

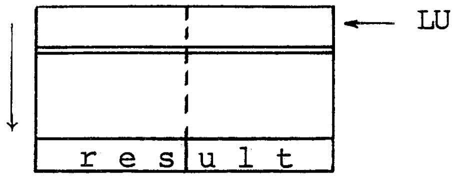
Value: $4E_{Hex}$

IC → XOR

STACK BEFORE:



STACK AFTER:



result := (operand1   operand2)   (operand1   operand2)

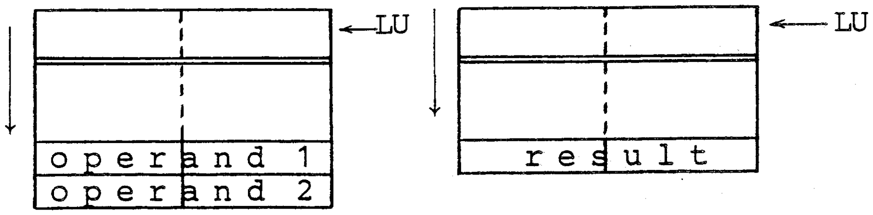
```
VAR
  operand1, operand2, result : word;
  i : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  FOR i:=0 TO 15 DO WITH operand1, operand2, result AS bitword DO
    CASE operand1(i) + operand2(i) OF
      0: result(i):=0;
      1: result(i):=1;
      2: result(i):=0;
    END;
  pus:= result;
END;
```

9.2.24 SHC 9.2.24

SHift Cyclic Value: 53_{Hex}

IC → SHC

STACK BEFORE: STACK AFTER:



operator: cyclic shift

Operands 1 is interpreted as an ordered set of 16 logical values, true (=1) or false (= 0).

Operand 2 is interpreted as a signed integer. The 16 bits of operand 1 are shifted cyclically the number of positions specified by operand 2. If operand 2 is positive, the shift is to the left; otherwise the shift is to the right.

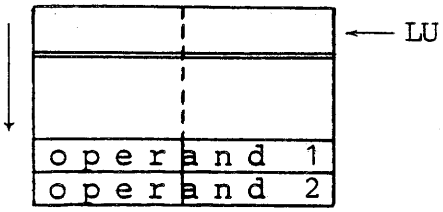
```
VAR
  operand1, result : word;
  operand2 : integer;
  i : integer;
  j : integer;
  b : bit;
BEGIN
  operand2:= pop;
  operand1:= pop;
  operand2:= operand2 mod 16;
  IF operand2 < 0 THEN
    FOR i:=-1 DOWNTO operand2 DO
    WITH operand1 AS bitword DO
    BEGIN
      b:= operand1(15);
      FOR j:= 15 DOWNTO 1 DO
        operand1(j):= operand1(j-1);
      operand1(0):=b;
    END
    ELSE
    FOR i:= 1 TO operand2 DO
    BEGIN
    WITH operand1 AS bitword DO
      b:=operand1(0);
      FOR j:=1 TO 15 DO
        operand1(j-1):=operand1(j);
      operand1(15):=b;
    END;
  pus:= result;
END;
```

9.2.25 CRC16

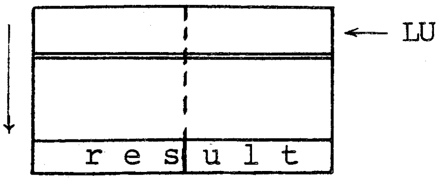
Cyclic Redundancy Check   Value: $4F_{Hex}$

IC → $\boxed{\text{CRC16}}$

STACK BEFORE:



STACK AFTER:



operator: crc16

- Operand1 represents the polynomium

$$f\ (x) = a_{15}\ x^{15} + a_{14}\ x^{14} + \ldots + a_1\ x + a_0$$

where a_j = operand1.bit$_j$.

Note that bit$_0$ is the most significant bit.

- Operand2 represents the polynomium

$$g\ (x) = x^{16} + b_{15}\ x^{15} + b_{a4}\ x^{14} + \ldots + b_1\ x + b_0$$

where b_j = operand2.bit$_j$.

Note that x^{16} by convention is implicitly given.

The instruction delivers the remainder by the division

$$(f\ (x)\ *\ x^{8})/g\ (x)$$

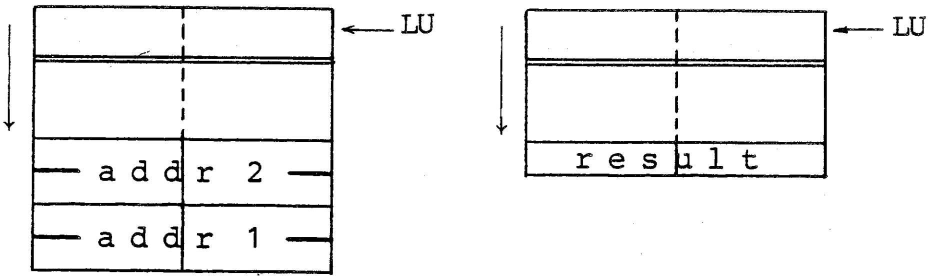
```
VAR
  operand1, operand2, result : integer;
  i : integer;
BEGIN
  operand2:= pop;
  operand1:= pop;
  FOR i:= 1 TO 8 DO
    IF ( operand1 and 1 ) = 1
      THEN operand1:= (operand1 SHIFT (-1)) XOR operand2
      ELSE operand1 := operand1 SHIFT (-1);
  result:= operand1;
  pus:= result;
END;
```

9.2.26 TEQAD

Test EQual ADdresses   Value: $3C_{Hex}$

IC → TEQAD

STACK BEFORE: STACK AFTER:



```
VAR
  adress1, adress2 : adr;
BEGIN
  adress1.disp:= pop;
  adress1.base:= basetype(pop);
  adress2.disp:= pop;
  adress2.base:= basetype(pop);
  IF ((adress1.base.nilbit=1) AND (adress2.base.nilbit=1) OR
     ((adress1.base = adress2.base) AND (adress1.disp = adress2.disp))
  THEN pus:= true
  ELSE pus:= false
END;
```

9.3 Operations on Sets

The representation of a set occupies a number of words. These words are regarded as a consecutive array of bits numbered from 0 on, the most significant being the rightmost bit in the last used byte in the set representation. The set (.operand 1 .. operand 2.) is represented by setting all bits from operand 1 to operand 2 to one in the consecutive array of bits.

Note that the sets (.a..b.) and (.0..b.) occupy the same space ("a" and "b" are greater than zero).

The size (in bytes) of the resulting set is defined in the following word, when the set is pushed upon the evaluation stack, but omitted in the representation of a setvariable.

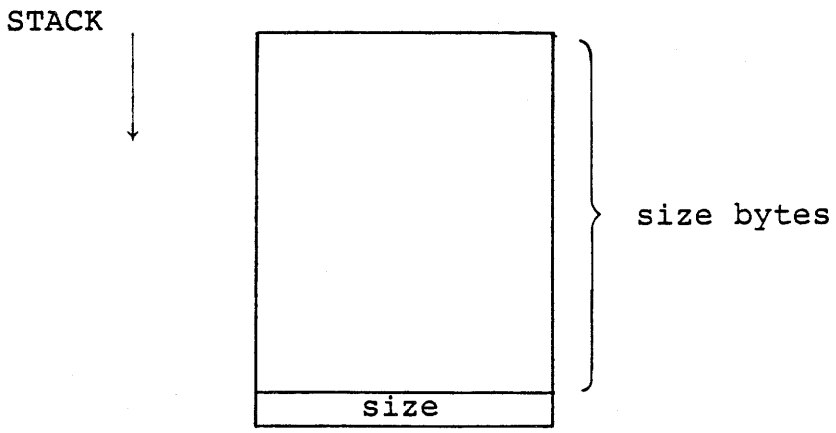


Figure 30: Evaluation Stack Representation of the Set (.a..b.), 0 <= a <= b, size = (1 + b DIV 16) * 2.

Set operations may be interrupted during execution, and then resumed after interrupt handling at a higher level. Set operations may therefore be activated in two modes, namely, the normal mode and the resume mode. The mode is detected by means of the resume bit in the ib register of the register set. Execution in the normal mode starts with the setting of the resume bit. The set instructions either dump the register stack or presume that it is empty, since the register stack is used for temporary variables.

During the execution of a set operation, the interrupt situation is tested within the looping parts of the operation. If an interrupt with higher priority occurs, the contents of the necessary working registers are dumped on the top of the stack and the instruction is terminated without updating the instruction pointer. When the operation is resumed, the register contents are reestablished and the operation continues. In the algorithms below, this dump-exit-resume sequence is indicated by means of the procedure call setresume.

9.3.1 Construct a Set from a Subrange 9.3.1

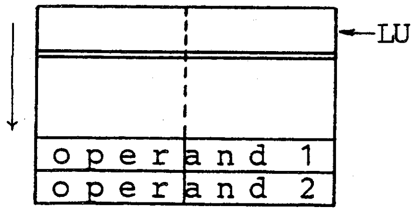
Operands 1 and 2 are retrieved, and a set is constructed and pushed on the stack as the result. The set is initialized to contain the integers in the subrange (operand 1 .. operand 2). Operand 1 must be greater than or equal to zero. If operand 2 is less than operand 1, the set will be empty.

9.3.1.1 SETCR

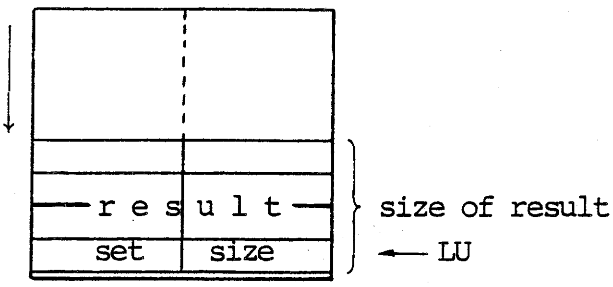
SET CReate   Value: 56_{Hex}

IC → SETCR

STACK BEFORE:



STACK AFTER:



size of result (including the size word): if operand2 >= operand1 then:

(1 + operand2 DIV 16) * 2 + 2

otherwise: 2

Operands 1 and 2 are interpreted as the ordinal numbers of the first and last elements to be included in the set. The smallest ordinal number of an element in a set is 0.

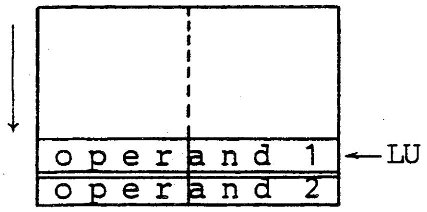
```
VAR
  operand1, operand2: integer;
  size: integer;
  oldlu: word;
  zerobytes,onebytes: integer;
  shiftword, lastword: word;
BEGIN
  operand2:= pop;
  operand1:= pop;
  IF operand1 < 0 THEN exception(setcrexception);
  IF operand1 > operand2
  THEN size:= 0
  ELSE size:= (1 + operand2 DIV 16) * 2;
  oldlu:= lu;
  checkdumstack(size + 2);
  stack(lu-1):= size;
  IF size > 0
  THEN BEGIN
    zerobytes:= ( operand1 DIV 16) * 2;
    onebytes := size - zerobytes;
    shiftword:= #hffff shift (-operand1 MOD 16);
    lastword:= NOT (#hffff shift (-operand2 MOD 16));
    FOR i:= 1 TO zerobytes DIV 2 DO
    BEGIN
      stack(oldlu-1+2*i):=0;
      IF interrupt THEN setresume;
    END;
    IF onebytes=0
    THEN stack(oldlu+zerobytes+1):=shiftword AND lastword
    ELSE BEGIN
      stack(oldlu + zerobytes + 1):= shiftword;
      FOR i:= 1 TO (onebytes DIV 2)-2 DO
      BEGIN
        stack(oldlu+zerobytes+1+2*i):=#hffff;
        IF interrupt THEN setresume;
      END;
      stack(oldlu+zerobytes+onebytes-1):= lastword;
    END;
  END;
END;
```

9.3.1.2 SETAD

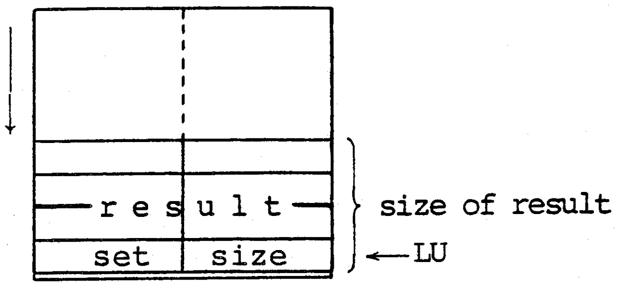
SET ADjust Value: $5F_{Hex}$

IC → $\boxed{\text{SETAD}}$

STACK BEFORE:



STACK AFTER:



size of result: operand2+2

Operand1 is interpreted as a set, which is truncated or enlarged to a new set of size operand2 bytes. If the set is truncated, it is tested that only words containing 0-bits are truncated.

```
VAR
  size1, size2, i: integer;
  oldlu, bitword: word;
BEGIN
  size2:= pop;
  IF odd(size2) THEN exception(oddoperand);
  size1:= stack(lu-1);
  IF size1 < size2
  THEN BEGIN
    oldlu:= lu;
    checkdumpstack(size2-size1);
    FOR i:=1 TO (size2-size1) DIV 2 DO
    BEGIN
      stack(oldlu-1 + (i-1)*2) := 0;
      IF interrupt THEN setresume;
    END;
  END
  ELSE BEGIN
    FOR i:=1 TO (size1-size2) DIV 2 DO
    BEGIN
      bitword:= stack(lu-1 - 2*i);
      IF bitword<>0 THEN exception(truncationexception);
      IF interrupt THEN setresume;
      lu:= lu - (size1-size2);
    END;
  END;
  stack(lu-1):= size2;
END;
```

9.3.2 Operations on Sets Giving a Set as the Result 9.3.2

Operands 1 and 2, both of which are sets, are retrieved, and a result, which is also a set, is produced from these operands in accordance with the operator.

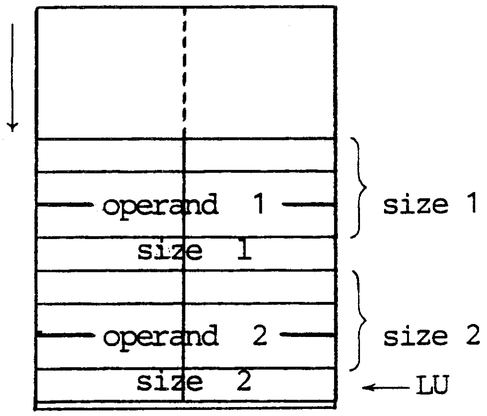
The size (in bytes) of operands 1 and 2 and the result is defined in the following word of each.

9.3.2.1 SETUN 9.3.2.1

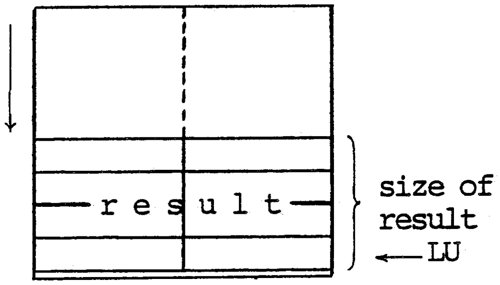
SET UNion Value: $5C_{Hex}$

IC → SETUN

STACK BEFORE:



STACK AFTER:



operator: union

size of result: max (size1, size2) + 2

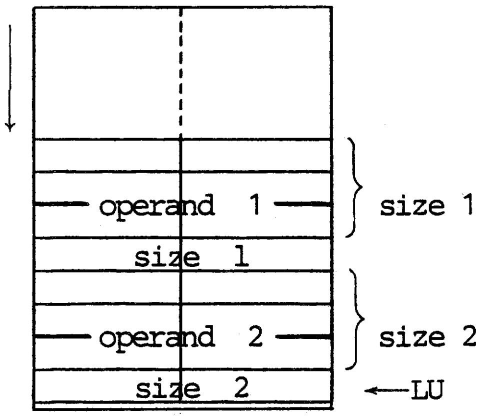
```
VAR
  size1, size2: integer;
  start1, start2: word;
  minlength, restlength: integer;
BEGIN
  size2:= stack(lu-1);
  size1:= stack(lu-size2-3);
  start2:= lu-size2-1;
  start1:= start2-size1-2;
  restlength:= size1 - size2;
  IF restlength < 0
  THEN minlength:= size1
  ELSE minlength:= size2;
  FOR i:=0 TO ( minlength DIV 2 ) - 1 DO
  BEGIN
    stack(start1+2*i):= stack(start1+2*i) OR stack(start2+2*i);
    IF interrupt THEN setresume;
  END;
  IF restlength < 0
  THEN BEGIN
    FOR i:=0 TO ( - restlength DIV 2 ) - 1 DO
    BEGIN
      stack(start1 + minlength + 2*i):= stack(start2 + minlength + 2*i);
      IF interrupt THEN setresume;
    END;
    stack(start1 + minlength - restlength):= minlength - restlength;
  END;
  lu:= start1 + minlength + abs(restlength) + 1;
END;
```

9.3.2.2 SETIN

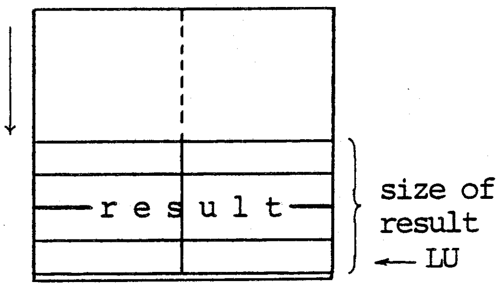
SET INtersection Value: $5D_{Hex}$

IC → $\boxed{\text{SETIN}}$

STACK BEFORE:



STACK AFTER:



operator: intersection

size of result: max (size1, size2) + 2

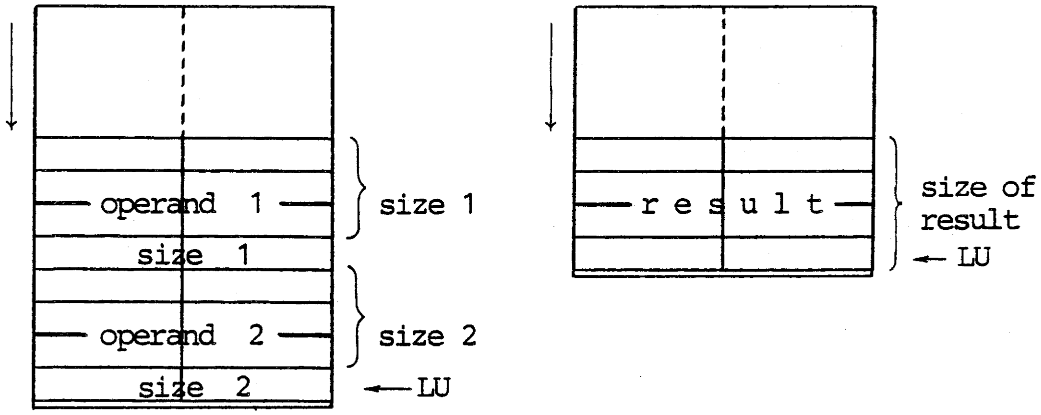
```
VAR
  size1, size2: integer;
  start1, start2: word;
  restlength, minlength: integer;
BEGIN
  size2:= stack(lu-1);
  size1:= stack(lu-size2-3);
  start2:= lu-size2-1;
  start1:= start2-size1-2;
  restlength:= size1 - size2;
  IF restlength < 0
  THEN minlength:= size1
  ELSE minlength:= size2;
  FOR i:=0 TO ( minlength DIV 2 ) - 1 DO
  BEGIN
    stack(start1+2*i):= stack(start1+2*i) AND stack(start2+2*i);
    IF interrupt THEN setresume;
  END;
  IF restlength > 0
  THEN BEGIN
    FOR i:=0 TO ( restlength DIV 2 ) - 1 DO
    BEGIN
      stack(start1 + minlength + 2*i):= 0;
      IF interrupt THEN setresume;
    END;
  END;
  IF restlength < 0
  THEN BEGIN
    FOR i:=0 TO ( - restlength DIV 2 ) - 1 DO
    BEGIN
      stack(start1 + minlength + 2*i):= 0;
      IF interrupt THEN setresume;
    END;
    stack(start1 + minlength - restlength):= minlength - restlength;
  END;
  lu:= start1 + minlength + abs(restlength) + 1;
END;
```

9.3.2.3 SETDI

SET DIfference Value: $5E_{Hex}$

IC → SETDI

STACK BEFORE: STACK AFTER:



operator: difference

size of result: max (size1, size2) + 2

```
VAR
  size1, size2: integer;
  start1, start2: word;
  minlength, restlength, i: integer;
BEGIN
  size2:= stack(lu-1);
  size1:= stack(lu-size2-3);
  start2:= lu-size2-1;
  start1:= start2-size1-2;
  restlength:= size1 - size2;
  IF restlength < 0
  THEN minlength:= size1
  ELSE minlength:= size2;
  FOR i:=0 TO ( minlength DIV 2 ) - 1 DO
  BEGIN
    stack(start1+2*i):= stack(start1+2*i) AND NOT ( stack(start2+2*i));
    IF interrupt THEN setresume;
  END;
  IF restlength < 0
  THEN BEGIN
    FOR i:=0 TO ( - restlength DIV 2 ) - 1 DO
    BEGIN
      stack(start1 + minlength + 2*i):= 0;
      IF interrupt THEN setresume;
    END;
    stack(start1 + minlength - restlength):= minlength - restlength;
  END;
  lu:= start1 + minlength + abs(restlength) + 1;
END;
```

9.3.3 Comparison of Sets 9.3.3

Operands 1 and 2, both of which are sets, are compared according to a relation. The result is true (= 1), if the relation holds; otherwise the result is false (= 0).

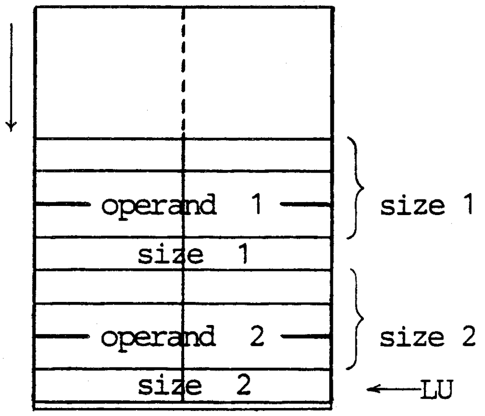
The size (in bytes) of operands 1 and 2 is defined in the following word of each.

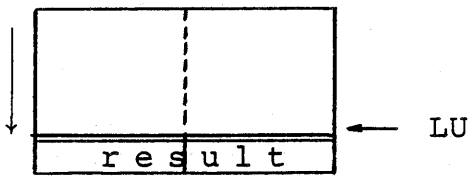
9.3.3.1 SETEQ 9.3.3.1

SET EQual Value: 59_{Hex}

IC → SETEQ

STACK BEFORE: STACK AFTER:





relation : equal

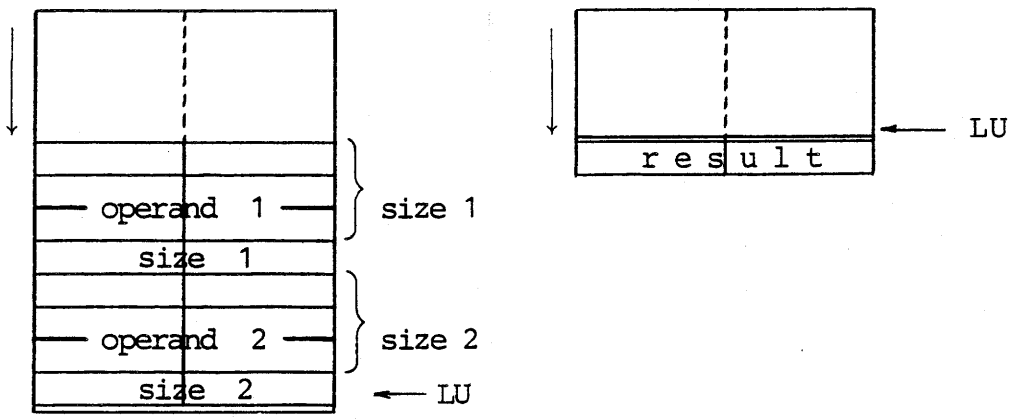
```
LABEL
  setfin;
VAR
  size1, size2: integer;
  start1, start2: word;
  minlength, restlength, i: integer;
BEGIN
  size2:= stack(lu-1);
  size1:= stack(lu-size2-3);
  start2:= lu-size2-1;
  start1:= start2-size1-2;
  restlength:= size1 - size2;
  IF restlength < 0
  THEN minlength:= size1
  ELSE minlength:= size2;
  lu:= start1-1;
  FOR i:=0 TO ( minlength DIV 2 ) - 1 DO
  BEGIN
    IF stack(start1+2*i) <> stack(start2+2*i)
    THEN BEGIN
      pus:= false;
      GOTO setfin;
    END;
    IF interrupt THEN setresume;
  END;
  IF restlength > 0
  THEN BEGIN
    FOR i:=0 TO ( restlength DIV 2 ) - 1 DO
    BEGIN
      IF stack(start1 + minlength + 2*i) <> 0
      THEN BEGIN
        pus:= false;
        GOTO setfin;
      END;
      IF interrupt THEN setresume;
    END;
  END;
  IF restlength < 0
  THEN BEGIN
    FOR i:=0 TO ( - restlength DIV 2 ) - 1 DO
    BEGIN
      IF stack(start2 + minlength + 2*i) <> 0
      THEN BEGIN
        pus:= false;
        GOTO setfin;
      END;
      IF interrupt THEN setresume;
    END;
  END;
  pus:= true;

setfin:
END;
```

9.3.3.2 SETSB

SET SuBset Value: $5A_{Hex}$

IC → SETSB

STACK BEFORE: STACK AFTER:



relation : subset

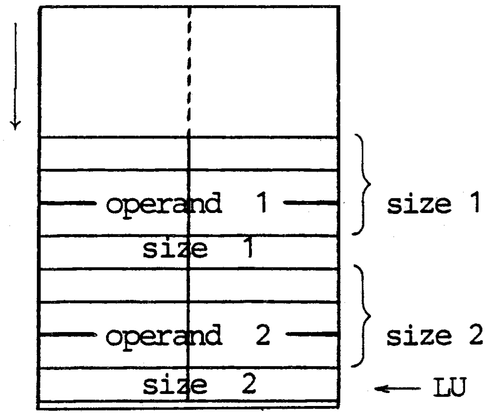
```
LABEL
  setfin;
VAR
  size1, size2: integer;
  start1, start2: word;
  minlength, restlength, i: integer;
BEGIN
  size2:= stack(lu-1);
  size1:= stack(lu-size2-3);
  start2:= lu-size2-1;
  start1:= start2-size1-2;
  restlength:= size1 - size2;
  IF restlength < 0
  THEN minlength:= size1
  ELSE minlength:= size2;
  lu:= start1-1;
  FOR i:=0 TO ( minlength DIV 2 ) - 1 DO
  BEGIN
    IF stack(start1+2*i) AND NOT (stack(start2+2*i)) <> 0
    THEN BEGIN
      pus:= false;
      GOTO setfin;
    END;
    IF interrupt THEN setresume;
  END;
  IF restlength > 0
  THEN BEGIN
    FOR i:=0 TO ( restlength DIV 2 ) - 1 DO
    BEGIN
      IF stack(start1 + minlength + 2*i) <> 0
      THEN BEGIN
        pus:= false;
        GOTO setfin;
      END;
      IF interrupt THEN setresume;
    END;
  END;
  pus:= true;

setfin:
END;
```

9.3.3.3 SETSP

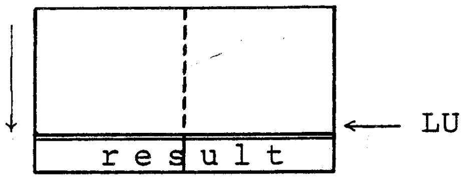
SET SuPerset Value: $5B_{Hex}$

IC → SETSP

STACK BEFORE:



STACK AFTER:



relation : superset

```
LABEL
  setfin;
VAR
  size1, size2: integer;
  start1, start2: word;
  minlength, restlength: integer;
BEGIN
  size2:= stack(lu-1);
  size1:= stack(lu-size2-3);
  start2:= lu-size2-1;
  start1:= start2-size1-2;
  restlength:= size1 - size2;
  IF restlength < 0
  THEN minlength:= size1
  ELSE minlength:= size2
  lu:= start1-1;
  FOR i:=0 TO ( minlength DIV 2 ) - 1 DO
  BEGIN
    IF NOT (stack(start1+2*i)) AND stack(start2+2*i) <> 0
    THEN BEGIN
      pus:= false;
      GOTO setfin;
    END;
    IF interrupt THEN setresume;
  END;
  IF restlength < 0
  THEN BEGIN
    FOR i:=0 TO ( - restlength DIV 2 ) - 1 DO
    BEGIN
      IF stack(start2 + minlength + 2*i) <> 0
      THEN BEGIN
        pus:= false;
        GOTO setfin;
      END;
      IF interrupt THEN setresume;
    END;
  END;
  pus:= true;

setfin:
END;
```

9.3.4 Test for Membership of a Set

Operands 1 is interpreted as a set. Operand2 is interpreted as the ordinal number (unsigned integer) of an element in the set operand 1. The result is true (= 1), if the operand 1 contains the element with the ordinal number operand 2; otherwise the result is false (= 0).

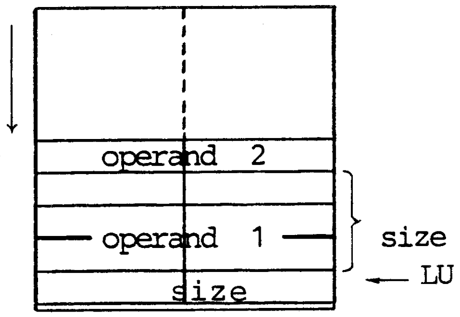
The size (in bytes) of the set is defined in the following word.

9.3.4.1 SETTM

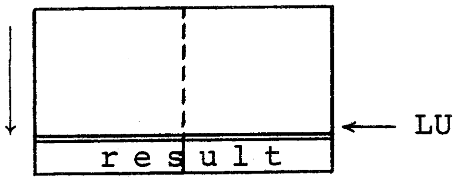
SET Test Membership Value: 57_{Hex}

IC → SETTM

STACK BEFORE:



STACK AFTER:



The smallest ordinal number of an element in a set is 0, and it is represented by the most significant bit of the first word (smallest address).

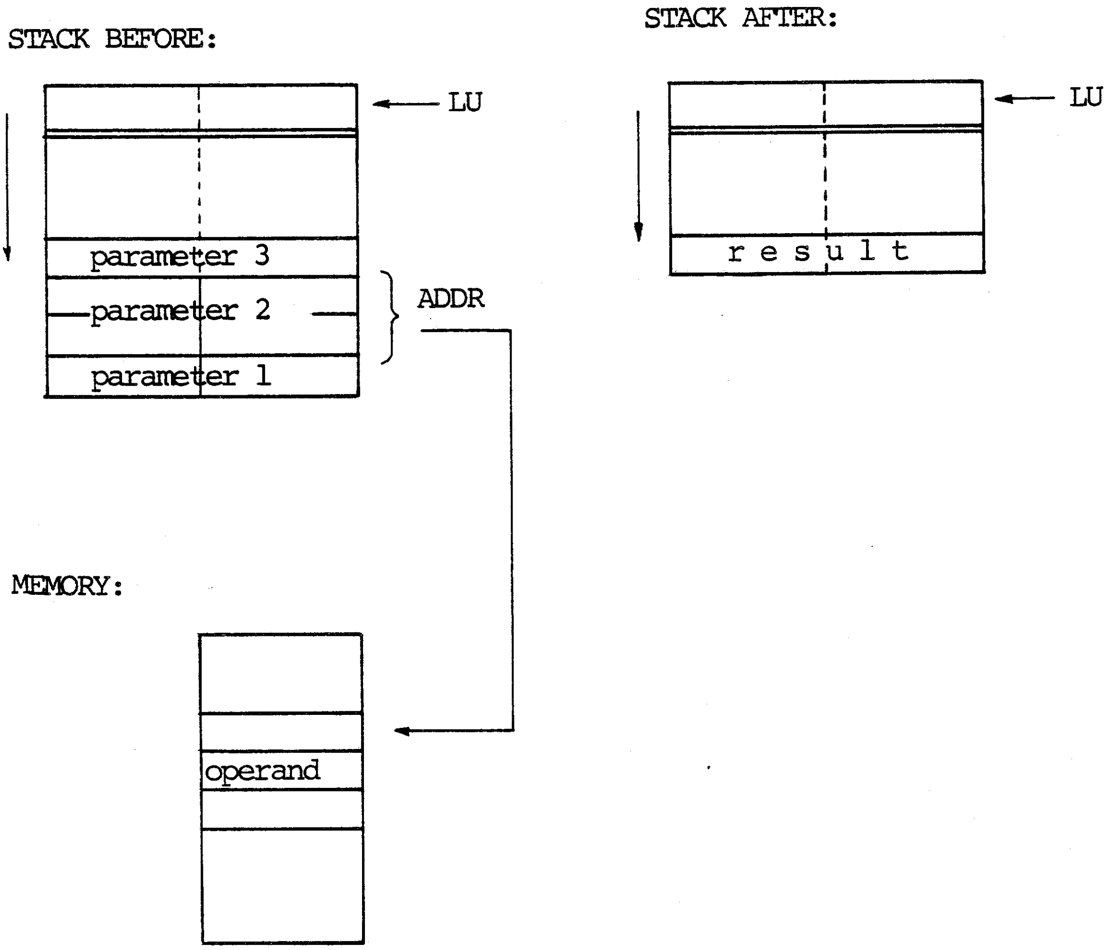
```
VAR
  size : integer;
  operand : integer;
  work : word;
BEGIN
  size:= stack(lu-1);
  operand:= stack(lu-1-size-2);
  IF size*8 <= operand
  THEN pus:= false
  ELSE BEGIN
    work:= stack(lu-1-size+(operand DIV 16)*2);
    pus:=  work shift ((operand AND #h0F)-15) AND 1;
  END;
  lu:= lu-size-2;
END;
```

9.3.4.2 SETATM

SET Address Test Membership

Value: 58_{Hex}

IC → SETATM



The smallest ordinal number of an element in a set is 0, and it is represented by the most significant bit of the first word (smallest address).

```
(* SETATM *)


VAR
  size : integer;
  setadr : adr;
  operand, work : word;
BEGIN
  size:= pop;
  setadr.disp:= pop;
  setadr.base:= basetype(pop);
  operand:= pop;
  IF setadr.base.nilbit=1 THEN exception(nilpointer);
  IF size*8 <= operand
  THEN pus:= false
  ELSE BEGIN
    work:= memword(adroffset(setadr, (operand DIV 16)*2));
    pus:= (work shift ((operand and h0F)-15)) AND 1;
  END;
END;
```

10. INPUT/OUTPUT

A device is regarded as a set of registers:

function
control
status in
status out
data in
data out
eoi
interrupt

A given device may have a subset of the above registers. The structure of the individual registers is device dependent.

Algorithmetic Descriptions

The following constants and types are used in conjunction with the device concept:

```
read_data     = #h000;
write_data    = #h040;
read_status   = #h080;
write_control = #h0C0;

function_type = read_data .. write_control;

devicetype = RECORD
               function : function_type;
               control : word;
               statusin : word;
               status out : word;
               data in : word;
               data out : word;
               eoi : bit; (* end of information *)
               interrupt : bit
             END;
deviceno = 0..127;
device : ARRAY(deviceno) OF devicetype;
```

The type message is defined in section 3.11.

A buffer area is described by a number of indices which satisfy the relation

$0 \le first \le last < top \le -1$ (65535) (* unsigned *)

The following routines are used in conjunction with the I/O instructions.

```
PROCEDURE xmitword(f: functiontype;
                   data: word;
                   dev: deviceno);
BEGIN
  device(dev).function:= f;
  CASE f OF
    write_date: device(dev).
                dataout:= data;
    read_status: device(dev).
                statusout:= data;
    write_control: device(dev).
                control:= data
  END
END;

FUNCTION waitinput(dev: deviceno): word;
BEGIN
                      waitinput:=device(dev).datain;
                    END;

                    FUNCTION waitstatus(dev: deviceno): word;
                    BEGIN
                      waitstatus:= device(dev).statusin;
                    END;

                    PROCEDURE updateeoi;
                    BEGIN
                      IF eoi THEN ps.eoi:= 1 ELSE ps.eoi:= 0;
                    END;

                    FUNCTION getaddrword(VAR startadr: adr): boolean;
                    VAR
                      count: word;
                    BEGIN
                      startadr.disp:=pop;
                      startadr.base:= basetype(pop);
                      count:= pop;
                      count:= count - 2;
                      pus:=count;
                      IF odd(count) THEN exception(oddoperand);
                      pus:= word(startadr.base);
                      pus:= startadr.disp + 2;
                      IF count = 0 THEN getadrword:= true
                                   ELSE getadrword:= false;
                    END;
```

```
FUNCTION getadrbyte(VAR startadr: adr): boolean;
VAR
  count: word;
BEGIN
  startadr.disp:=pop;
  startadr.base:= basetype(pop);
  count:= pop;
  count:= count - 1;
  pus:= count;
  pus:= word(startadr.base);
  pus:= startadr.disp + 1;
  IF count = 0 THEN getadrbyte:= true
               ELSE getadrbyte:= false;
END;

FUNCTION blockstart: boolean;
BEGIN
  blockstart:= true;
  IF ps.level0 THEN exception(level0io);
  IF ps.to=1 THEN blockstart:=false
  ELSE BEGIN
    ps.eoi:=0;
    device(ps.level).interrupt:=0;
  END;
END;

PROCEDURE blockend;
VAR
  count, next: word;
BEGIN
  pop; pop;  (* remove startaddr *)
  count:= pop;
  next:= pop - count;
  pus:= next;
END;

PROCEDURE clearcurrent;
BEGIN
  device(curlevel).interrupt:= 0;
END;
```

10.1 Write Control 10.1

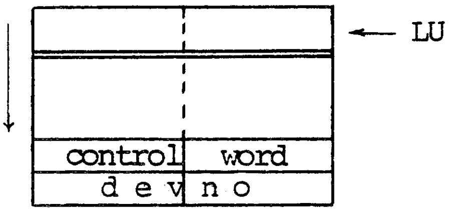
The parameter is interpreted as a device number. The value of the operand is transferred to the control register of the device.

10.1.1 IOWC 10.1.1

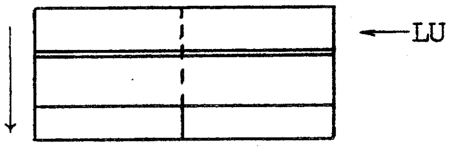
Input/Output Write Control Value: 21_{Hex}

IC → IOWC

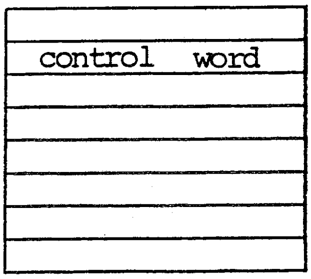
STACK BEFORE:



STACK AFTER:



device (devno):



```
VAR
  devno : integer;
BEGIN
  devno:= pop;
  xmitword(writecontrol,pop,devno);
END;
```

10.2 Write Word

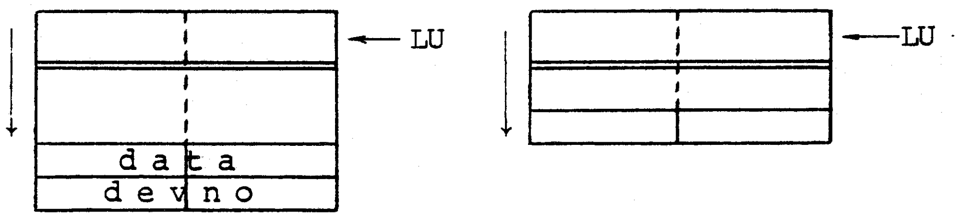
The parameter is interpreted as a device number. The value of the operand is transferred to the dataout register of the device. On end of information, the value of the eoi register is transferred to the eoi field of the ps register.

10.2.1 IOWW

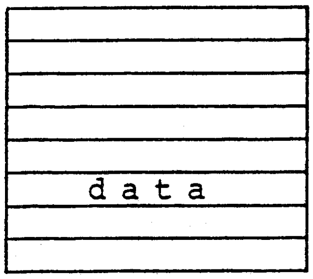
Input/Output Write Word Value: 24_{Hex}

IC → IOWW

STACK BEFORE: STACK AFTER:



device (devno):



```
VAR
  devno : integer
BEGIN
  devno:= pop;
  xmitword(writedata,pop,devno);
  IF eoi THEN ps.eoi:= 1;
END;
```

10.3 General Output 10.3

This instruction, which is used for device testing and maintenance, permits the execution of special, device-dependent functions not provided by the other I/O instructions.

Operand 1 is interpreted as a device number. The function defined by operand 2 is performed on the device. The word operand 3 is transferred, according to the function, to the dataout, statusout, or control register of the device.

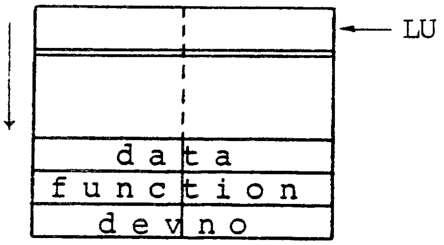
On end of information, the value of the eoi register is transferred to the eoi field of the ps register.

10.3.1 IOGO 10.3.1

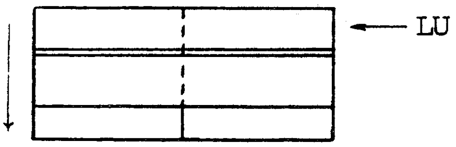
Input/Output General Output Value: 25_{Hex}

IC → $\boxed{\text{IOGO}}$

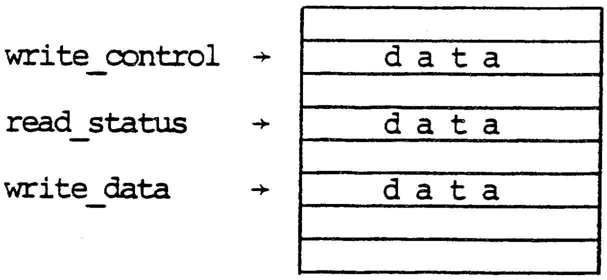
STACK BEFORE:



STACK AFTER:



device (devno):



```
VAR
  devno: integer;
  data, function: word;
BEGIN
  devno:=pop;
  function:=pop;
  data:=pop;
  xmitword(function,data,devno);
  IF eoi THEN ps.eoi:= 1;
END;
```

10.4 Read Status 10.4

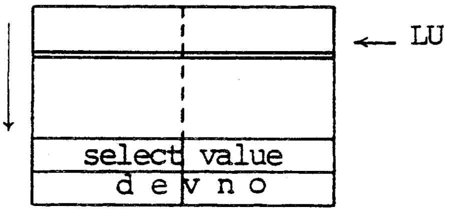
The first operand is interpreted as a device number. The result is the status information selected by the second operand from the device. First the second operand is transferred to the status out register of the device, and then the contents of the status in register are transferred as the result.

10.4.1 IORS 10.4.1

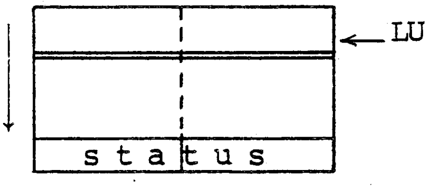
Input/Output Read Status Value: 22_{Hex}

IC → IORS

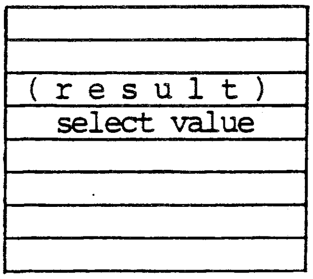
STACK BEFORE:



STACK AFTER:



device (devno):



```
VAR
  devno : integer;
BEGIN
  devno:= pop;
  xmitword(readstatus,pop,devno);
  pus:= waitstatus(devno);
END;
```

10.5 Read Word 10.5

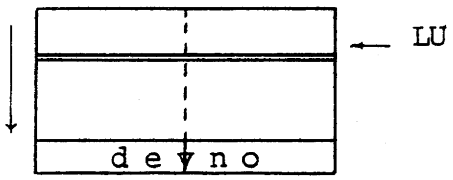
The operand is interpreted as a device number. The result is the contents of the datain register of the device. The value of the eoi register is transferred to the eoi field of the ps register.

10.5.1 IORW 10.5.1

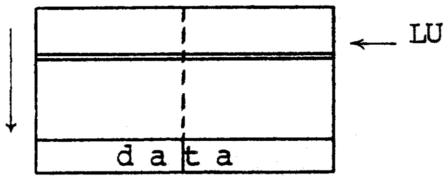
Input/Output Read Word Value: 23_{Hex}

IC → IORW

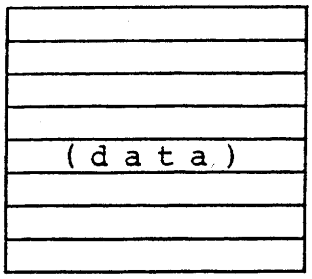
STACK BEFORE:



STACK AFTER:



device (devno):



```
VAR
  devno : integer;
BEGIN
  devno:= pop;
  xmitword(readdata,0,devno);
  pus:= waitinput(devno);
                    IF eoi THEN ps.eoi:= 1 ELSE ps.eoi:= 0;
                  END;
```

10.6 General Input

This instruction, which is used for device testing and maintenance, permits the execution of special, device-dependent functions not provided by the other I/O instructions.

Parameter 1 is interpreted as a device number. The function defined by parameter 2 is performed on the device. The word parameter 3 is tramsferred to the statusout register of the device, and the resulting word is obtained, according to the function, from the datain or statusin register.

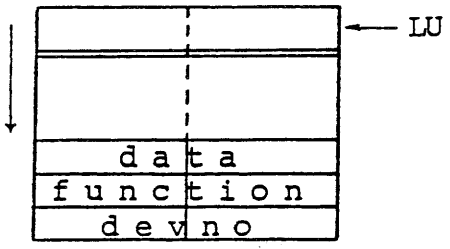
The value of the eoi register is transferred to the eoi field of the ps register.

10.6.1 IOGI

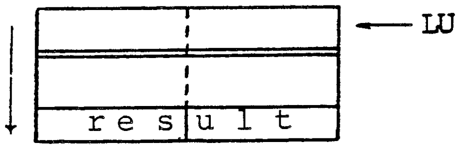
Input/Output General Input   Value: 26_{Hex}

IC → $\boxed{\text{IOGO}}$

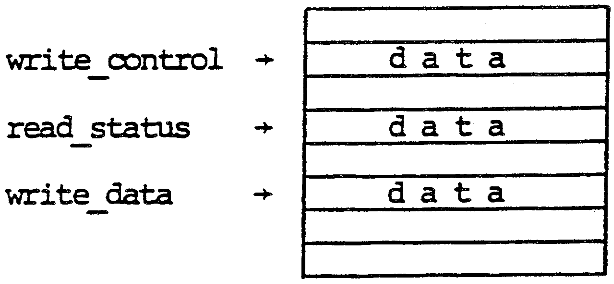
STACK BEFORE:



STACK AFTER:



device (devno):



```
VAR
  devno : integer;
  function : function_type;
  data : word;
BEGIN
  devno:= pop;
  function:= pop;
  data:= pop;
  xmitword(function,data,devno);
  IF function = read_data
  THEN pus:= waitinput(devno)
  ELSE pus:= waitstatus(devno);
IF eoi THEN ps.eoi:= 1 ELSE ps.eoi:= 0;
END;
```

10.7 Clear Current Interrupt 10.7

If in the current register set the timeout field of the ps register has the value 0, the interrupt register of the device with the current level is assigned the value = 0.

10.7.1 IOCCI 10.7.1

Input/Output Clear Current Interrupt Value: 29_{Hex}

IC → $\boxed{\text{IOCCI}}$

device (curlevel):

interrupt=false

```
BEGIN
  clearcurrent;
END;
```

10.8 Execute Next Instruction After Clearing Interrupt 10.8

If in the current register set the timeout field of the ps register has the value 0, the interrupt register of the device with the current level is assigned the value 0, whereupon the next instruction on this level is executed.

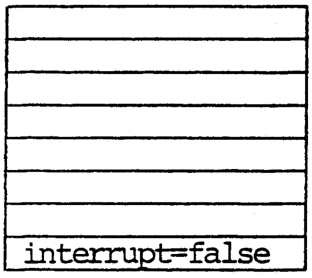
10.8.1 IONCI 10.8.1

Input/Output Execute Next Instruction After Clearing Interrupt

Value: 27_{Hex}

IC →	IONCI	next

device (curlevel):

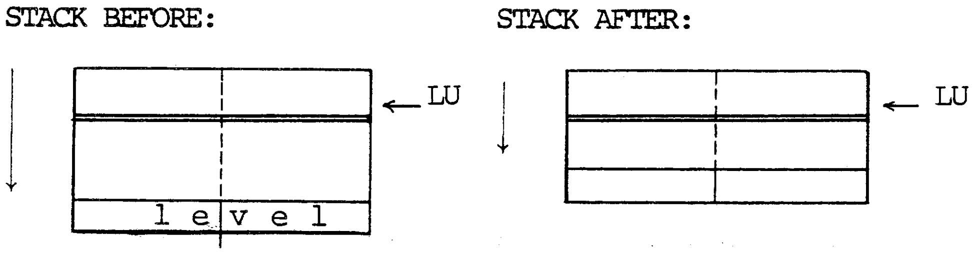


```
BEGIN
  clearcurrent;
  GOTO nextinstruction;
END;
```

10.9.1 CSLEV

Control Set LEVel Value: $1E_{Hex}$

IC → CSLEV

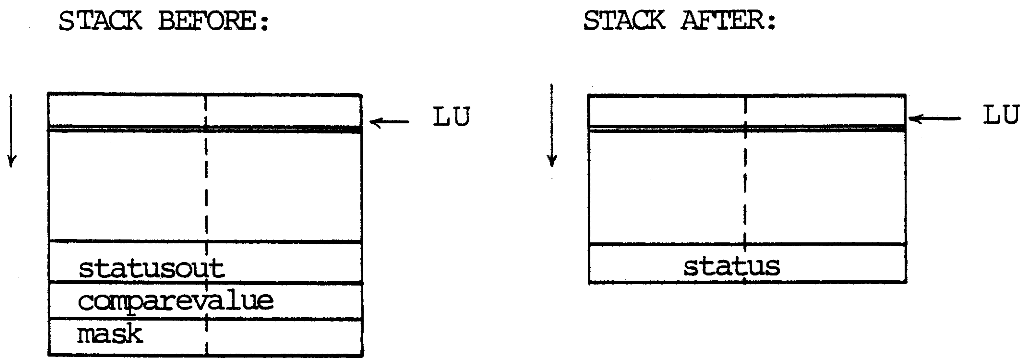


```
BEGIN
  activate(pop);
END;
```

10.10.1 IORSC 10.10.1

Input/Output Read Status and Compare Value: $2D_{Hex}$

IC → IORSC


STACK BEFORE:

STACK AFTER:

```
VAR
  devno: integer;
  status, mask, comparevalue, value: word;
BEGIN
  mask:= pop;
  comparevalue:= pop;
  status:= pop;
  devno:= ps.level;
  xmitword(readstatus,status,devno);
  value:= waitstatus(devno);
  IF (value AND mask)=comparevalue
  THEN pus:= value
  ELSE BEGIN
    device(devno).interrupt:= 0;
    pus:= status;
    pus:= comparevalue;
    pus:= mask;
    GOTO repeatinstruction;
  END;
END;
```

10.11 Get Current Device Address

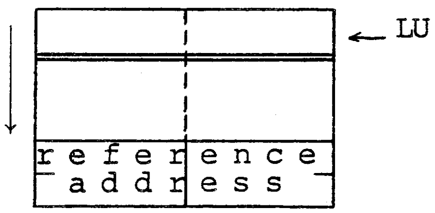
The parameter is interpreted as the address of a reference. The reference must be the address of a message of the kind 'channel message'. The result is the device address (device number) contained in the message. No input/output is performed.

10.11.1 IOCDA

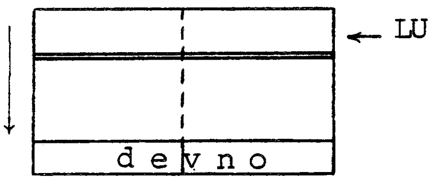
Input/Output Get Current Device Address Value: $2A_{Hex}$

IC → IOCDA

STACK BEFORE:



STACK AFTER:



```
VAR
  refadr: adr;
  ref: addr;
  kind: integer;
BEGIN
  refadr.disp:= pop;
  refadr.base:= basetype(pop);
  IF refadr.base.nilbit=1 THEN exception(nilpointer);
  ref:=memaddr(refadr);
  IF ref.base.nilbit THEN exception(nilreference);
  kind:=mem(adroffset(ref,kindoffset));
  IF kind >= 0 THEN exception(nochannel);
  pus:= kind AND #h7f;
END;
```

10.12 Initialize Block Transfer 10.12

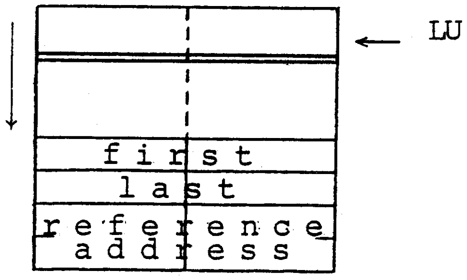
This instruction is used to initialize the contents of the stack preparatory to the execution of a read/write block instruction.

10.12.1 IOIBX 10.12.1

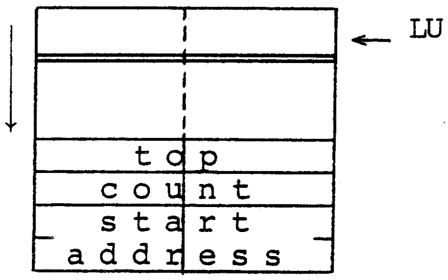
Input/Output Initialize Block Xfer Value: $2B_{Hex}$

IC → IOIBX

STACK BEFORE:



STACK AFTER:



```
VAR
  refadr: adr;
  ref: addr;
  startadr: adr;
  kind: integer;
  first, msgsize, top, count: word;
BEGIN
  refadr.disp:= pop;
  refadr.base:= basetype(pop);
  IF refadr.base.nilbit=1 THEN exception(nilpointer);
  ref:=memaddr(refadr);
  IF ref.base.nilbit=1 THEN exception(nilref);
  kind:=
  mem(addroffset(ref,kindoffset));
  IF kind<0 THEN exception(datamesexcp);
  msgsize:= mem(adroffset(ref,sizeoffset));
  top:= pop + 1;
  first:= pop;
  IF msgsize < ((top + 1) DIV 2) THEN exception(sizetoosmall);
  IF first >= top THEN exception(lastfirst);
  count:= top - first;
  startadr:=memadr(addroffset(ref,startoffset));
  startadr.disp:= startadr.disp + first;
  pus:= top;
  pus:= count;
  pus:= word(startadr.base);
  pus:= startadr.disp;
END;
```

10.13 Write Block of Bytes/Words

A block of bytes/words is transferred from a buffer, one byte/word at a time, to the dataout register of the device with the current interruption level as its device number, until the buffer is empty or a timeout occurs.

The buffer is defined by the operand and parameter 1: the former is the address (ADDR) of the first byte/word of the buffer, and the latter the number of bytes in the buffer.

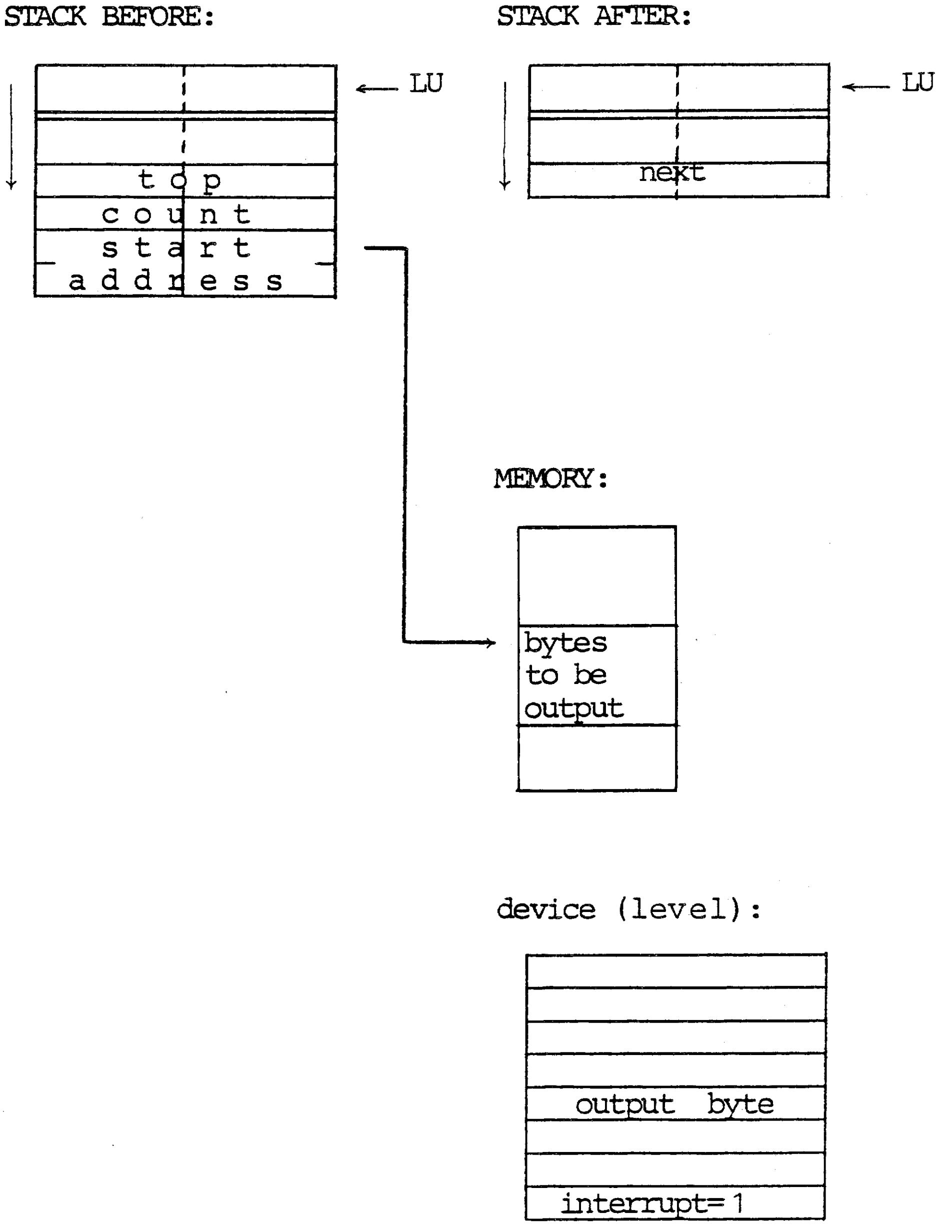
For each byte/word transferred the instruction is repeated and the stack parameters remain in the register stack.

When all bytes/words have been transferred or a timeout has occurred the residual count is subtracted from the 'top' parameter to produce the result 'next', which is the index of the byte following the last byte/word written. Depending on the used instruction, execution after the transfer of the last byte/word is either continued or suspended until a succeeding interrupt has occurred.

10.13.1 IOWBB

Input/Output Write Block of Bytes   Value: 73_{Hex}

IC → IOWBB

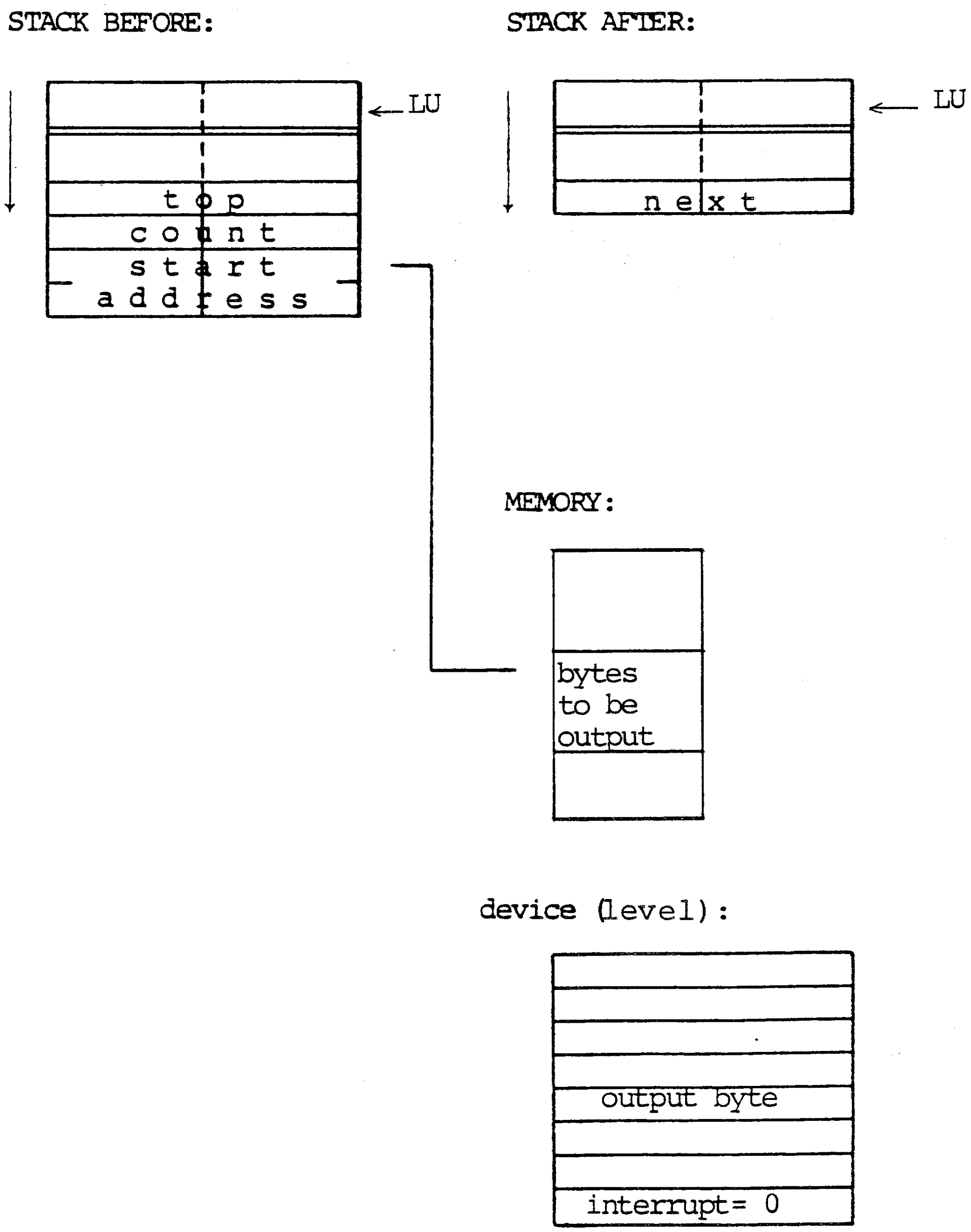


```
VAR
  startadr : adr;
  finish : boolean;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    finish:= getadrbyte(startadr);
    xmitword(writedata,membyte(startadr),ps.level);
    IF NOT finish GOTO repeatinstruction
    ELSE BEGIN
      device(ps.level).interrupt:=1;
      blockend;
    END;
  END;
END;
```

10.13.2 IOWBBC

Input/Output Write Block of Bytes and Clear Value: 72_{Hex}

IC → IOWBBC

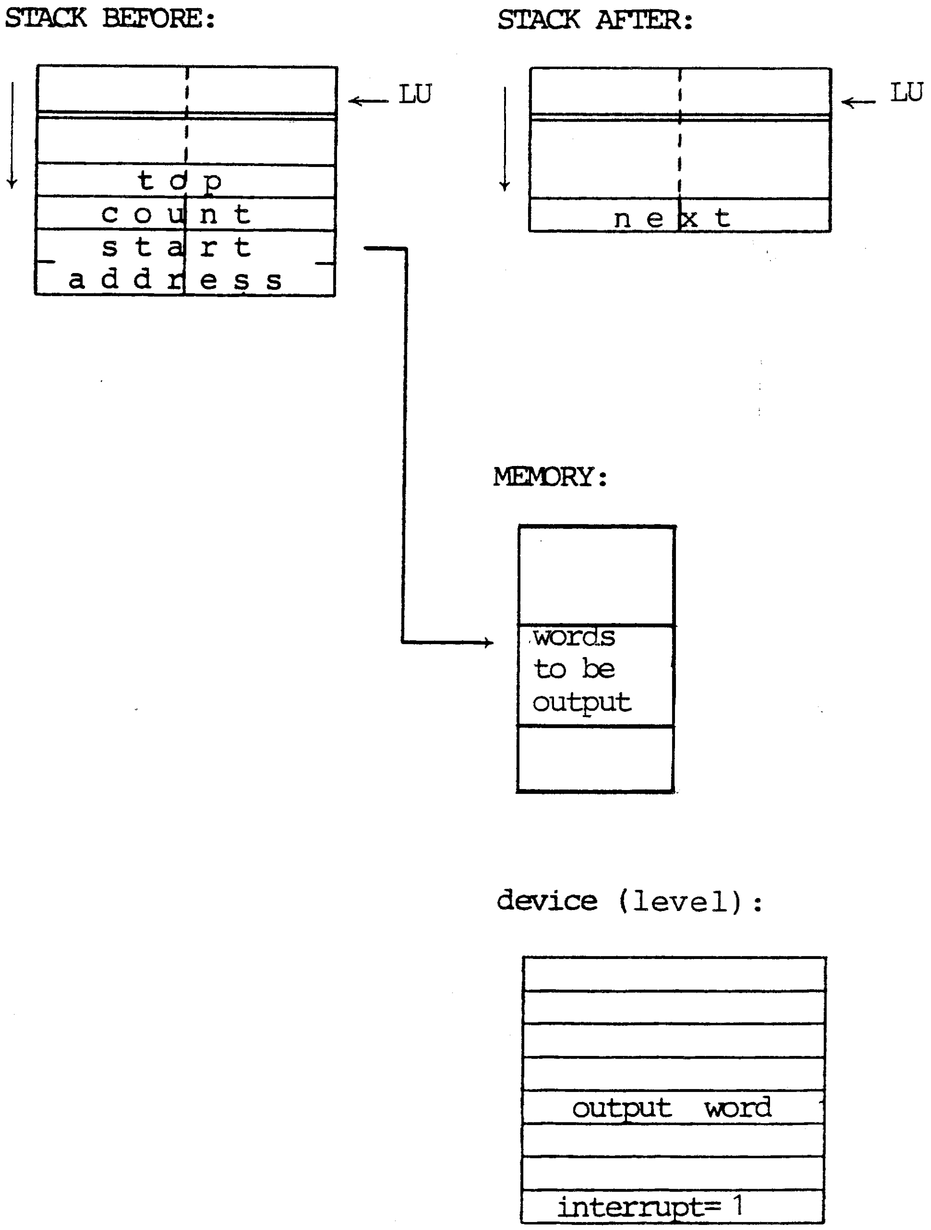


```
VAR
  startadr : adr;
  finish : boolean;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    finish:= getadrbyte(startadr);
    xmitword(writedata,membyte(startadr),ps.level);
    IF NOT finish THEN GOTO repeatinstruction
    ELSE blockend
  END;
END;
```

10.13.3 IOWBW

Input/Output Write Block of Words   Value: 77_{Hex}

IC → IOWBW

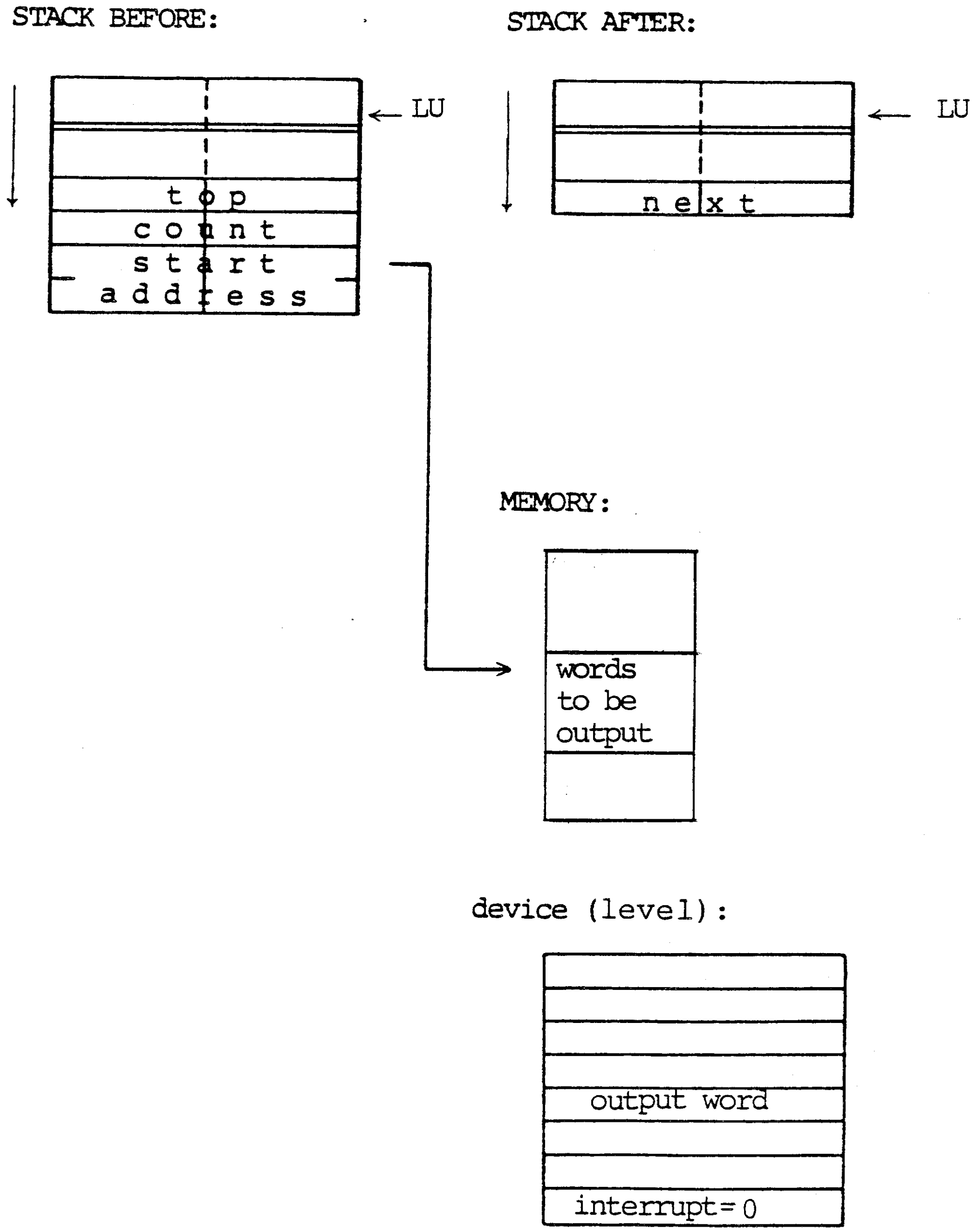


```
VAR
  startaddr : adr;
  finish : boolean;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    finish:= getaddrword(startaddr);
    xmitword(writedata,memword(startaddr),ps.level);
    IF NOT finish THEN GOTO repeatinstruction
    ELSE BEGIN
      device(ps.level).interrupt:=1;
      blockend;
    END;
  END;
END;
```

10.13.4 IOWBWC

Input/Output Write Block of Words and Clear Value: 76_{Hex}

IC → $\boxed{\text{IOWBWC}}$



```
VAR
  startadr : adr;
  finish : boolean;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    finish:= getadrbyte(startadr);
    xmitword(writedata,memword(startadr),ps.level);
    IF NOT finish THEN GOTO repeatinstruction
    ELSE blockend
  END;
END;
```

10.14 Read Block of Bytes/Words

A block of bytes/words is transferred to a buffer, one byte/word at a time, from the datain register of the device with the current interruption level as its device number, until the buffer is full or a timeout occurs or the eoi register of the device contains the value 1. The value of eoi is always transferred to the eoi field of the ps register.

The buffer is defined by operand 1 and operand 2: the former is the address (ADDR) of the first byte of the buffer, and the latter the number of bytes in the buffer.

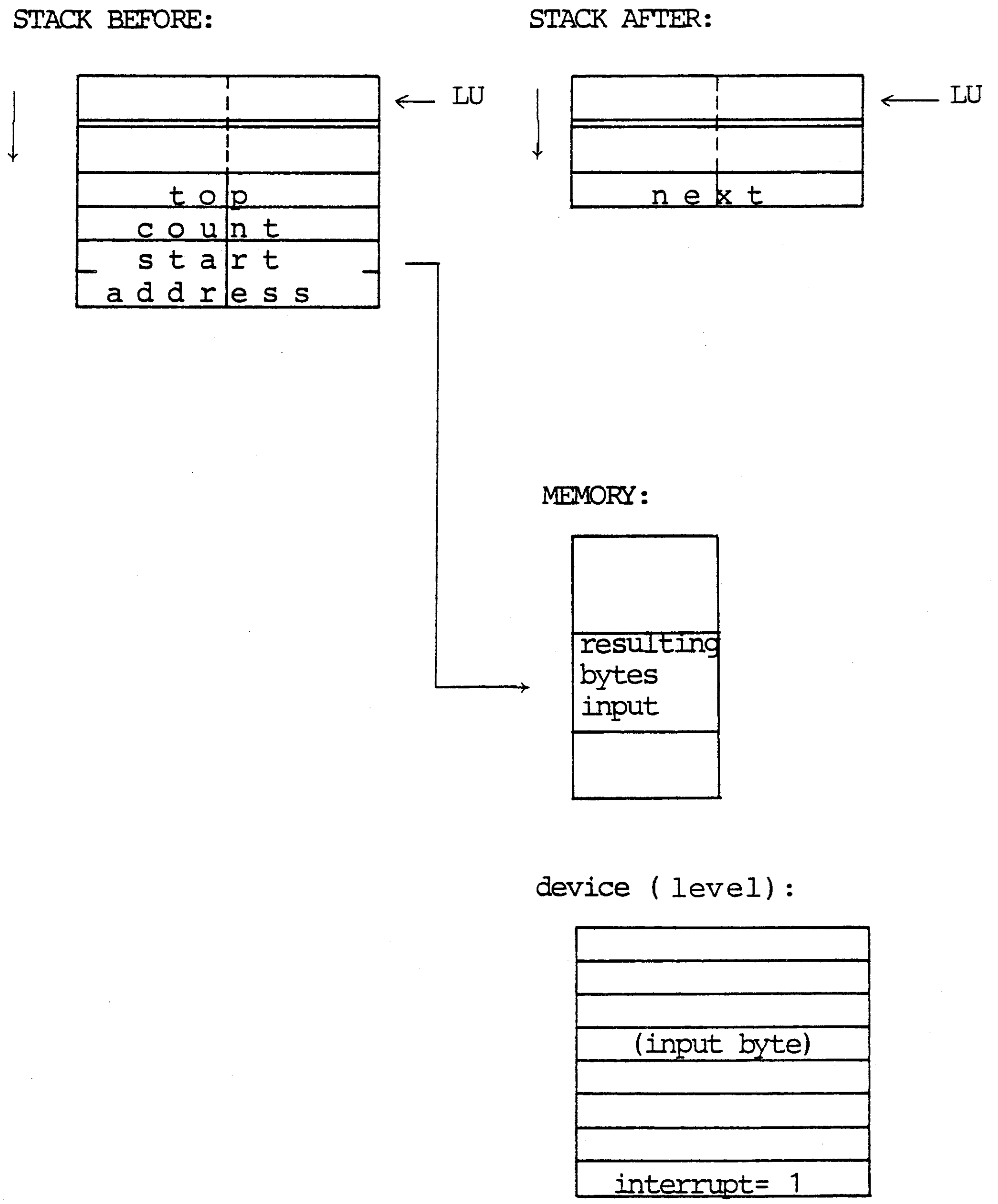
For each byte/word transferred the instruction is repeated and the stack parameters remain in the register stack.

When the transfer is finished, the residual count is subtracted from the 'top' parameter to produce the result 'next', which is the index of the byte following the last byte/word read. Depending on the used instruction, execution after the transfer of the last byte/word is either continued or suspended until a succeeding interrupt has occurred.

10.14.1 IORBB

Input/Output Read Block of Bytes   Value: 71_{Hex}

IC → IORBB

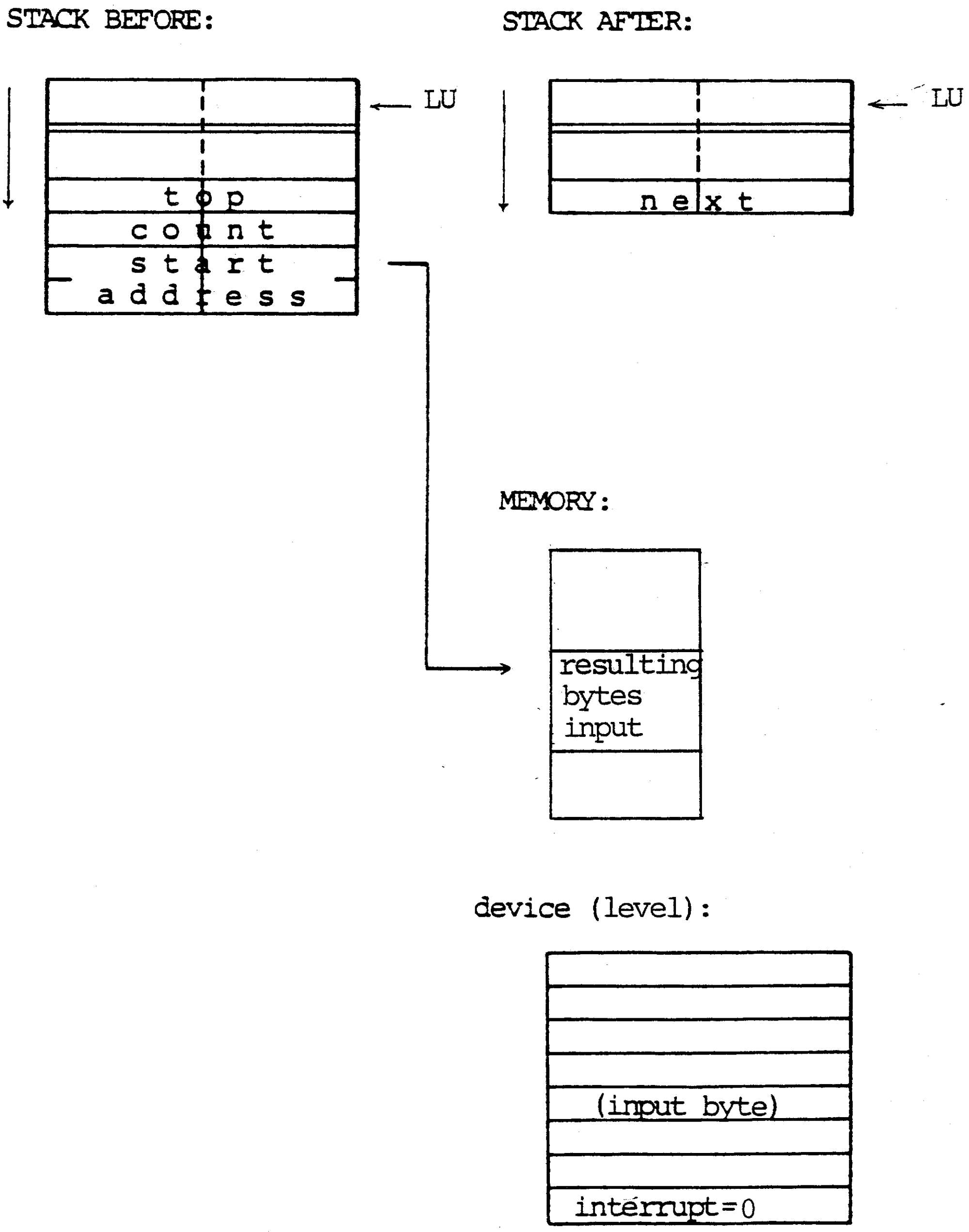


```
VAR
  startadr : adr;
  finish : boolean;
  data: word;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    xmitword(readdata,0,ps.level);
    data:= waitinput(ps.level);
    IF eoi THEN BEGIN ps.eoi:= 1; blockend; END
    ELSE BEGIN
      finish:=getadrbyte(startadr);
      memword(startadr):=data;
      IF NOT finish THEN GOTO repeatinstruction
      ELSE BEGIN
        device(ps.level).interrupt:=1;
        blockend;
      END;
  END;
END;
```

10.14.2 IORBBC

Input/Output Read Block of Bytes and Clear Value: 70_{Hex}

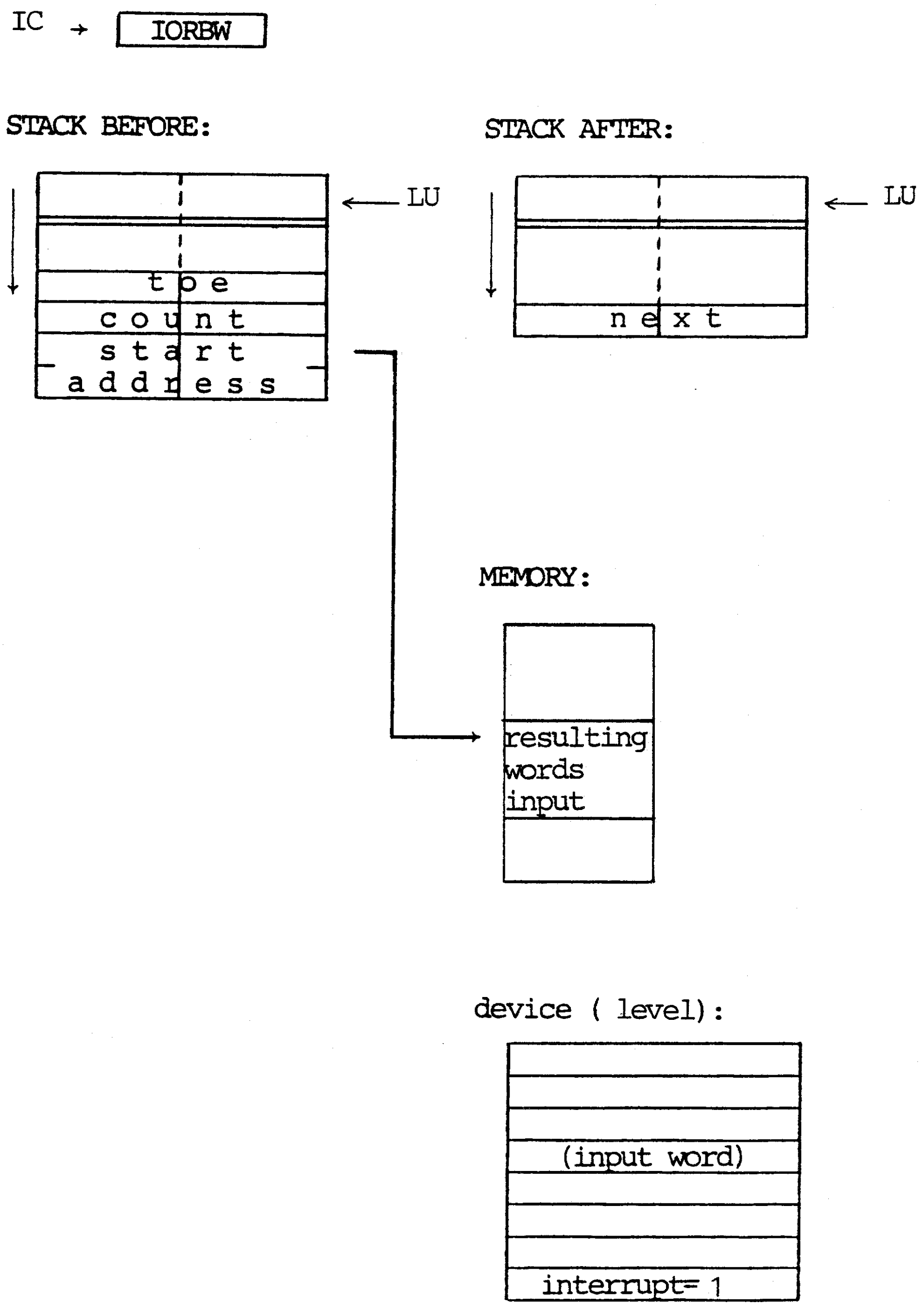
IC → IORBBC



```
VAR
  startadr : adr;
  finish : boolean;
  data: word;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    xmitword(readdata,0,ps.level);
    data:= waitinput(ps.level);
    IF eoi THEN BEGIN ps.eoi:= 1; blockend; END
    ELSE BEGIN
      finish:=getaddrbyte(startadr);
      memword(startadr):=data;
      IF NOT finish
      THEN GOTO repeatinstruction
      ELSE blockend;
    END;
  END;
END;
```

10.14.3 IORBW

Input/Output Read Block of Words Value: 75_{Hex}

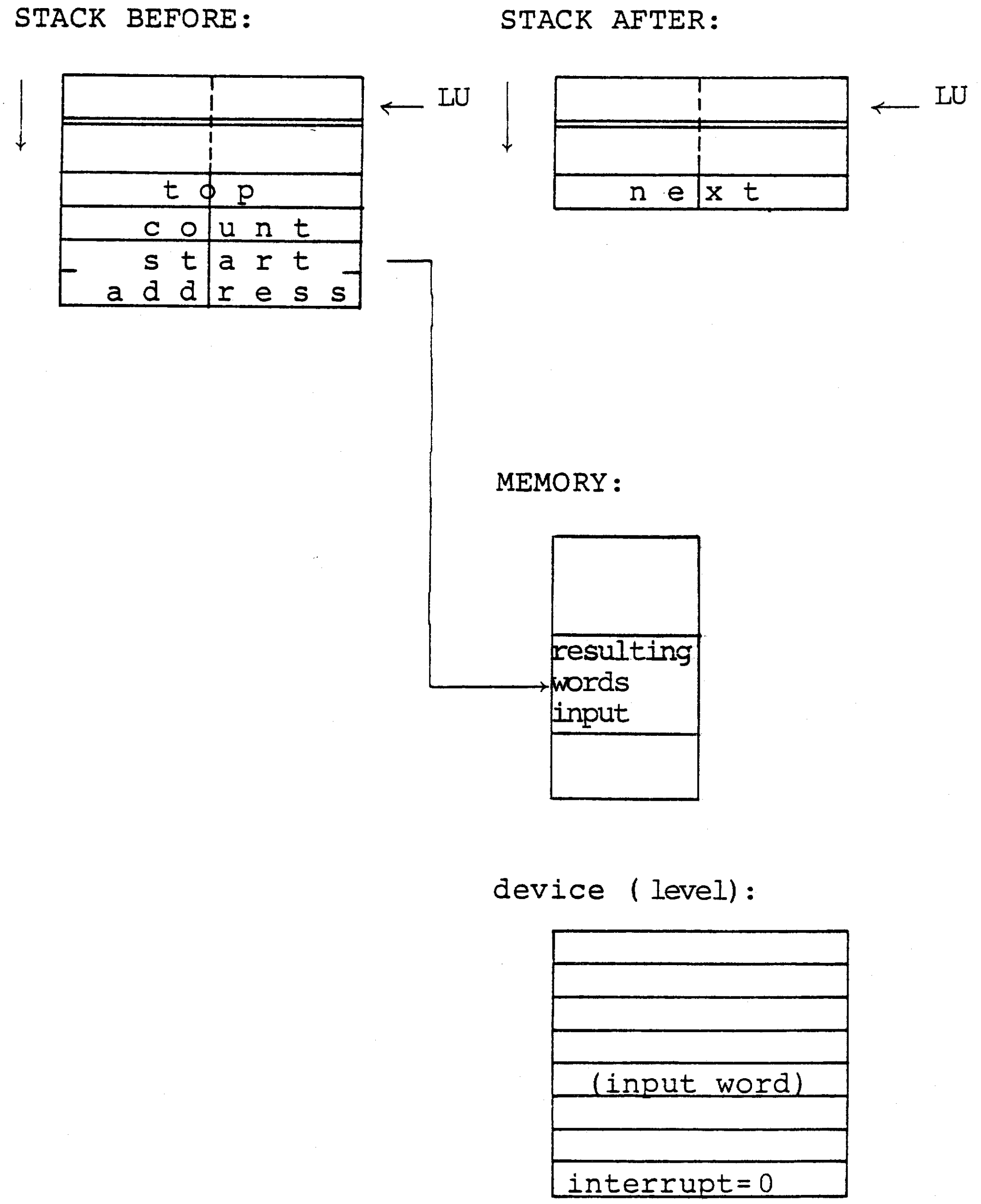


```
VAR
  startadr : adr;
  finish : boolean;
  data: word;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    xmitword(readdata,0,ps.level);
    data:= waitinput(ps.level);
    IF eoi
    THEN BEGIN
      ps.eoi:= 1;
      blockend;
    END
    ELSE BEGIN
      finish:=getadrword(startadr);
      memword(startadr):=data;
      IF NOT finish THEN GOTO repeatinstruction
      ELSE BEGIN
        blockend;
        device(ps.level).interrupt:=1;
      END;
    END;
  END;
END;
```

10.14.4 IORBWC

Input/Output Read Block of Words and Clear Value: 74_{Hex}

IC → IORBWC



```
VAR
  startadr : adr;
  finish : boolean;
  data: word;
BEGIN
  IF NOT blockstart THEN blockend
  ELSE BEGIN
    xmitword(readdata,0,ps.level);
    data:= waitinput(ps.level);
    IF eoi THEN BEGIN ps.eoi:= 1; blockend; END
    ELSE BEGIN
      finish:=getadrword(startadr);
      memword(startadr):=data;
      IF NOT finish THEN GOTO repeatinstruction
      ELSE blockend;
    END;
  END;
END;
```

11. MONITOR CONTROL AND SYNCHRONIZATION

The instructions described in this chapter support monitor control, synchronization, and message buffer manipulation.

The instructions operate on a number of structures, including the following:

- incarnation stack (see section 3.2)
- semaphores (see section 3.10)
- messages (see section 3.11)

The algorithmic descriptions in this chapter employ a number of data structures, common variables, functions, and procedures. Those which have not been defined in previous chapters are defined below.

```
TYPE
  semstatetype = (open, passiv, locked);

PROCEDURE setpswait(inst: word);
BEGIN
  WITH inst AS pstype DO
  BEGIN
    IF pss = 1 THEN ps.pss:= 1;
    IF pst = 1 THEN ps.pst:= 1;
    IF psi = 1 THEN ps.psi:= 1;
  END;
END;

FUNCTION control(inst: word): boolean;
BEGIN
  WITH inst AS bitword DO control:= (inst(15)=1);
END;

FUNCTION semstate(semword: word): semstatetype;
BEGIN
  WITH semword AS bitword DO
    IF semword(15) THEN semstate:=passiv
    ELSE IF semword(0) THEN semstate:=locked
                       ELSE semstate:=open;
END;
```

```
PROCEDURE sendcontrol( cw: word; level: leveltype);
BEGIN
  device(level).function:= writecontrol;
  device(level).control:= cw;
END;


FUNCTION notnil( w: word): word;
BEGIN
  WITH w AS bitword DO w(15):= 0;
  notnil:= w;
END;


FUNCTION inmem( w: word): boolean;
BEGIN
  WITH w AS bitword DO
  inmem:= (w(1)=0);
END;


FUNCTION wqempty(w: word): boolean;
BEGIN
  WITH w AS bitword DO wqempty:=w(15);
END;


FUNCTION locked(w: word): word;
BEGIN
  WITH w AS bitword DO
  BEGIN
    w(0):= 1;
    w(1):= 1;
    w(15):= 0;
  END;
  locked:= w;
END;


PROCEDURE getsemref(control: boolean;
                    VAR refadr, semadr: adr;
                    VAR sem : double;
                    VAR controlword: word);
BEGIN
  semadr.disp:=pop;
  semadr.base:=basetype(pop);
  IF semadr.base.nilbit=1 THEN exception(nilpointer);
  sem:=memdouble(semadr);
  refadr.disp:=pop;
  refadr.base:=basetype(pop);
  IF control THEN controlword:= pop;
END;


PROCEDURE activate(l: level; r: 0..121);
BEGIN
  IF l.level0 THEN activeflags(curreg):=l.prio
              ELSE device(l.level).interrupt:=0;
END;
```

```
PROCEDURE deactivate(l: level);
BEGIN
  IF l.level0 THEN activeflags(curreg):=0
             ELSE device(l.level).interrupt:=0;
END;


PROCEDURE activatemap(l: leveltype; regset: 0..121);
BEGIN
  activate(l,curreg);
  IF NOT l.level0
  THEN intmap(l.level):=curreg;
  ps.level:= l;
END;


PROCEDURE deactivatemap(l: leveltype; regset: 0..121);
BEGIN
  IF l.level0 THEN deactivate(l)
  ELSE BEGIN
    intmap(l.level):= dummyreg;
    device(l.level).interrupt:=0;
  END;
END;


FUNCTION regindex(w: word): 0..121;
BEGIN
  regindex:= (w DIV 8) AND #h7F;
END;


FUNCTION nilreg(w: word): boolean;
BEGIN
  WITH w AS bitword DO
    nilreg:= (w(15) = 1);
END;


PROCEDURE clearpswait;
BEGIN
  ps.pst:= 0;
  ps.pss:= 0;
  ps.psi:= 0;
  ps.mr:= 0;
END;
```

11.1 Signal/Return 11.1

Determined by the state of a semaphore a message is either put into the semaphore queue or the first process incarnation is removed from the semaphore queue, handled the message and activated.

The CSIGN instruction has the addresses of the reference and semaphore variables in the registerstack. The CRELE instruction retrieves the semaphore address indirectly through the message header pointed out by the reference variable addressed by the address on top of the registerstack.

Common routines for Signal/Return:

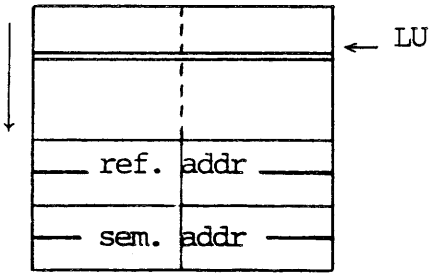
```
PROCEDURE sopen(semadr: adr; sem,ref: double);
VAR
  first,last: double;
BEGIN
  last:= sem;
  first:= memdouble(adr(sem));
  memdouble(adr(last)):= ref;
  memdouble(adr(ref)):= first;
  memdouble(semadr):= ref;
END;


PROCEDURE spassiv(semadr: adr; sem,ref: double);
BEGIN
  memdouble(adr(ref)):= ref;
  memdouble(semadr):= ref;
END;


PROCEDURE slocked(semadr: adr; sem,ref: double);
VAR
  first,second,last,thisps: word;
  memregset, helpadr: adr;
BEGIN
  IF NOT inmem(sem.w1 )
  THEN WITH registerset(regindex(sem.w1 )) DO
  BEGIN
    ps.mr:= 1;
    second:=lf; last:= lm;
    lf:= ref.w1;
    lm:= ref.w2;
    activate(ps.level,sem.w1 );
  END
  ELSE BEGIN
    first:= sem.w1;
    memregset.base:= monitorreg.memregbase.base;
    memregset.disp:= monitorreg.memregdisp + first;
    thisps:= mem(memregset);
    WITH thisps AS pstype DO thisps.mr:=1;
    mem(memregset):=thisps;
    second:= mem( adroffset(memregset,lfoffset) );
    last:= mem( adroffset( memregset,lmoffset ) );
    memdouble( adroffset(memregset, res0offset) ):= ref;
    IF wqempty(monitorreg.last)
    THEN monitorreg.first:= first + monitorreg.memregdisp
    ELSE BEGIN
      helpadr.base:= memregset.base;
      helpadr.disp:= monitorreg.last + lfoffset;
      mem(helpadr):= first + monitorreg.memregdisp;
    END;
    setinterrupt(monitorreg.monitorlevel);
    monitorreg.last:= first + monitorreg.memregdisp;
  END;
  IF NOT nilreg(second)
  THEN BEGIN
    IF inmem(second)
    THEN BEGIN
      memregset.base:= monitorreg.memregbase.base;
      memregset.disp:= monitorreg.memregdisp + second;
      mem(adroffset(memregset,lmoffset)):= last;
    END
    ELSE registerset(regindex(second)).lm:=last;
  END;
  mem(semadr):=second;
END;
```

11.1.1 CSIGN

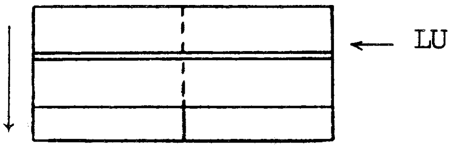
Control SIGN Value: 10_{Hex}

IC → CSIGN

STACK BEFORE:



STACK AFTER:



The state of the semaphore determines the operation performed. The data structure modifications are not shown.

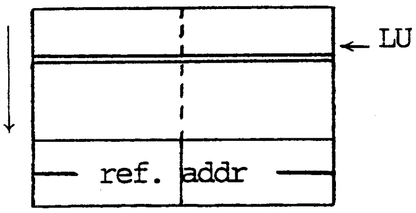
```
VAR
  ref, sem: double;
  refadr, semadr: adr;
  dummyword: word;
BEGIN
  getsemref(false,refadr,semadr,sem,dummyword);
  IF refadr.base.nilbit=1 THEN exception(nilpointer);
  ref:=memdouble(refadr);
  IF addr(ref).base.nilbit=1 THEN exception(nilreference);
  IF addr(ref).nullbyte <> 0 THEN exception(reflocked);
  memaddr(refadr).base.nilbit:= 1;
  CASE semstate(sem.w1) OF
    open:   sopen(semadr,sem,ref);
    passiv: spassiv(semadr,sem,ref);
    locked: slocked(semadr,sem,ref);
  END;
END;
```

11.1.2 CRELE

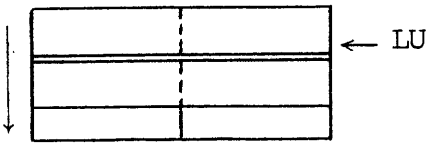
Control RELEase Message Value: 12_{Hex}

IC → | CRELE | parameter |

STACK BEFORE:



STACK AFTER:



```
VAR
  refadr, semadr: adr;
  ref, sem: double;
BEGIN
  refadr.disp:= pop;
  refadr.base:= basetype(pop);
  IF refadr.nilbit=1 THEN exception(nilpointer);
  ref:=memdouble(refadr);
  IF addr(ref).nullbyte<>0 THEN exception(reflocked);
  semadr:= memadr(addroffset(addr(ref),nextbyte)));
  sem:= memdouble(semadr);
  memaddr(refadr).base.nilbit:= 1;
  CASE semstate(sem.w1) OF
    open:   sopen(semadr,sem,ref);
    passiv: spassiv(semadr,sem,ref);
    locked: slocked(semadr,sem,ref);
  END;
END;
```

11.2 Wait

The instructions in this group are used to wait for some event to occur. Most commonly the receiption of a message, but some of the instructions are also able to respond to interrupts and/or activation by the timersystem. When a process incarnation performs a wait instruction on a semaphore and a message is waiting (the semaphore is open) the message is removed from the queue of messages. When a process incarnation performs a wait instruction on a passiv or locked semaphore, the operation is partitioned into two parts. The first part puts the incarnation into the queue of process incarnations waiting on the particular semaphore and deschedules the incarnation. Later on when another incarnation invokes the waiting incarnation, the second part of the wait instruction stores the received message and continues execution. If the wait instruction waits for activation by the timersystem and/or by an interrupt, the first part of the wait instruction just deschedules the incarnation and the second part continues execution.

When a wait instruction waits for more than one event to occur, the following rules of precedence is used.

The first part of wait:

1) own.timer = 0
   the incarnation continues execution as activated by the timer

2) a message is already received
   the incarnation continues execution as activated by a message

The second part of wait:

1) the receiption of a message
2) activation by the timersystem
3) the receiption of an interrupt

The partitioning of the wait instructions are done by means of the resume bit in the IB register in the registerset of the process incarnation executing the wait instruction. The resume bit is set during the first part of the wait. Each time a new process incarnation is scheduled for execution the resume bit is tested. If the instruction is a wait and the bit is set, the second part of the instruction is executed.

To allow the unpartitioned execution of the first part of the wait and the deliver of a controlword to an external device, a special class of wait instructions is able to perform this action.

Common routines for wait:

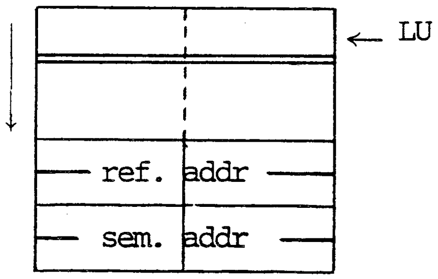
```
PROCEDURE wopen(inst: byte; controlword: word;
                 refadr, semadr: adr; sem: double);
VAR
  first, last, newfirst: double;
BEGIN
  last:=sem;
  first:=memdouble(adrdouble(last));
  IF (first.w2<>sem.w2) OR (first.w1<>sem.w1)
  THEN BEGIN (* semaphore queue not going empty *)
    newfirst:=memdouble(adr(first));
    memdouble(adr(last)):=newfirst;
  END
  ELSE memaddr(semadr).base.nilbit:= 1; (* queue going empty *)
  memdouble(refadr):= first;
  IF inst<>cwait
  THEN BEGIN
    IF control(inst)  THEN sendcontrol(controlword);
    pus:= 1;
  END;
  GOTO nextprocess;
END;


PROCEDURE wlocked(passiv: boolean; inst: byte;
                  controlword: word;
                  refadr, semadr: adr; sem: double);
VAR
  first, last: word;
  memregset: adr;
BEGIN
  pus:=lf;
  pus:=lm;
  pus:=word(refadr.base);
  pus:=refadr.disp
  deactivate(ps.level);
  setpswait(ps, inst);
  pus:= word(semadr.base);
  pus:= semadr.disp;
  IF control(inst) THEN sendcontrol(controlword);
  lf:= locked(context) + 1; (* nil *)
  IF passiv
  THEN BEGIN
    lm:= locked(context);
    mem(semadr):=locked(context);
  END

  ELSE BEGIN
    first:=sem.w1;
    IF NOT inmem(first)
    THEN BEGIN
      last:= registerset(regindex(first)).lm;
      registerset(regindex(first)).lm:=locked(context);
    END
    ELSE BEGIN
      memregset.base:=monitorreg.memregbase.base;
      memregset.disp:=monitorreg.memregdisp + first;
      last:= mem(adroffset(memregset,lmoffset));
      mem(adroffset(memregset,lmoffset)):= locked(context);
    END;
    IF NOT inmem(last)
    THEN registerset(regindex(last)).lf:= locked(context)
    ELSE BEGIN
      memregset.base:= monitorreg.memregbase.base;
      memregset.disp:= monitorreg.memregdisp + last;
      mem(adroffset(memregset,lfoffset)):= locked(context);
    END;
    lm:= last;
  END;
  GOTO resumeinstruction;
END;

PROCEDURE wait(control: boolean; inst: byte);
VAR
  refadr, semadr: adr;
  ref, sem: double;
  controlword: word;
BEGIN
  getsemref(control(inst),refadr,semadr,sem,controlword);
  IF refadr.base.nilbit=1 THEN exception(nilpointer);
  ref.w1:= mem(refadr);
  IF NOT adr(ref).base.nilbit=1 THEN exception(refnotnil);
  CASE semstate(sem.w1) OF
    open:   wopen(inst, controlword, refadr, semadr, sem);
    passiv: wlocked(true, inst, controlword, refadr, semadr, sem);
    locked: wlocked(false, inst, controlword, refadr, semadr, sem);
  END;
END;

FUNCTION timerwait(control: boolean; inst: byte): boolean;
VAR
  incadr: adr;
BEGIN
  incadr.base:= sb.base;
  incadr.disp:= gf;
  timer:= mem(incadr);
  IF timer<=0
  THEN WITH inst AS pstype DO
  BEGIN
    IF inst.pss = 1
    THEN BEGIN
      pop; pop; (* remove semadr *)
      pop; pop; (* remove refadr *)
    END;
    IF control THEN sendcontrol(pop);
    timerwait:= false;
  END
  ELSE timerwait:= true;
END;


PROCEDURE resumemultiplewait;
VAR
  refadr: adr;
BEGIN
  IF (ps.pss=1) AND (ps.mr=1)
  THEN BEGIN (* message received *)
    pop; pop; (* remove semaddr *)
    refadr.disp:= pop;
    refadr.base:= basetype(pop);
    mem(refadr):=lf;
    mem(adroffset(refadr,2)):=lm;
    lm:= pop; lf:= pop;
    clearpswait;
    pus:=1; (* semaphore activation *)
  END;
  IF (ps.pst=1) AND (own.timer=0)
  THEN BEGIN
    IF ps.pss=1
    THEN BEGIN
      pop; pop; (* remove semaddr *)
      pop; pop; (* remove refaddr *)
      lm:= pop; lf:= pop;
    END;
    clearpswait;
    pus:= 2; (* timer activation *)
  END
  ELSE BEGIN
    IF ps.psi=1
    THEN BEGIN
      IF ps.pss=1
      THEN BEGIN
        unchain(context); (* defined in section 11.4 *)
        pop; pop;
        pop; pop;
        lm:= pop; lf:= pop;
      END;
      clearpswait;
      pus:= 0; (* interrupt activation *)
    END
    ELSE BEGIN
      deactivate(ps.level);
      GOTO resumeinstruction;
    END; (* else psi *)
  END; (* else pst *)
END;
```

11.2.1 CWAIT

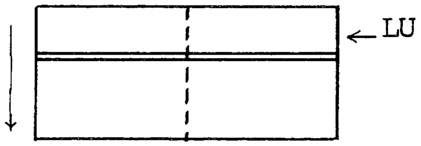
Control WAIT Value: 40_{Hex}

IC → CWAIT

STACK BEFORE:



STACK AFTER:



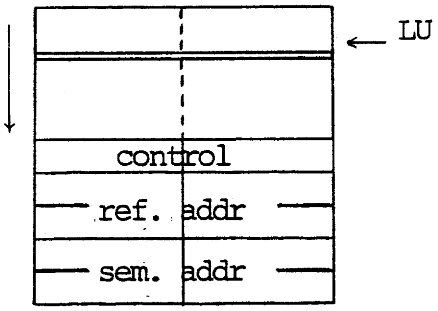
```
(* first part *)


BEGIN
  wait(false,cwait);
END;


(* resume part *)


VAR
  refadr: adr;
BEGIN
  IF ps.mr=1
  THEN BEGIN
    pop; pop; (* remove semaddr *)
    refadr.disp:= pop;
    refadr.base:= pop;
    mem(refadr):=lf;
    mem(adroffset(refadr,2)):=lm;
    lm:= pop; lf:= pop;
    clearpswait;
  END
  ELSE BEGIN
    deactivate(ps.level);
    GOTO resumeinstruction;
  END;
END;
```

11.2.2 MCIS

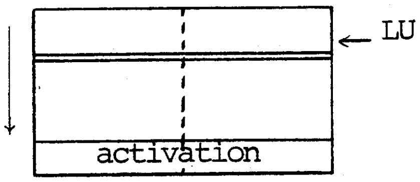
Multiple Control Interrupt and Semaphore   Value: 61_{Hex}

IC → $\boxed{\text{MCIS}}$

STACK BEFORE:



STACK AFTER:



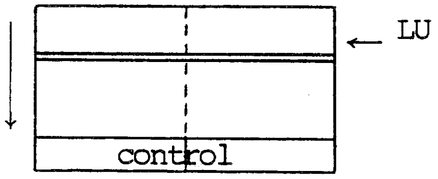
```
(* first part *)

BEGIN
  wait(true,mcis);
END;

(* resume part *)

BEGIN
  resumemultiplewait;
END;
```

11.2.3 MCIT

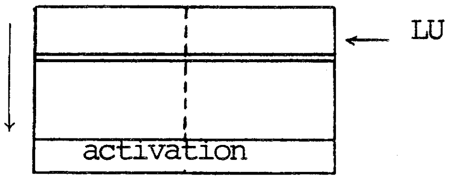
Multiple Control Interrupt and Timer   Value: $A1_{Hex}$

IC → $\boxed{\text{MCIT}}$

STACK BEFORE:



STACK AFTER:



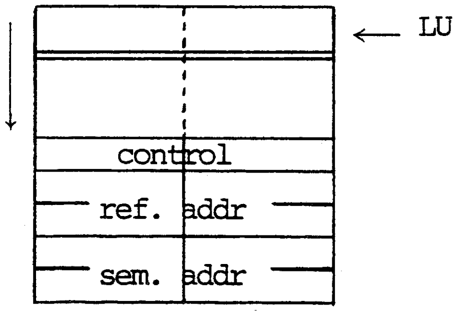
```
(* first part *)


BEGIN
  IF timerwait(true,mcit)
  THEN BEGIN
    setpswait(mcit);
    deactivate(ps.level);
    GOTO resumeinstruction;
  END;
END;


(* second part *)


BEGIN
  resumemultiplewait;
END;
```

11.2.4 MCIST

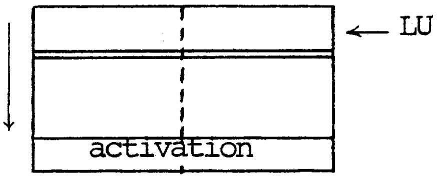
Multiple Control Interrupt, Semaphore, and Timer Value: E1$_{\text{Hex}}$

IC → MCIST

STACK BEFORE:



STACK AFTER:



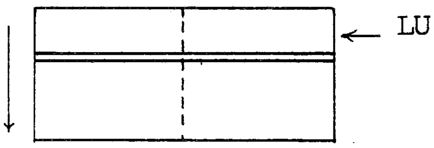
```
(* first part *)

BEGIN
  IF timerwait(true, mcist)
  THEN wait(true, mcist);
END;

(* second part *)

BEGIN
  resumemultiplewait;
END;
```

11.2.5 MWI

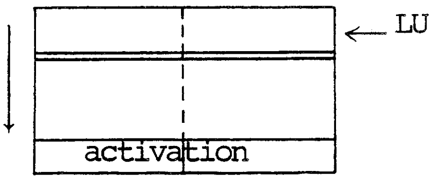
Multiple Wait Interrupt   Value: 20_{Hex}

IC → MWI

STACK BEFORE:



STACK AFTER:



(* first part *)

```
BEGIN
  setpswait(mwi);
  deactivate(ps.level);
  GOTO resumeinstruction;
END;
```

(* second part *)

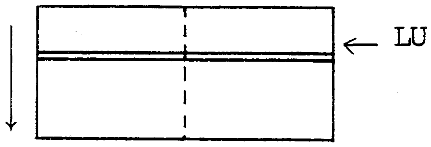
```
BEGIN
  resumemultiplewait;
END;
```

11.2.6 MWT

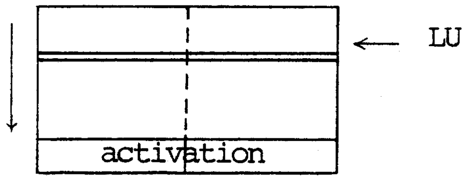
Multiple Wait Timer   Value: 80_{Hex}

IC → MWT

STACK BEFORE:



STACK AFTER:

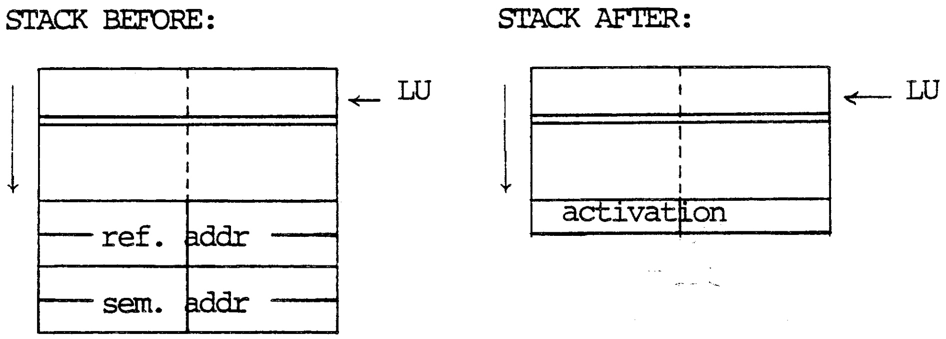


```
(* first part *)


BEGIN
  IF timerwait(false, mwt)
  THEN BEGIN
    setpswait(mwt);
    deactivate(ps.level);
    GOTO resumeinstruction;
  END;
END;


(* second part *)


BEGIN
  resumemultiplewait;
END;
```

11.2.7 MWIS

Multiple Wait Interrupt and Semaphore   Value: 60_{Hex}

IC → MWIS


STACK BEFORE:

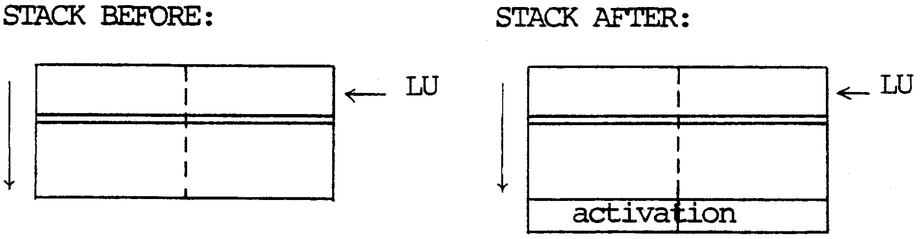
STACK AFTER:

```
(* first part *)

BEGIN
  wait(false, mwis);
END;

(* second part *)

BEGIN
  resumemultiplewait;
END;
```

11.2.8 MWIT

Multiple Wait Interrupt and Timer   Value: $A0_{Hex}$

IC → MWIT



(* first part *)

```
BEGIN
  IF timerwait(false, mwit)
  THEN BEGIN
    setpswait(mwit);
    deactivate(ps.level);
    GOTO resumeinstruction;
  END;
END;
```

(* second part *)

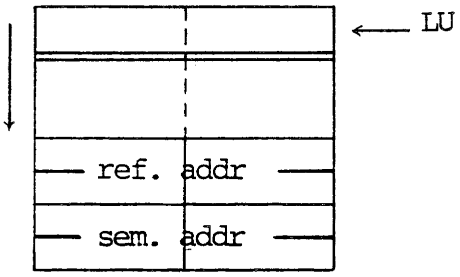
```
BEGIN
  resumemultiplewait;
END;
```

11.2.9 MWST

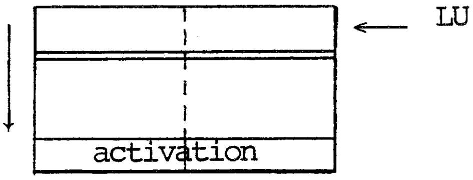
Multiple Wait Semaphore and Timer   Value: C0$_{\text{Hex}}$

IC → MWST

STACK BEFORE:



STACK AFTER:



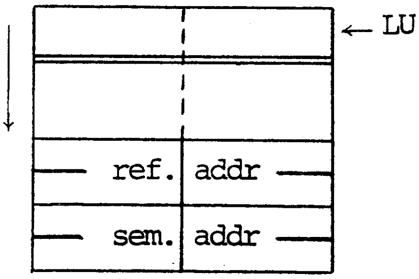
```
(* first part *)

BEGIN
  IF timerwait(false, mwst)
  THEN wait(false,mwst);
END;

(* second part *)

BEGIN
  resumemultiplewait;
END;
```

11.2.10 MWIST

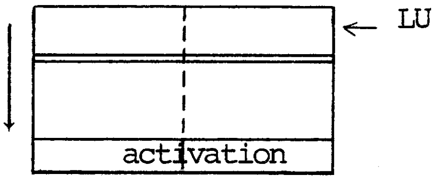
Multiple Wait Interrupt, Semaphore, and Timer   Value: E0_{Hex}

IC → MWIST

STACK BEFORE:



STACK AFTER:



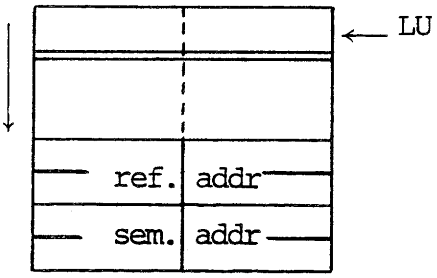
```
(* first part *)

BEGIN
  IF timerwait(false, mwist)
  THEN wait(false,mwist);
END;

(* second part *)

BEGIN
  resumemultiplewait;
END;
```

11.2.11 CSENS

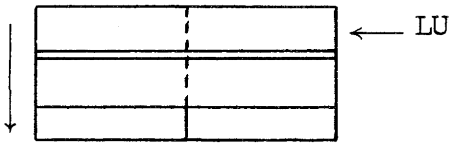
Control SENSe Semaphore　　　　Value: 15_{Hex}

IC → CSENS

STACK BEFORE:



STACK AFTER:



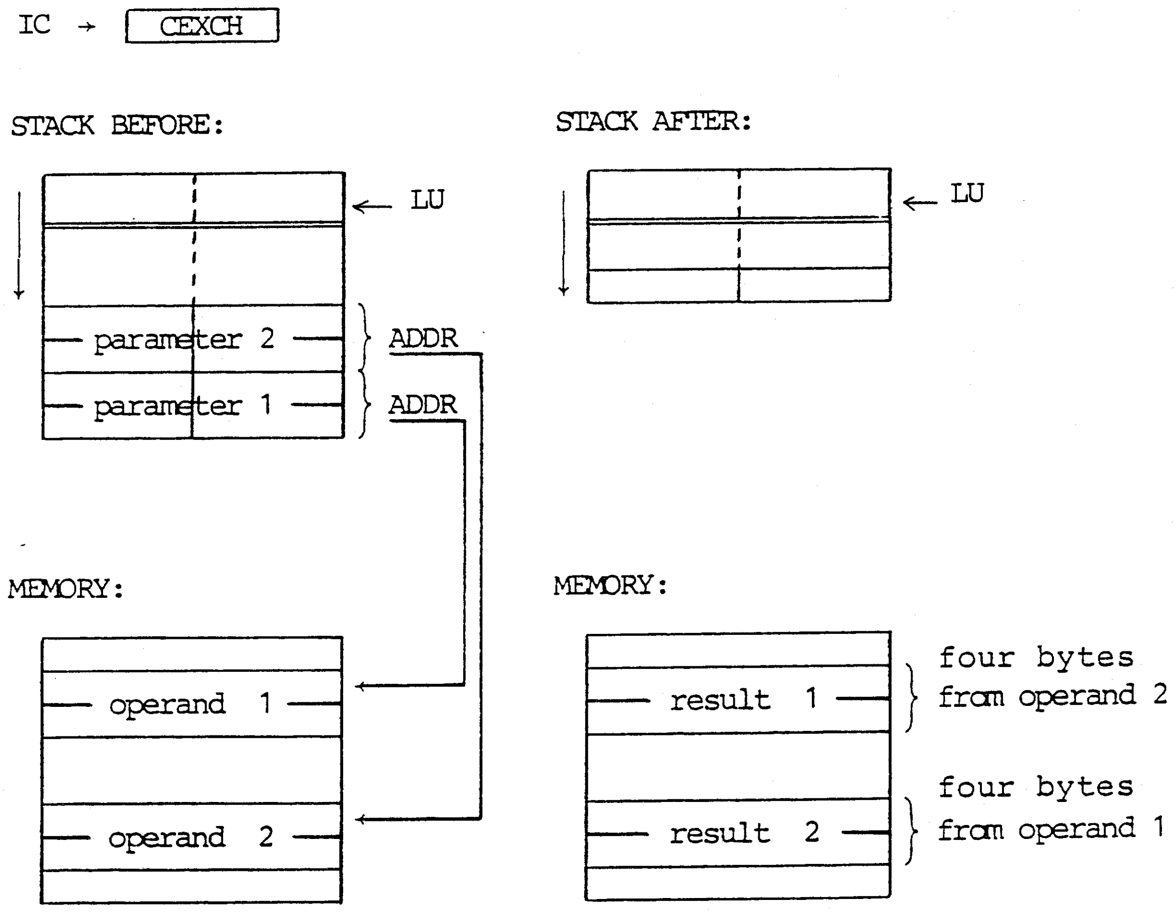
```
VAR
  refadr, semadr: adr;
  ref, sem: double;
  dummycontrol: word;
BEGIN
  getsemref( false, refadr, semadr, sem, dummycontrol);
  IF refadr.base.nilbit=1 THEN exception(nilpointer);
  ref.w1:=mem(refadr);
  IF NOT addrdouble(ref).base.nilbit=1 THEN
  exception(refnotnil);
  IF semstate(sem.w1)=open
  THEN wopen(false, refadr, semadr, sem, dummycontrol);
END;
```

11.3 Exchange Two Reference Variables 11.3

Two reference variables are exchanged indivisibly.

11.3.1 CEXCH 11.3.1

Control EXCHange Value: FE_{Hex}



```
(* CEXCH *)


cexch:
VAR
  adress1, adress2: adr;
  ref1,ref2: double;
BEGIN
  adress1.disp:= pop;
  adress1.base:= basetype(pop);
  adress2.disp:= pop;
  adress2.base:= basetype(pop);
  IF (adress1.base.nilbit=1) OR (adress2.base.nilbit=1)
    THEN exception(nilpointer);
  ref1:= memdouble(adress1);
  ref2:= memdouble(adress2);
  IF addr(ref1).nullbyte <> 0 OR addr(ref2).nullbyte <> 0
    THEN exception(reflocked);
  memdouble(adress1):=ref2;
  memdouble(adress2):=ref1;
END;
```

11.4 Control and Monitor Instructions 11.4

The instructions in this group are used to control process incarnations. The instructions except CSELL and MHALT are used by the runtime system exclusively.

The instruction CSELL is used to select a level > 0 or a certain priority at level = 0.

The instruction CSTART and CSTOP performs the indivisible parts of the monitors start and stop operations. The CSTOP instruction can involve the removing of a process incarnation from an arbitrary point of a semaphore chain.

MTIME is used to provide timeout service for process incarnations. This operation may involve the removing of a process incarnation from an arbitrary point of a semaphore chain too.

MRECHA is used to dump a process incarnation from a registerset to a memory registerset. This may involve a change of the semaphorechain containing the registerset.

CINWQ and COUTWQ inserts and removes memory registersets from the registerset waitqueue. The later must be used as a prefix to a waitinstruction.

MHALT permanently deschedules a process incarnation.

Common routines for unchain and rechain.

```
FUNCTION getpred(regset: regsetindex; memregset: adr): regsetindex;
BEGIN
  IF NOT inmem(regset)
  THEN getpred:=registerset(regindex(regset)).lm
  ELSE getpred:=mem(adroffset(adroffset(memregset,regset),lmoffset));
END;
```

```
FUNCTION getsucc(regset: regsetindex; memregset: adr): regsetindex;
BEGIN
  IF NOT inmem(regset)
  THEN getsucc:= registerset(regindex(regset)).lf
  ELSE getsucc:=mem(adroffset(adroffset(memregset,regset),lfoffset));
END;
```

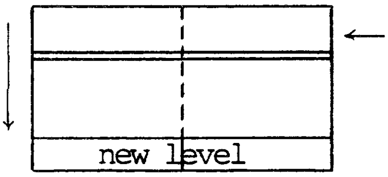
```
PROCEDURE putpred(regset,pred: regsetindex; memregset: adr);
BEGIN
  IF NOT inmem(regset)
  THEN registerset(regindex(regset)).lm:=pred
  ELSE mem(adroffset(adroffset(memregset,regset),lmoffset)):=pred;
END;


PROCEDURE putsucc(regset,succ: regsetindex; memregset: adr);
BEGIN
  IF NOT inmem(regset)
  THEN registerset(regindex(regset)).lf:= succ
  ELSE mem(adroffset(adroffset(memregset,regset),lfoffset)):= succ;
END;


PROCEDURE getsuccpred(regset: regsetindex;
                      VAR succ,pred: regsetindex;
                      VAR memregset: adr);
BEGIN
  memregset.base:= monitorreg.memregbase.base;
  memregset.disp:= monitorreg.memregdisp;
  pred:= getpred(regset,memregset);
  succ:= getsucc(regset,memregset);
END;


FUNCTION getsemadr(regset: regsetindex; memregset: adr): adr;
VAR
 gsadr: adr;
BEGIN
  IF NOT inmem(regset)
  THEN BEGIN
    gsadr.disp:=regset.pop;
    gsadr.base:=basetype(regset.pop);
    regset.pus:=word(gsadr.base);
    regset.pus:=gsadr.disp;
    getsemadr:=gsadr;
  END
  ELSE BEGIN
    gsadr.base:=
          basetype(mem(adroffset(adroffset(memregset,regset),sboffset)))
    gsadr.disp:=mem(adroffset(adroffset(memregset,regset),luoffset));
    getsemadr:=memadr(adroffset(gsadr,-4));
  END;
END;


FUNCTION getsem(regset: regsetindex; memregset: adr): word;
BEGIN
  getsem:= mem(getsemadr(regset,memregset));
END;


PROCEDURE reconnect( regset,succ,pred,
                     newsucc, newpred: regsetindex;
                     memregset: adr );
BEGIN
  IF nilreg(getsucc(pred,memregset))
  THEN BEGIN
    putsucc(pred,newsucc,memregset);
    IF nilreg(succ) THEN succ:=getsem(regset,memregset);
    putpred(pred,newpred,memregset);
  END
  ELSE BEGIN
    mem(getsemadr(regset,memregset)):=newsucc;
    IF nilreg(succ)
    THEN BEGIN
      IF succ=newsucc
      THEN RETURN
      ELSE succ:= newsucc
    END;
    putpred(succ,newpred,memregset);
  END;
END;


PROCEDURE rechain(regsetno, lregsetno: regsetindex);
VAR
  succ,pred: regsetindex;
  memregset: adr;
BEGIN
  getsuccpred(regsetno,succ,pred,memregset);
  putpred(lregsetno,pred,memregset);
  putsucc(lregsetno,succ,memregset);
  reconnect(regsetno,succ,pred,lregsetno,lregsetno,memregset);
END;


PROCEDURE unchain(regset: regsetindex);
VAR
  succ, pred: regsetindex;
  memregset: adr;
BEGIN
  getsuccpred(regset,succ,pred,memregset);
  reconnect(regset,succ,pred,succ,pred,memregset);
END;

FUNCTION insemaphore(ps: pstype; timer: integer): boolean;
BEGIN
  IF ps.pss = 0 THEN insemaphore:= false
  ELSE IF ps.mr = 1 THEN insemaphore:= false
  ELSE IF ps.pst = 0 THEN insemaphore:= true
  ELSE IF timer = 0 THEN insemaphore:= false
  ELSE insemaphore:= true;
END;
```

11.4.1 CSELL

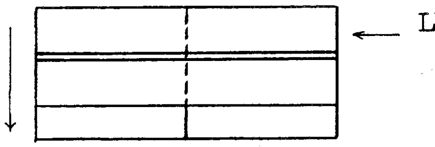
Control SELect Level Value: $1A_{Hex}$

IC → CSELL

STACK BEFORE:



STACK AFTER:



As a process incarnation executing on a level > 0 is scheduled by interrupts, the different combinations of level = 0, level > 0, new level = 0, and new level > 0 will cause different actions to be taken.

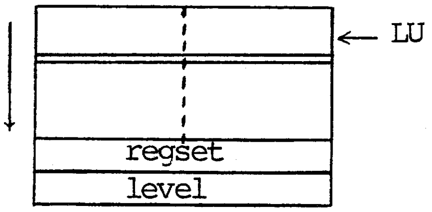
```
VAR
  newlevel, oldlevel : leveltype;
  incadr : adr;
BEGIN
  newlevel:= leveltype(pop);
  oldlevel:= ps.level;
  deactivatemap(oldlevel);
  activatemap(newlevel,context);
  ps.to:= 0;
  incadr.base:= sb.base;
  incadr.disp:= gf + leveloffset;
  membyte(incadr):=newlevel;
  ps.level:= newlevel;
END;
```

11.4.2 CSTART

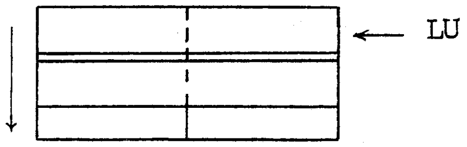
Control START   Value: $1C_{Hex}$

IC → CSTART

STACK BEFORE:



STACK AFTER:

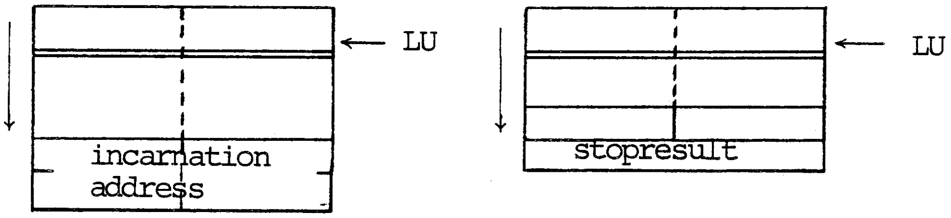


```
VAR
  level : leveltype
  regset : regsetindex;
BEGIN
  level:= pop;
  regset:= pop;
  activatemap(level,regset)
END;
```

Control STOP Value: $1B_{Hex}$

IC → CSTOP

STACK BEFORE: STACK AFTER:



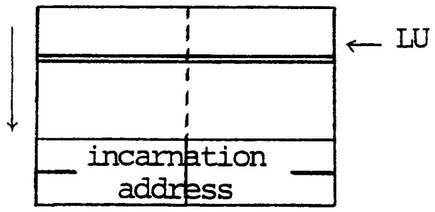
```
VAR
  incadr: adr;
  timer: integer;
  ta, mreg: bit;
  regset: regsetindex;
  stopstate: integer;
BEGIN
  incadr.disp:= pop;
  incadr.base:= basetype(pop);
  timer:= mem(incadr);
  IF timer=0
  THEN ta:= 1
  ELSE ta:= 0;
  regset:= mem(adroffset(incadr,regsetoffset));
  IF inmem(regset) THEN mreg:= 1 ELSE mreg:= 0;
  WITH registerset(regset) DO
  BEGIN
    stopstate:= ps.pst shift 5 +
                ps.pss shift 4 +
                ps.psi shift 3 +
                ps.mr  shift 2 +
                ta     shift 1 +
                mreg;
    IF mreg = 0 THEN
      IF ps.level0 THEN deactivate(ps.level)
                   ELSE intmap(ps.level):= dummyreg;
    IF insemaphore(ps,timer) THEN unchain(regset);
  END;
  pus:= stopstate;
END;
```

11.4.4 MTIME

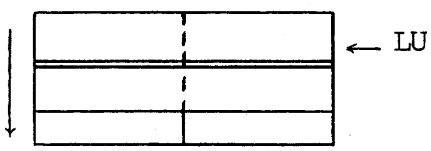
Monitor function TIME count down Value: 17_{Hex}

IC → $\boxed{\text{MTIME}}$

STACK BEFORE:



STACK AFTER:



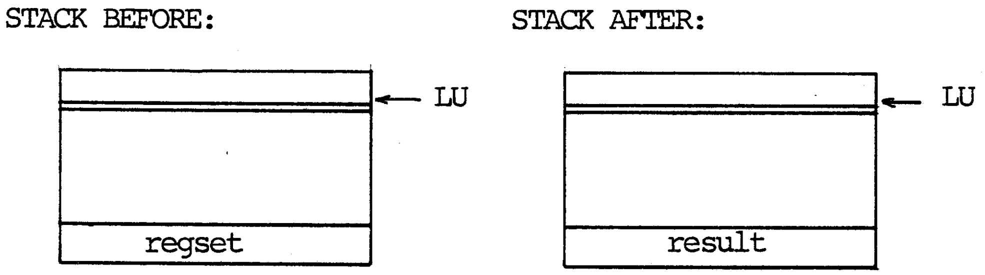
This instruction is used to provide timeout service for process incarnations requesting it.

```
VAR
  incadr: adr;
  timer: integer;
  psval: pstype;
  regset: regsetindex;
  memregset: adr;
BEGIN
  incadr.disp:= pop;
  incadr.base:= basetype(pop);
  WHILE NOT (incadr.base.nilbit=1) AND NOT interrupt DO
  BEGIN
    timer:= mem(incadr);
    IF timer > 0
    THEN BEGIN
      mem(incadr):= timer - 1;
      IF timer - 1 = 0
      THEN BEGIN
        regset:= mem(adroffset(incadr,regsetoffset));
        IF inmem(regset)
        THEN BEGIN
          memregset.base:= monitorreg.memregbase.base;
          memregset.disp:= monitorreg.memregdisp + regset);
          psval:= pstype(mem(memregset));
        END
        ELSE psval:= registerset(regindex(regset)).ps
        IF (ps.pst=1) OR ((ps.pss=0) AND (ps.psi=0))
        THEN BEGIN
          IF insemaphore(ps,timer-1) THEN unchain(regset);
          IF inmem(regset)
          THEN BEGIN
            IF wqempty(monitorreg.last)
            THEN monitorreg.first:= memregset.disp
            ELSE mem(adroffset(memregset,lfoffset)):=
                                   memregset.disp;
            monitorreg.last:= memregset.disp;
            device(monitorreg.monitorlevel).interrupt:= 1;
          END
          ELSE activate(psval.level,regset)
        END
      END;
           END;
           incadr:= memadr(adroffset(incadr,tchainoffset));
         END;
         IF interrupt
         THEN BEGIN
           pus:= incadr.disp;
           pus:= word(incadr.base);
           GOTO repeatinstruction;
         END;
       END;
```

11.4.5 MRECHA 11.4.5

Monitor RECHAin Value: 81_{Hex}

IC → MRECHA

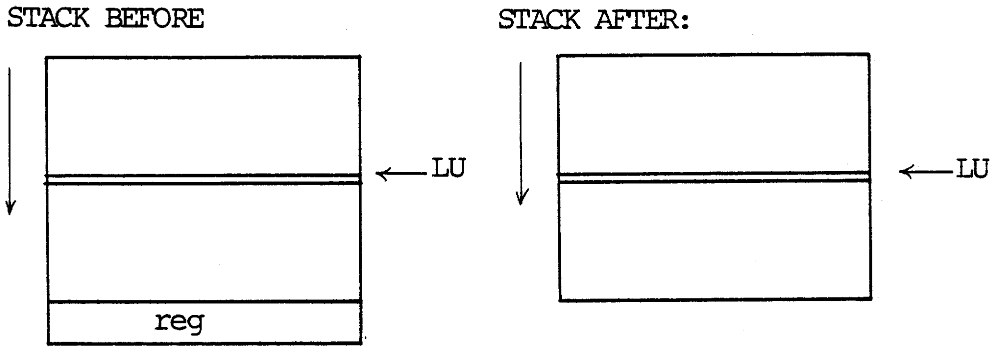


```
VAR
  regset: regsetindex;
  timer: integer;
  memregset: adr;
  incadr: adr;
  mregset: word;
BEGIN
  regset:= pop;
  WITH registerset(regindex(regset)) DO
  IF NOT ps.level0 THEN pus:= -1
  ELSE BEGIN
    IF ps.exit THEN pus:= 0
    ELSE BEGIN
      IF (ps.pss=0) AND (ps.pst=0) AND (ps.psi=0) THEN pus:= -2
      ELSE BEGIN
        IF (ps.pss=1) AND (ps.mr=1) THEN pus:=-2
        ELSE BEGIN
          incadr.base:= sb.base;
          incadr.disp:= gf;
          IF (ps.pst=1) AND (mem(incadr)=0) THEN pus:= -2
          ELSE BEGIN
            mregset:= mem(adroffset(incadr,mregsetoffset));
            IF nilreg(mregset) THEN pus:=-3
            ELSE BEGIN
              mem(adroffset(incadr, regsetoffset)):=mregset;
              memregset.base:= monitorreg.memregbase.base;
              memregset.disp:= monitorreg.memregdisp;
              mem(memregset):= word(ps);
              IF ps.pss=1
              THEN BEGIN
                rechain(regset,mregset);
                pus:= 2;
              END
              ELSE pus:= 1;
            END;
          END;
        END;
      END;
    END;
  END;
END;
```

11.4.6 CINWQ

Control INto WaitQueue Value: 13_{Hex}

$IC \longrightarrow \boxed{\text{CINWQ}}$



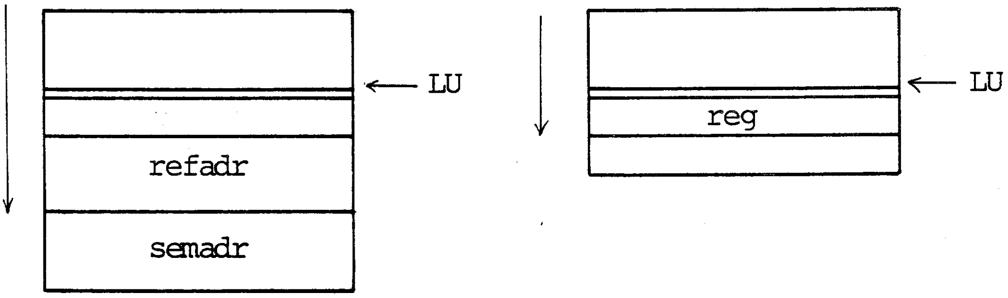
```
VAR
  reg,last: word;
  memregset: adr;

BEGIN
  reg:= pop;
  last:= monitorreg.last.
  memregset.base:= monitorreg.memregbase.base;
  IF nilreg(last)
  THEN monitorreg.first:= reg
  ELSE BEGIN
    memregset.disp:= monitorreg.last;
    mem(adroffset(memregset,lfoffset)):= reg;
  END;
  monitorreg.last:= reg;
END;
```

11.4.7 COUTWQ

Control OUT WaitQueue   Value: 14_{Hex}

IC → COUTWQ

STACK BEFORE: STACK AFTER:



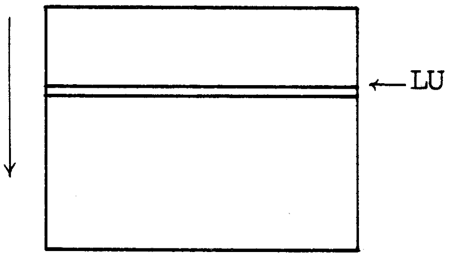
```
VAR
  first,last,i: word;
  memregset: adr;
BEGIN
  last:= monitorreg.last;
  IF nilreg(last) THEN GOTO nextinstruction
  ELSE BEGIN
    first:= monitorreg.first;
    IF last = first
    THEN monitorreg.last:= 1 (* nil *)
    ELSE BEGIN
      memregset.base:= monitorreg.memregbase.base;
      memregset.disp:= first;
      first:= mem(adroffset(memregset,lfoffset));
    END
    FOR i:= 1 TO 4 DO pop;
    nextbyte;
    pus:= first;
  END;
END;
```

11.4.8 MHALT

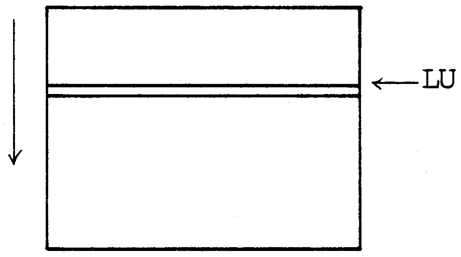
Monitor HALT   Value: 18_{Hex}

IC ⟶ $\boxed{\text{MHALT}}$

STACK BEFORE:



STACK AFTER:



```
VAR

BEGIN
  deactivate(ps.level);
  incadr.base:= sb.base;
  incadr.disp:= gf + leveloffset;
  ps.level0:= 1;
  ps.prio:= 0;
  ps.exit:= true;
  membyte(incadr):= byte(ps.level);
END;
```

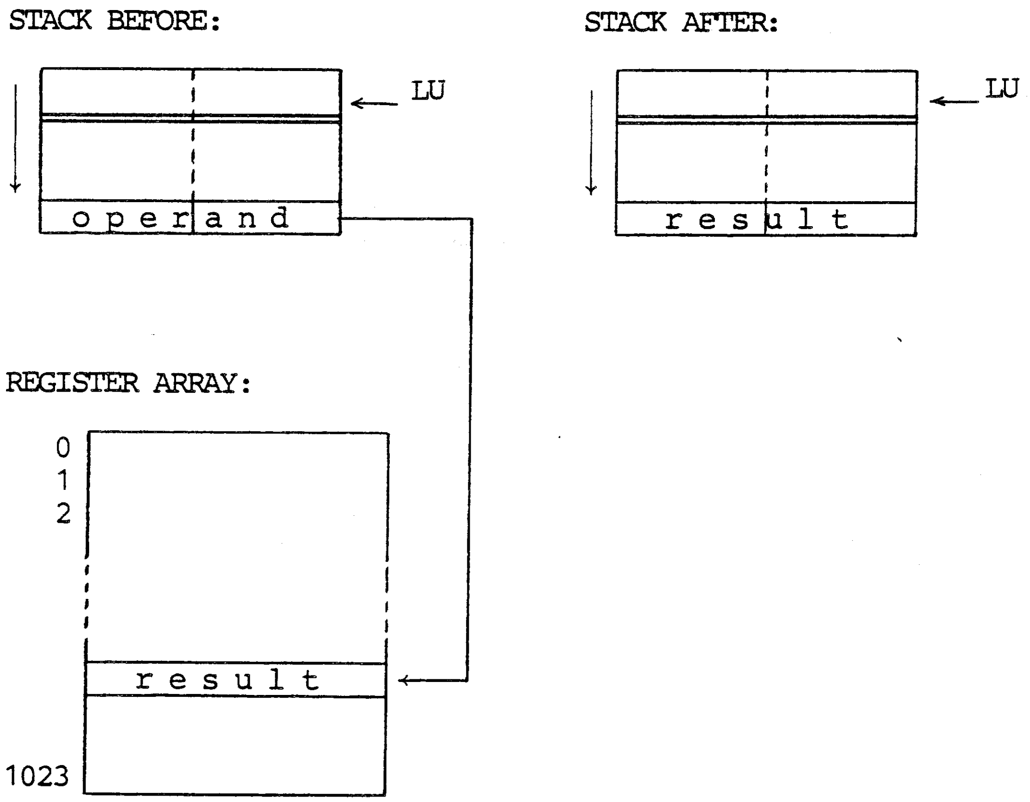
11.5 Register Array Operations 11.5

The instructions in this group are used to manipulate the register array (section 3.3) and the RAM memory of the control microprocessor (subsection 3.12.1).

11.5.1 CRGET 11.5.1

Control Register GET Value: 91_{Hex}

IC → CRGET



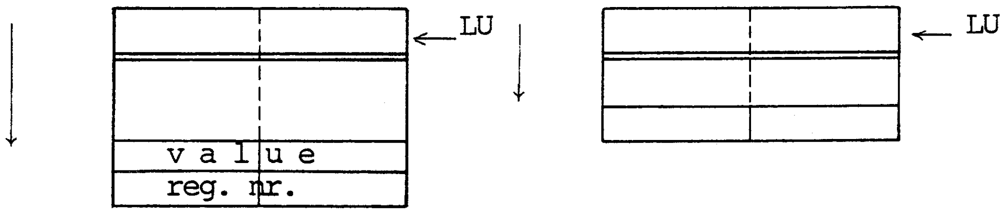
```
VAR
  regaddr: integer;
BEGIN
  regaddr:= pop;
  pus:= registers(regaddr);
END;
```

11.5.2 CRPUT

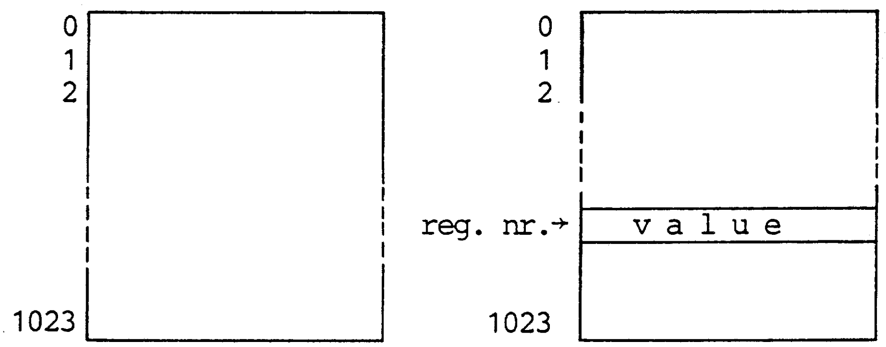
Control Register PUT   Value: $B1_{Hex}$

IC → CRPUT

STACK BEFORE: STACK AFTER:



REGISTER ARRAY BEFORE: REGISTER ARRAY AFTER:

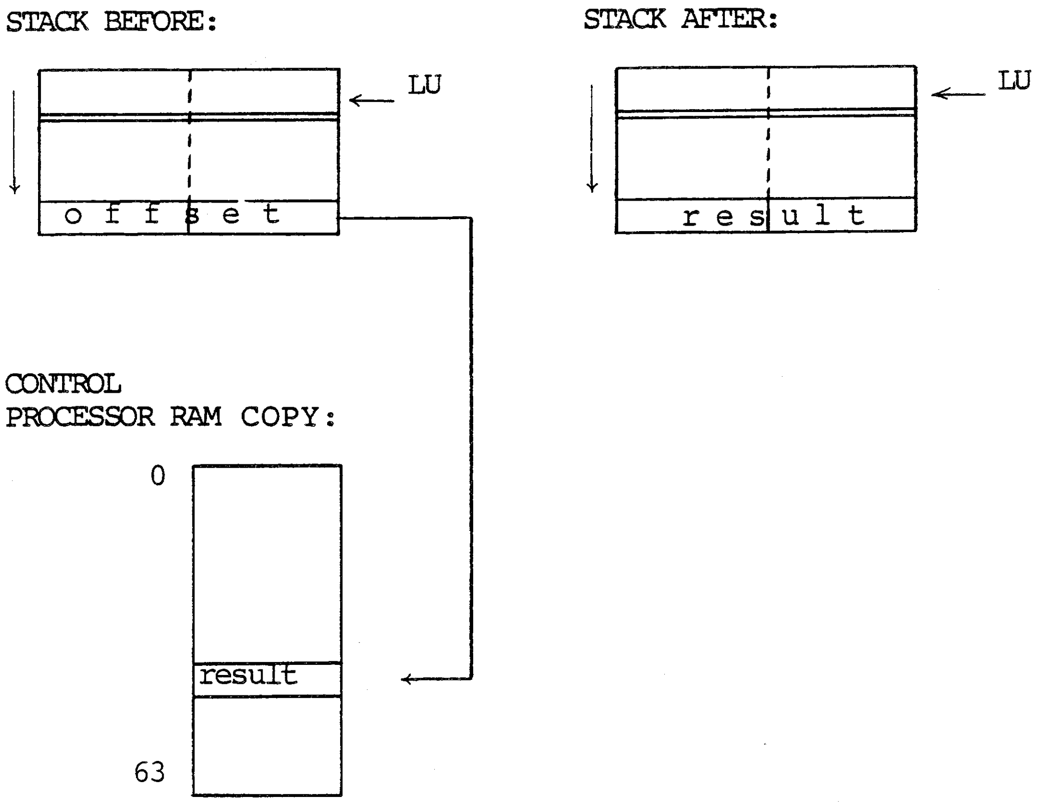


```
VAR
  regaddr: integer;
BEGIN
  regaddr:= pop;
  registers(regaddr):= pop;
END;
```

11.5.3 CRRAM 11.5.3

Control Read RAM of Control Processor Value: DE_{Hex}

IC → CRRAM

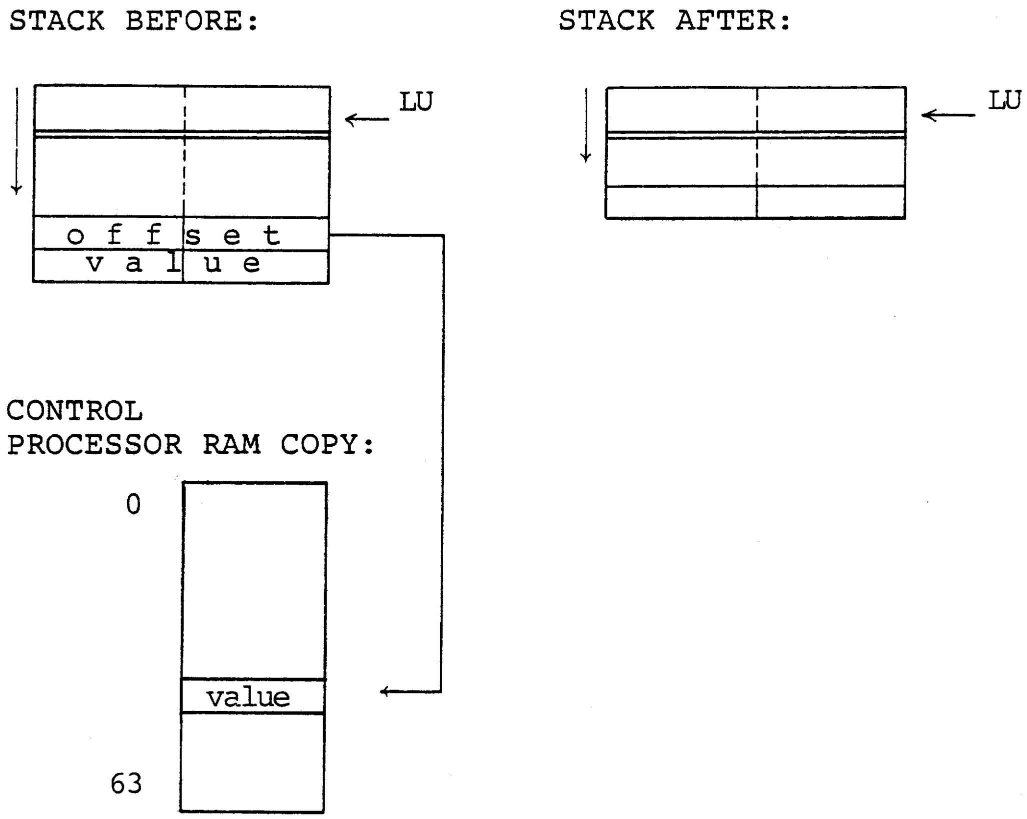


```
VAR
  cbyteadr: adr;
BEGIN
  cbyteadr.base:= basetype(#hCO);
  cbyteadr.disp:= pop AND #h3f;
  pus:= word(membyte(cbyteadr));
END;
```

11.5.4 CWRAM

Control Write RAM of Control Processor   Value: DF_{Hex}

IC → CWRAM



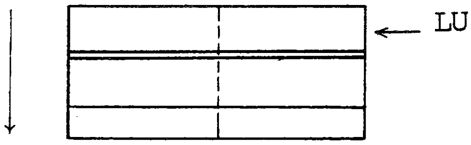
```
VAR
  addr : addr;
  val : word;
  cbyteadr : adr;
BEGIN
  cbyteadr.base:= basetype(#hC0);
  val:= pop;
  cbyteadr.disp:= pop AND #h3f;
  IF com8085.cow = 0
  THEN BEGIN
    com8085.cow:= val*256 + 128 + cbyteadr.disp;
    membyte(cbyteadr):= val;
    set8085interrupt;
  END
  ELSE BEGIN
    pus:= cbyteadr.disp;
    pus:= val;
    GOTO resumeinstruction;
  END;
END;
```

11.5.5 CGREG

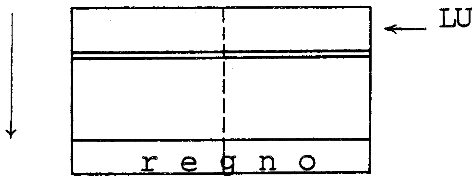
Control Get REGister   Value: 1F$_{Hex}$

IC → CGREG

STACK BEFORE:



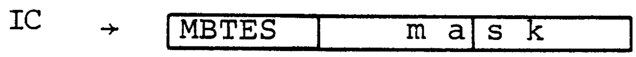
STACK AFTER:



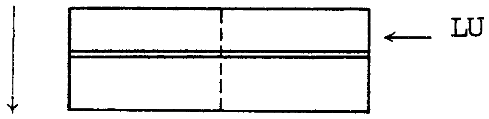
```
BEGIN
  pus:= context ;
END;
```

11.5.6 MBTES 11.5.6

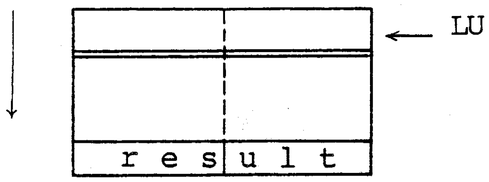
Miscellaneous Bits TESt in Status Register Value: $D1_{Hex}$



STACK BEFORE:



STACK AFTER:

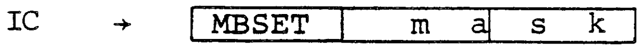


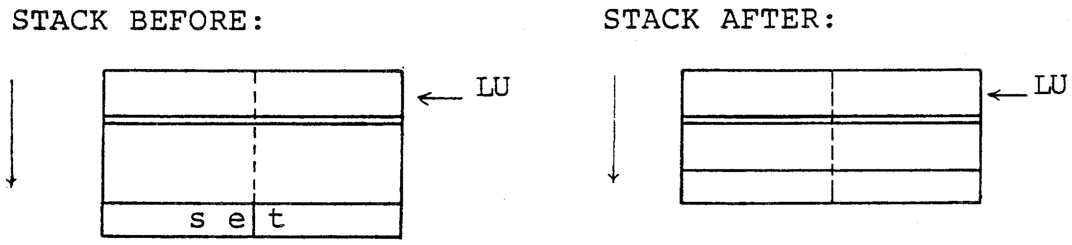
```
VAR
  mask: word;
BEGIN
  mask:= nextword;
  WITH ps AS word DO
  IF (ps AND mask) = 0
    THEN pus:= false
    ELSE pus:= true
END;
```

11.5.7 MBSET

Miscellaneous Bits SET or Clear in Status Register

D0$_{Hex}$





```
VAR
  mask,set: word;
BEGIN
  mask:= nextword;
  set:= pop;
  WITH ps AS word DO
    IF set = 0
    THEN ps:= ps AND NOT mask
    ELSE ps:= ps OR mask
END;
```

11.6 Push and Pop 11.6

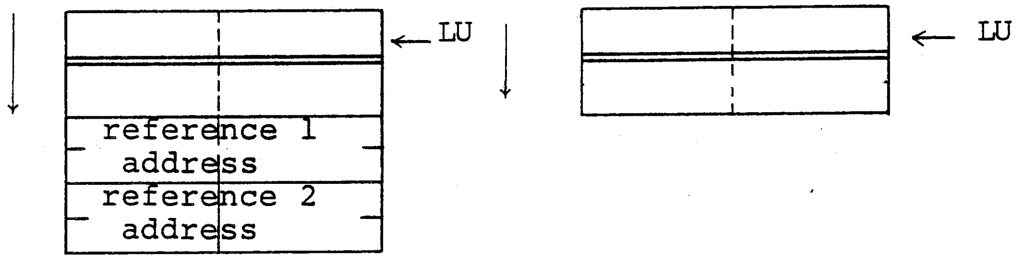
The instructions in this group are used for manipulation of message stacks, i.e. they support the language constructs push and pop.

11.6.1 LPUSH 11.6.1

Language Support PUSH Value: $7C_{Hex}$

IC → $\boxed{\text{LPUSH}}$

STACK BEFORE: STACK AFTER:



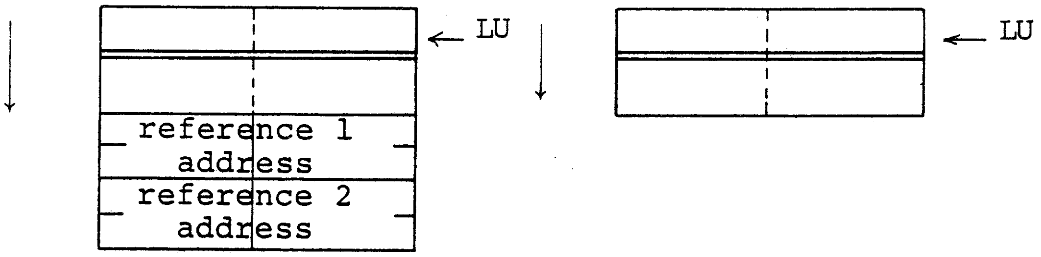
```
VAR
  ref1adr, ref2adr: adr;
  ref1, ref2: addr;
  r1stackadr: adr;
  msg1kind: word;
BEGIN
  ref2adr.disp:= pop;
  ref2adr.base:= basetype(pop);
  ref1adr.disp:= pop;
  ref1adr.base:= basetype(pop);
  IF ref1adr.base.nilbit=1 THEN exception(nilpointer);
  IF ref2adr.base.nilbit=1 THEN exception(nilpointer);
  ref1:= memaddr(ref1adr);
  IF ref1.base.nilbit=1 THEN exception(nilreference);
  IF ref1.nullbyte<>0 THEN exception(reflocked);
  r1stack:= memadr(addroffset(ref1,stackoffset));
  IF r1stack.nilbit=1 THEN exception(ref1stacked);
  ref2:= memaddr(ref2adr);
  IF ref1 = ref2 THEN exception(r1eqr2);
  IF (ref2.base.nilbit=0) AND (ref2.nullbyte<>0)
  THEN exception(reflocked);
  memadr(ref1adr).base.nilbit := 1;
  memaddr(addroffset(ref1,stackoffset)):= ref2;
  memaddr(ref2adr):= ref1;
  msg1kind:= mem(addoffset(ref1,kindoffset));
  IF (ref2.base.nilbit=0) AND (msg1kind=headerkind) THEN
  BEGIN
    memadr(addroffset(ref1,startoffset)):=
                         memadr(addroffset(ref2,startoffset));
    mem(addroffset(ref1,sizeoffset)):= mem(addroffset(ref2,sizeoffset));
  END;
END;
```

11.6.2 LPOP

Language Support POP Value: $7D_{Hex}$

IC → $\boxed{\text{LPOP}}$

STACK BEFORE: STACK AFTER:



```
VAR
  ref1adr, ref2adr: adr;
  ref1, ref2: addr;
  ref2stack: adr;
  msg2kind: word;
BEGIN
  ref2adr.disp:= pop;
  ref2adr.base:= basetype(pop);
  ref1adr.disp:= pop;
  ref1adr.base:= basetype(pop);
  IF ref1adr.base.nilbit=1 THEN exception(nilpointer);
  IF ref2adr.base.nilbit=1 THEN exception(nilpointer);
  ref1:= memaddr(ref1adr);
  ref2:= memaddr(ref2adr);
  IF ref1.base.nilbit=0 THEN exception(refnotnil);
  IF ref2.base.nilbit=1 THEN exception(nilreference);
  IF ref2.nullbyte<>0 THEN exception(lockexception);
  ref2stack:=memadr(addroffset(ref2,stackoffset));
  memaddr(ref1adr):= ref2;
  memadr(adroffset(ref2adr,1)):= ref2stack;
  memadr(addroffset(ref2,stackoffset)).base.nilbit:= 1;
  msg2kind:= mem(addroffset(ref2,kindoffset));
  IF msg2kind=headerkind
  THEN BEGIN
    memword(addroffset(ref2,sizeoffset)):= 0;
    memadr(addroffset(ref2,startoffset)).base.nilbit:= 1;
  END;
END;
```

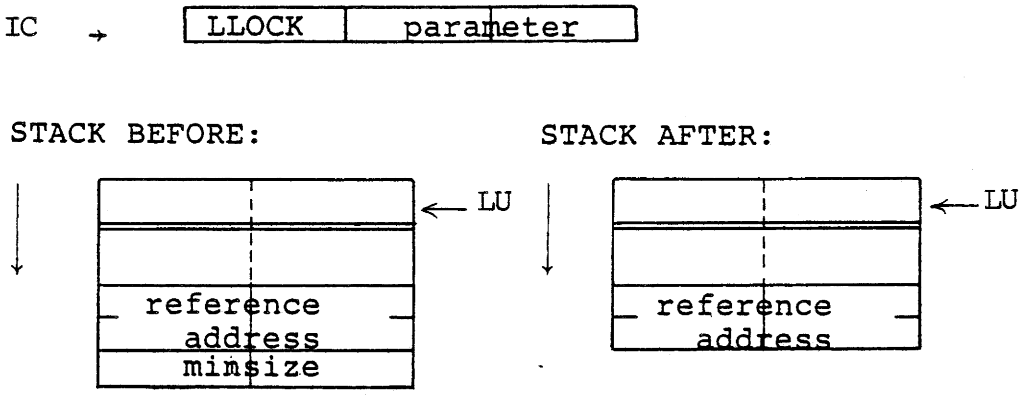
11.7 Lock Support

The instructions in this group support the language construct: lock statement.

```
procedure getlock(var lockcount: byte; VAR lockadr: adr);
begin
  lockadr.disp:= pop;
  lockadr.base:= basetype(pop);
  if lockadr.base.nilbit=1 then exception(nilpointer);
  lockcount:= membyte(lockadr);
end;
```

11.7.1 LLOCK

Language Support LOCK Type on Message   Value: $7F_{Hex}$

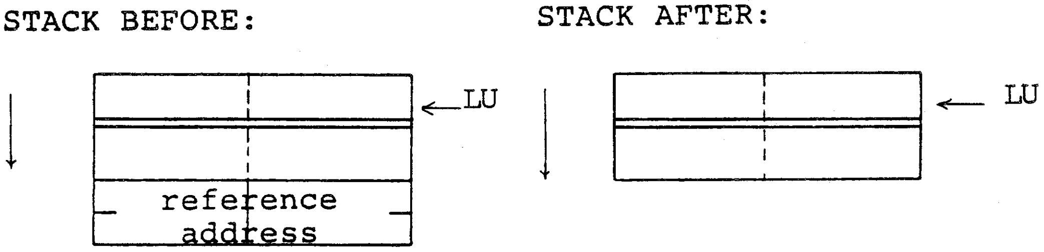


```
VAR
  minsize, ldisp: word;
  ref: addr;
  refadr, stackadr: adr
BEGIN
  ldisp:= nextword;
  minsize:= pop;
  refadr.disp:= pop;
  refadr.base:= basetype(pop);
  IF refadr.base.nilbit=1 THEN exception(nilpointer);
  ref:= memaddr(refadr);
  IF ref.base.nilbit=1 THEN exception(nilreference);
  IF mem(addroffset(ref,kindoffset)) < 0 THEN exception(locktype);
  IF mem(addroffset(ref,sizeoffset)) < minsize THEN exception(locksize);
  stackadr:= memadr(addroffset(ref,startoffset));
  stackbyte(lf+ldisp):= byte(stackadr.base);
  stackword(lf+ldisp+1):= stackadr.disp;
  pus:= word(refadr.base);
  pus:= refadr.disp;
        END;
```

11.7.2 LRESE

Language Support RESErve Reference Value: $7E_{Hex}$

IC → LRESE



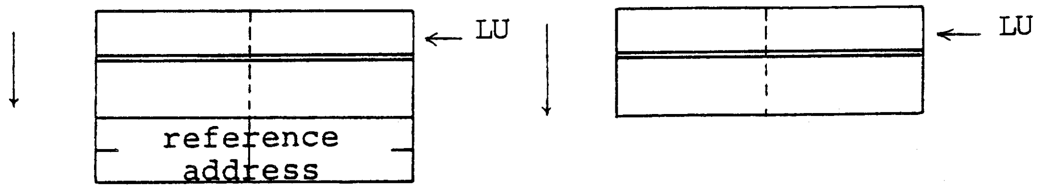
(* LRESE, reserve reference, i.e. increase lockcount *)

```
VAR
  lockcount : integer
  lockadr : adr;
BEGIN
  getlock(lockcount,lockadr);
  IF lockcount=255 THEN exception(lockoverflow);
  membyte(lockadr):=lockcount+1;
END;
```

11.7.3 LRELE

Language Support RELEase Reference   Value: 16_{Hex}

IC → LRELE



STACK BEFORE:

STACK AFTER:

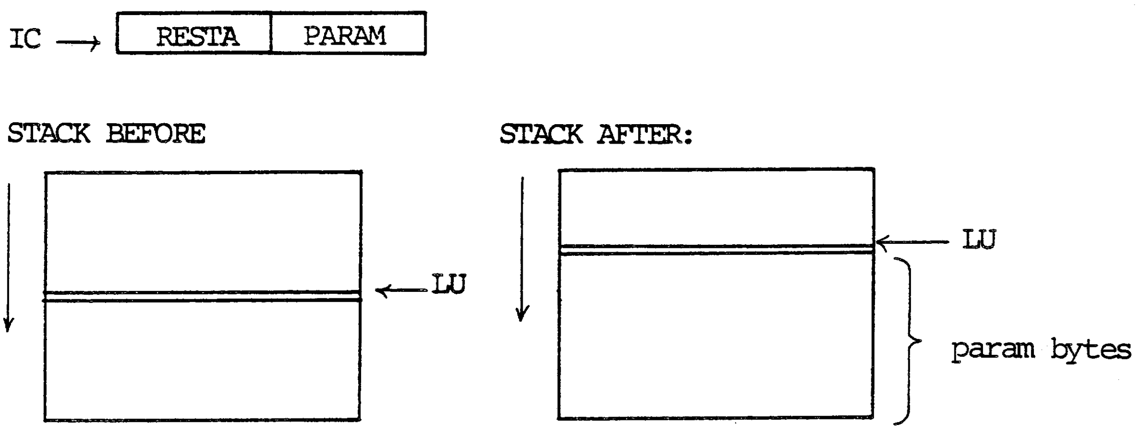
```
(* LRELE, release reference, decrease lockcount *)

VAR
  lockcount : integer;
  lockadr : adr;
BEGIN
  getlock(lockcount,lockadr);
  IF lockcount=0 THEN exception(lockoverflow);
  membyte(lockadr):=lockcount-1;
END;
```

11.8 Register Stack Adjust 11.8

These instructions is used to move the register stack to and/or from memory.

11.8.1 RESTA 11.8.1

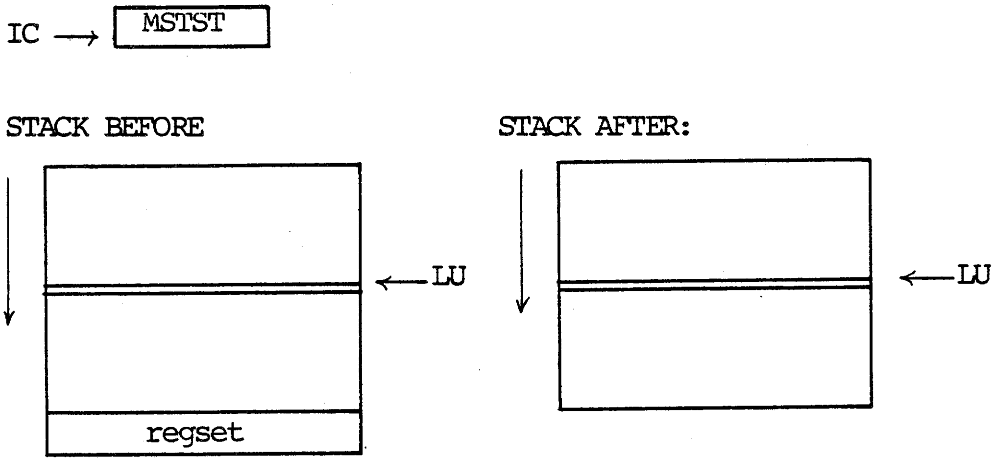
REgister STack Adjust Value: 95_{Hex}



```
VAR
  count, i: integer;
BEGIN
  checkdumpstack(stacksize)
  count:= nextbyte;
  lu:= lu - count
  FOR i:= 1 TO count DIV 2 DO
    pus:= stack(lu - 1 + 2*i);
END;
```

11.8.2 MSTST

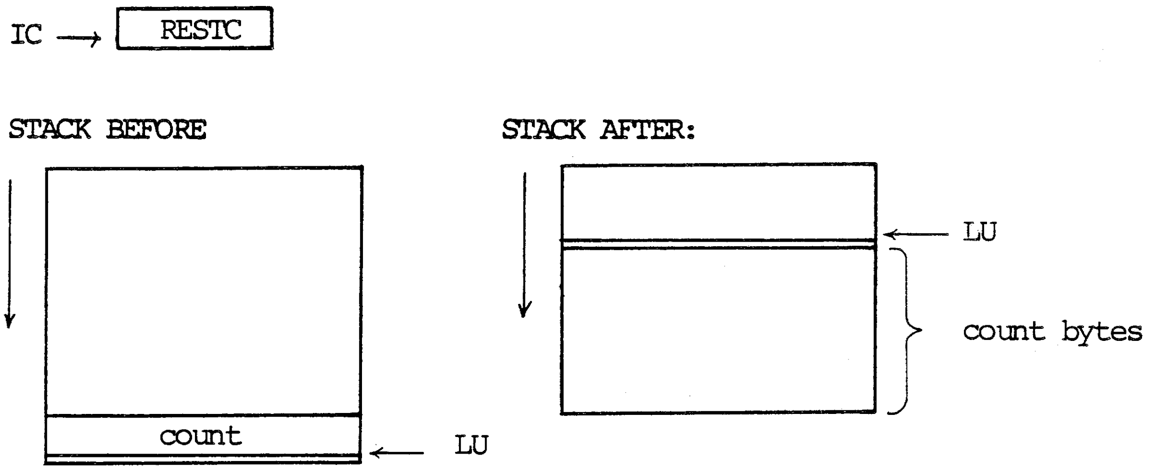
Monitor STore STack Value: 29_{Hex}



```
VAR
  regset: word;
  count, i: integer;
  stackadr: adr;
BEGIN
  WITH registerset(regindex(pop)) DO
  BEGIN
    stackadr.base:= sb.base;
    stackadr.disp:= lu;
    count:= stacksize;
    FOR i:= count DIV 2 DOWNTO 1 DO
      mem(adroffset(stackadr,-1+2*i)):= pop;
    mem(adroffset(stackadr,1+count)):= count;
    lu:= lu + count + 2;
  END;
END;
```

11.8.3 RESTC 11.8.3

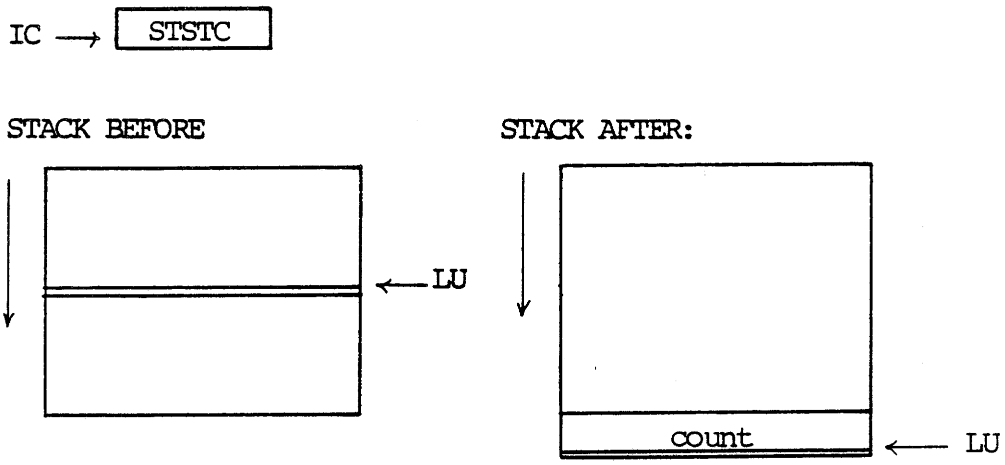
REtrive STack with Count Value: $D5_{Hex}$



```
VAR
  count, i: integer;
BEGIN
  count:= stack(lu-1);
  FOR i:= count DIV 2 DOWNTO 1 DO
    pus:= stack(lu - 1 - 2*i);
  lu:= lu - count - 2;
  stackbyte(gf + instcodeoffset):= 0;
END;
```

11.8.4 STSTC

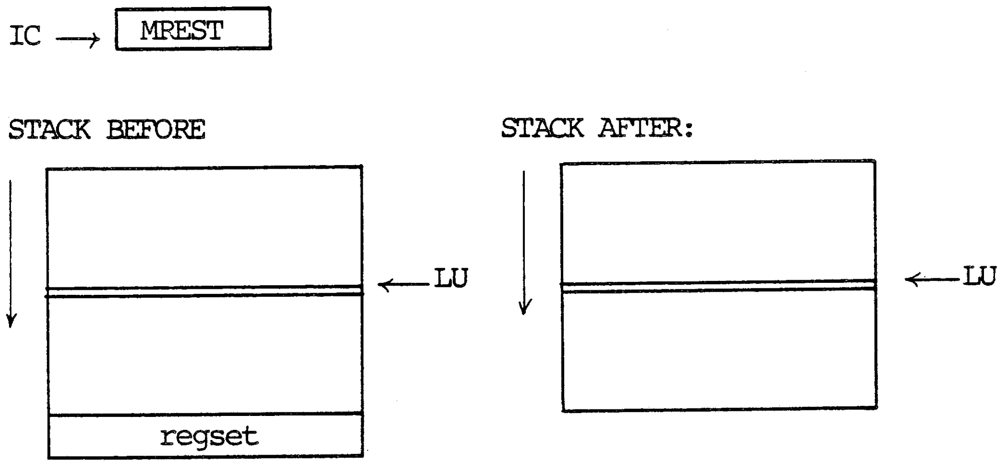
STore STack with Count   Value: D4$_{\text{Hex}}$



```
VAR
  count, i: integer;
BEGIN
  count:= stacksize;
  FOR i:= count DIV 2 DOWNTO 1 DO
    stack(lu - 1 + 2*i):= pop;
  stack(lu + count + 1):= count;
  lu:= lu + count + 2;
END;
```

11.8.5 MREST

Monitor REtrieve STack Value: 28_{Hex}



```
VAR
  regset: word;
  count, i: integer;
  stackadr: adr;
BEGIN
  WITH registerset(regindex(pop)) DO
  BEGIN
    stackadr.base:= sb.base;
    stackadr.disp:= lu;
    count:= mem(adroffset(stackadr, - 1));
    FOR i:= count DIV 2 DOWNTO 1 DO
      pus:= mem(adroffset(stackadr, - 1 - 2*i));
    lu:= lu - count - 2;
    stackadr.disp:= gf;
    membyte(adroffset(stackadr,instcodeoffset)):= 0;
  END;
END;
```

12. INDEXING AN ARRAY 12.

The following routine is used to test that a value lies within a given subrange:

```
FUNCTION checkrange(VAR dopevector: adr;
                    VAR index,  normindex: integer): boolean;
VAR
  lower, upper: integer;
BEGIN
  lower:= mem(dopevector);
  upper:= mem(adroffset(dopevector,2));
  IF (index < lower) OR (upper < index)
  THEN checkrange:= false
  ELSE BEGIN
    checkrange:=true;
    normindex:=index-lower;
  END;
  dopevector.disp:= dopevector.disp+4;
END;
```

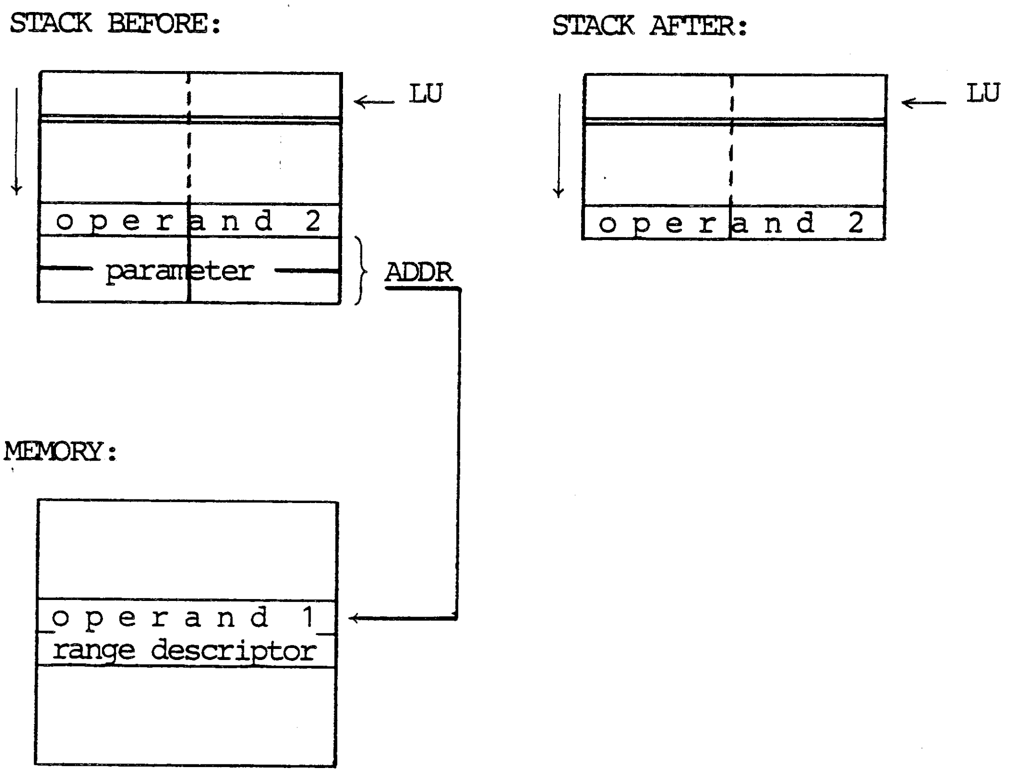
12.1 Range Test

These instructions test that a value lies between a given subrange. The value is not changed.

12.1.1 INTRS

INdex Test Range via Stack Value: $6C_{Hex}$

IC → INTRS

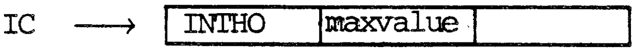


A range descriptor is an object of type DOUBLE. The first WORD is interpreted as a signed integer, and specifies the lower bound of the range. The second WORD is also interpreted as a signed integer, and specifies the upper bound of the range. The first byte of a range descriptor must be on a word boundary.

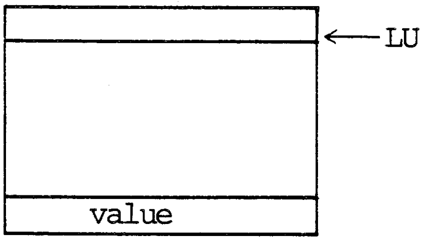
```
VAR
  index: integer
  nindex: integer;
  dopeadress: adr;
BEGIN
  dopeadress.disp:= pop;
  dopeadress.base:= basetype(pop);
  index:= pop;
  IF checkrange(dopeadress,index,nindex)
  THEN pus:=index
  ELSE exception(indexexception);
END;
```

12.1.2 INTHO

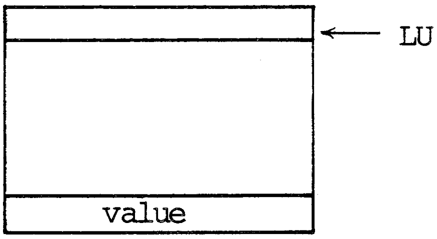
INdex Test Here 0 Value: C1_{Hex}



STACK BEFORE:



STACK AFTER:



```
VAR
  index: integer;
BEGIN
  index:= pop;
  IF ( index < 0 ) OR ( index > nextword )
    THEN exception(indexexception);
  pus:= index;
END;
```

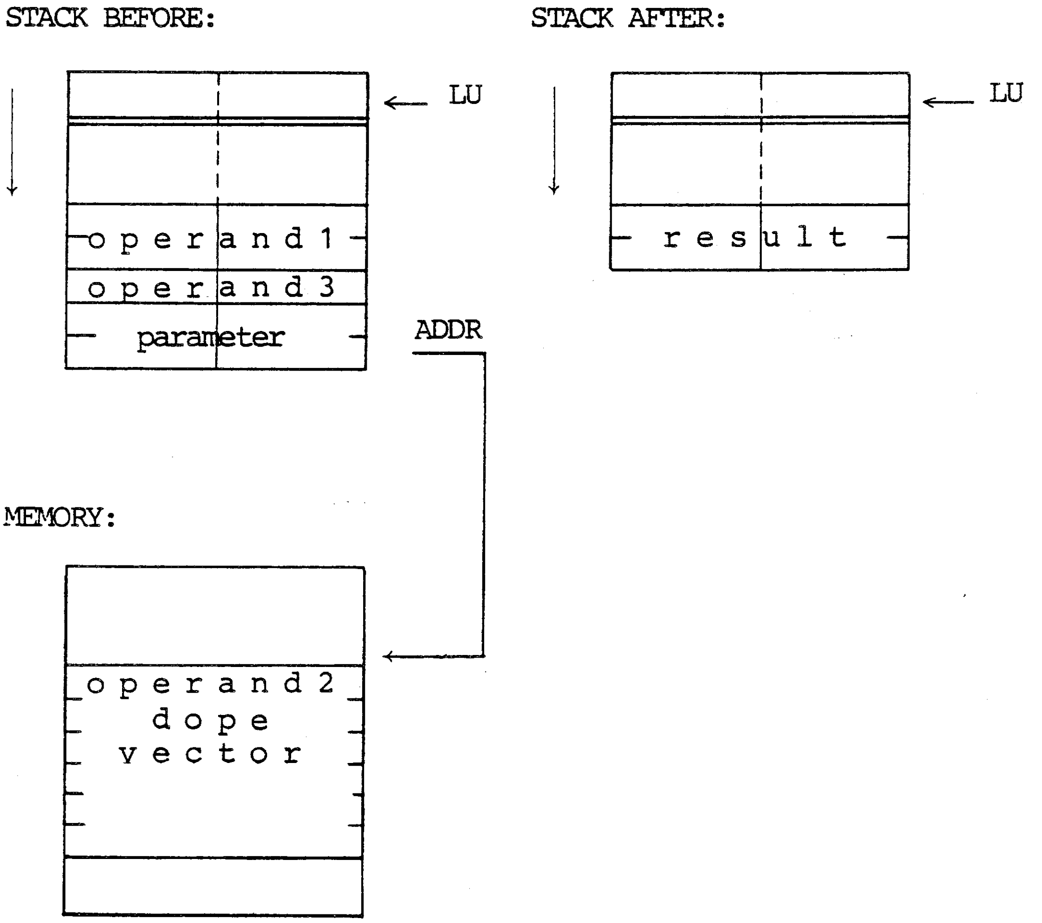
12.2 Indexing an Array

These instructions check that an array-index lies in a given range described by the first and last indexvalue and calculates the address of the array element.

INDEX

Value: $6D_{Hex}$

IC → INDEX



A dope vector is an object which describes a one-dimensional array. The object consists of a DOUBLE, which is a range descriptor for the index type, followed by a WORD, which specifies the number of bytes occupied by each element of the array. The first byte in a dope vector must be on a word boundary.

```
VAR
  index,nindex : integer;
  dopeadress : adr;
  size : integer;
  operand : adr;
BEGIN
  dopeadress.disp:=pop;
  dopeadress.base:=basetype(pop);
  index:=pop;
  IF checkrange(dopeadress,index,nindex)
  THEN BEGIN
    size:=  mem(dopeadress);
    operand.disp:=pop;
    pus:=operand.disp +  size*nindex;
  END
  ELSE exception(indexexception);
END;
```

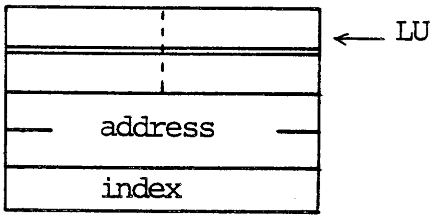
12.2.2 INDHO

INDex Here 0 Value: 38_{Hex}

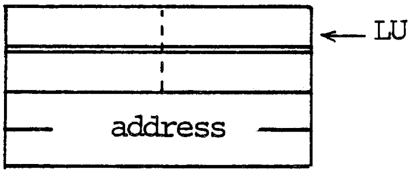
IC ⟶

INDHO	maxvalue	length

STACK BEFORE:



STACK AFTER:



```
VAR
  index: integer;
  operand: adr;
BEGIN
  index:=pop;
  IF (index < 0) OR (index > nextword)
  THEN exception(indexexception);
  operand.disp:=pop;
  pus:=operand.disp + index*nextword;
END;
```

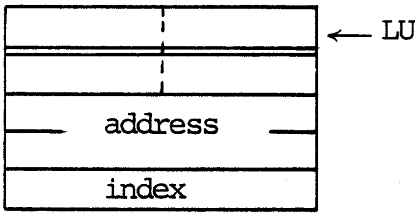
12.2.3 INDH1

INDex Here 1   Value: 39_{Hex}

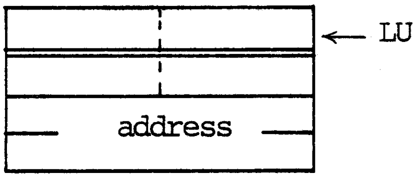
IC →

INDHI	maxvalue	length

STACK BEFORE:



STACK AFTER:



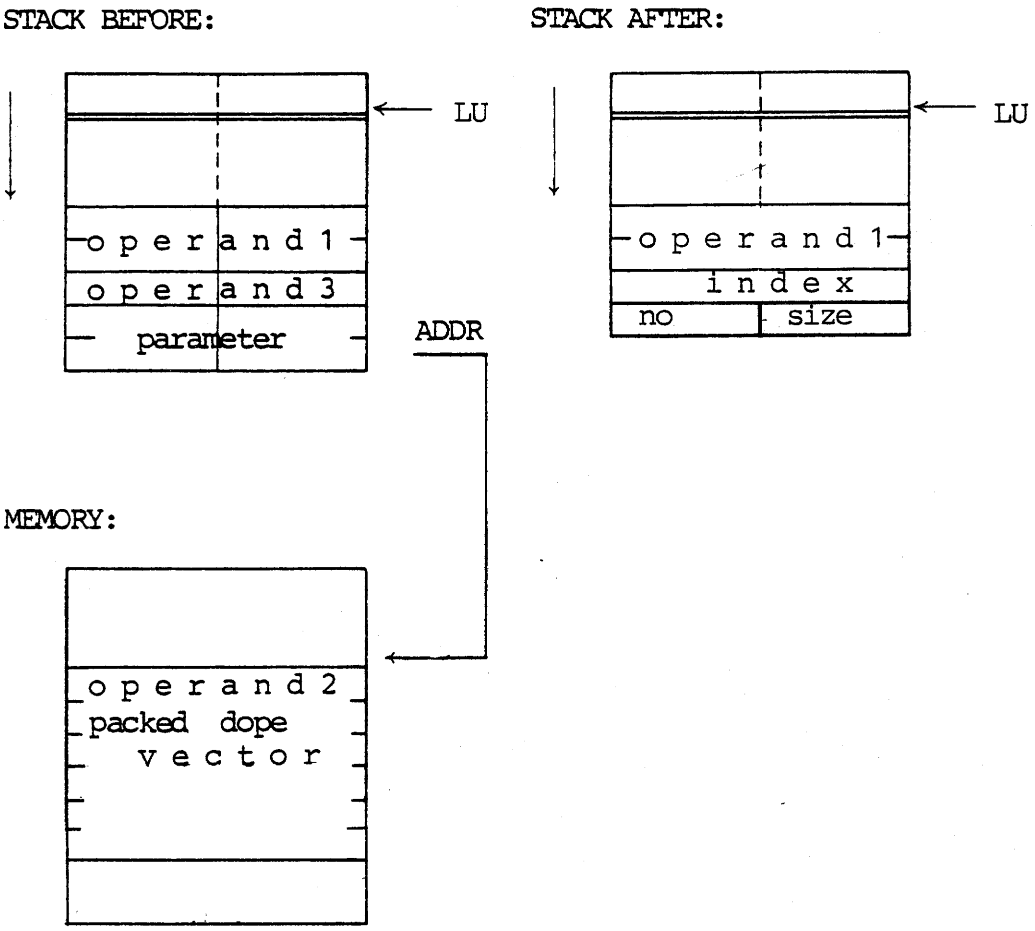
```
VAR
  index: integer;
  operand: adr;
BEGIN
  index:=pop;
  IF (index < 1) OR (index > nextword)
    THEN exception(indexexception);
  operand.disp:=pop;
  pus:=operand.disp + (index-1)*nextword;
END;
```

12.3 Push an Element of a Packed Array

These instructions check that an array-index lies in a given range and calculates the address of the word containing a field in a packed array and a field descriptor. To minimize interrupt disable time the operation is partitioned into two instructions. INTPA retrieves the dopevector and tests the indexvalue. INPDV calculates the address and field descriptor.

INdex Test Packed Array Value: $6E_{Hex}$

IC → INTPA



A packed dope vector is an object which describes a one-dimensional packed array. The object consists of a DOUBLE, which is a range descriptor for the index type, followed by two BYTEs; the first is the number of array elements packed in a single WORD, and the second is the size of one element in bits. The first byte in a packed dope vector must be on a word boundary.

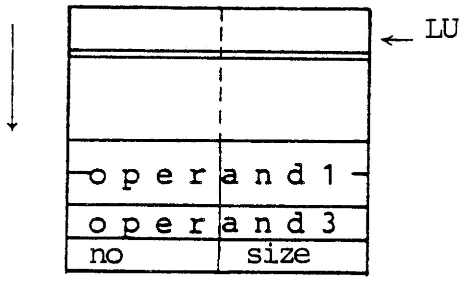
```
VAR
  dopeadress : adr;
  index : integer;
  nindex : integer;
BEGIN
  dopeadress.disp:= pop;
  dopeadress.base:= basetype(pop);
  index:= pop;
  IF checkrange(dopeadress,index,nindex)
  THEN BEGIN
    pus:= nindex;
    pus:= mem(dopeadress);
  END
  ELSE exception(indexexception);
END;
```

12.3.2 INPDV

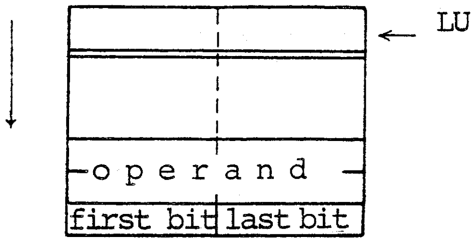
INdex Packed Dope Vector Value: $6F_{Hex}$

IC → INPDV

STACK BEFORE:



STACK AFTER:



```
VAR
  val : word;
  nindex : integer;
  size, no : byte;
  operand : adr;
  firstbit, lastbit : byte;
BEGIN
  val:= pop;
  size:= val and 255;
  no:= swap(val) and 255;
  nindex:= pop;
  operand.disp:= pop;
  operand.disp:= operand.disp + (nindex div no) * 2;
  firstbit:= (nindex mod no)*size;
  lastbit := firstbit + size - 1;
  pus:= operand.disp;
  pus:= firstbit  * 256 + lastbit;
END;
```

13. MISCELLANEOUS 13.

13.1 No Operation 13.1

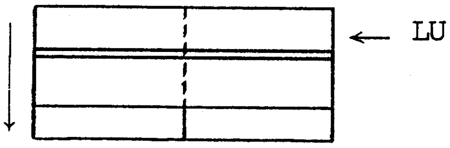
The execution of this instruction has no effect. The instruction has no operands.

13.1.1 MNOOP 13.1.1

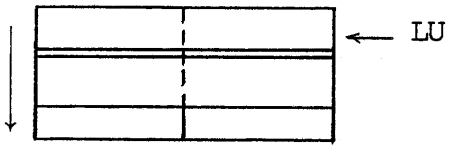
Miscellaneous NO OPeration Value: $2F_{Hex}$

IC → $\boxed{\text{MNOOP}}$

STACK BEFORE:



STACK AFTER:



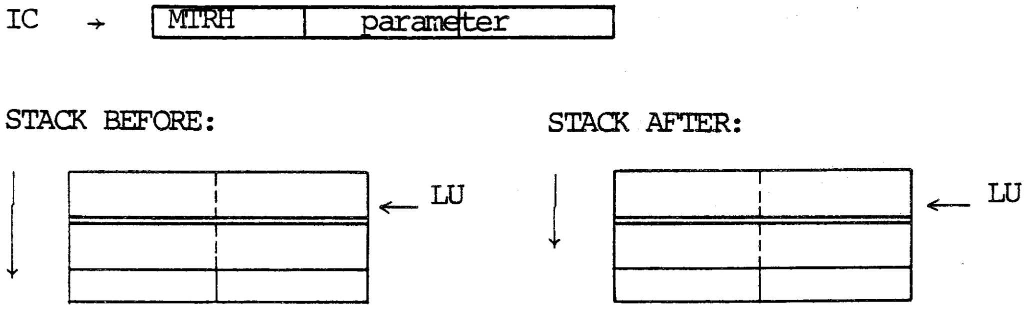
```
BEGIN
END;
```

13.2 Exception 13.2

This instruction performs an unconditional jump to the program point defined in the exception point field of the incarnation descriptor.

13.2.1 MTRH 13.2.1

Miscellaneous Trap Here Value: $2C_{Hex}$



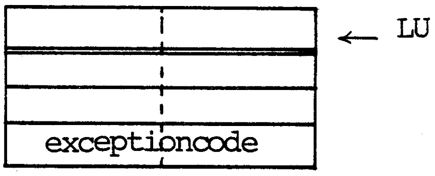
```
BEGIN
  exception(nextword);
END;
```

13.2.2 MTRS

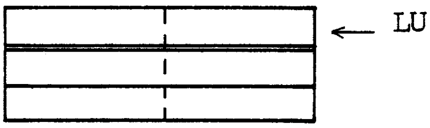
Miscellaneous Trap Stack   Value: $2E_{Hex}$

IC → MTRS

STACK BEFORE:



STACK AFTER:



```
BEGIN
  exception(pop);
END;
```

14. AUTOLOAD 14.

The autoload function can be initiated in the following way:

- Power Restart
- Watchdog Restart

Power Restart

The built-in test programs are activated, controlled by the MODE switch, (see subsection 15.2.1) and the CPU initializes the registers, whereafter control is passed to the autoload program residing in memory module #hE0.

Power Restart occurs:

- when power is turned ON manually on the Operator's Control Panel (OCP) or on the power supply,
- after a temporary power failure,
- on manual activation of the autoload button on the OCP or the AUTO push-button on the power supply.

Watchdog Restart

The CPU initializes the registers, whereafter control is passed to the autoload program. No built-in test programs are activated. The watchdog function can be activated manually by means of the 'Y' debug-console command as well as from the software (see subsection 3.12.4).

14.1 Initialization

The following algorithm is executed by the microprogram:

```
init:

BEGIN
  REPEAT UNTIL stopmode;
  (* initialize micromachine *)
  FOR i:=0 TO #h7f DO
  WITH registerset(i) DO BEGIN
    REPEAT x:=pop UNTIL stacksize=0;
    FOR j:=0 TO 7 DO registers(i*8+j):=#hffff;
    intmap(i):=dummyregset;
    activeflags(i):=0;
  END;
  j:= 0
  FOR i:= 0 TO 15 DO
  BEGIN
    j:= 2*j + 1;
    masks(i):= j;
  END;
  FOR i:=#h3e0 TO #h3ff DO registers(i):=0;
  reg8085(fifo56):=  version;
  parityerror:= false;
  context:= dummyreg;
  startadr.base:= basetype(#he0);
  startadr.disp:= #h018;
  ib:=ibtype(membyte(startadr));
  ic:=memword(adroffset(startadr,1));
END;
```

15. SWITCHES AND INDICATORS 15.

15.1 Operator's Control Panel 15.1

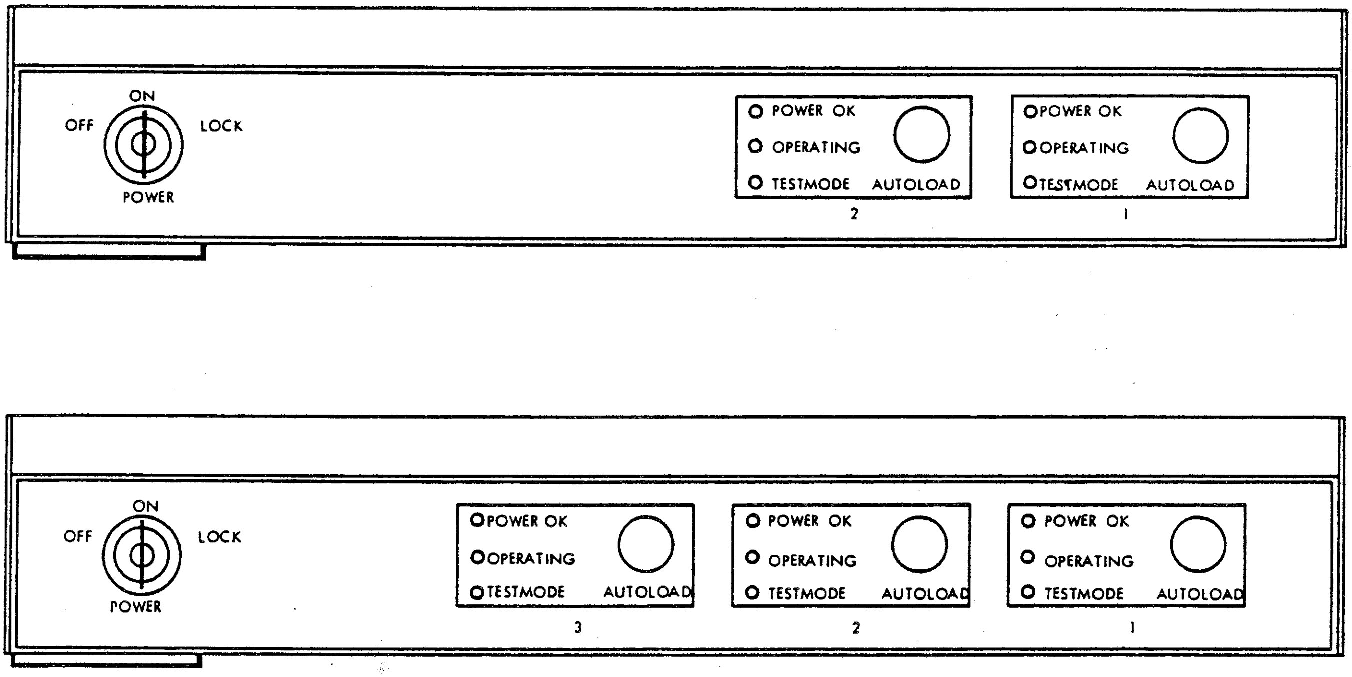


Figure 31: OCP for rack with two RC3502 or three RC3502.

Power off of the RC3502(s) is done by turning the power key to the OFF position.

Power on of the RC3502(s) is done by turning the key to the ON position (or further on to the LOCK position).

The AUTOLOAD button(s) is (are) enabled when the key is in the ON position, and disabled, when in the LOCK position.

The AUTOLOAD button initiates autoloading of the RC3502 in question.

The POWER OK indicator is illuminated during power OK condition on the RC3502.

The OPERATING lamp indicates that the RC3502 is running normally.

The TEST MODE lamp indicates that the RC3502 is executing the built-in test programs.

15.2 Processor Front Panel 15.2

The front panel of the processor boards contains five switches, five indicators and a jack.

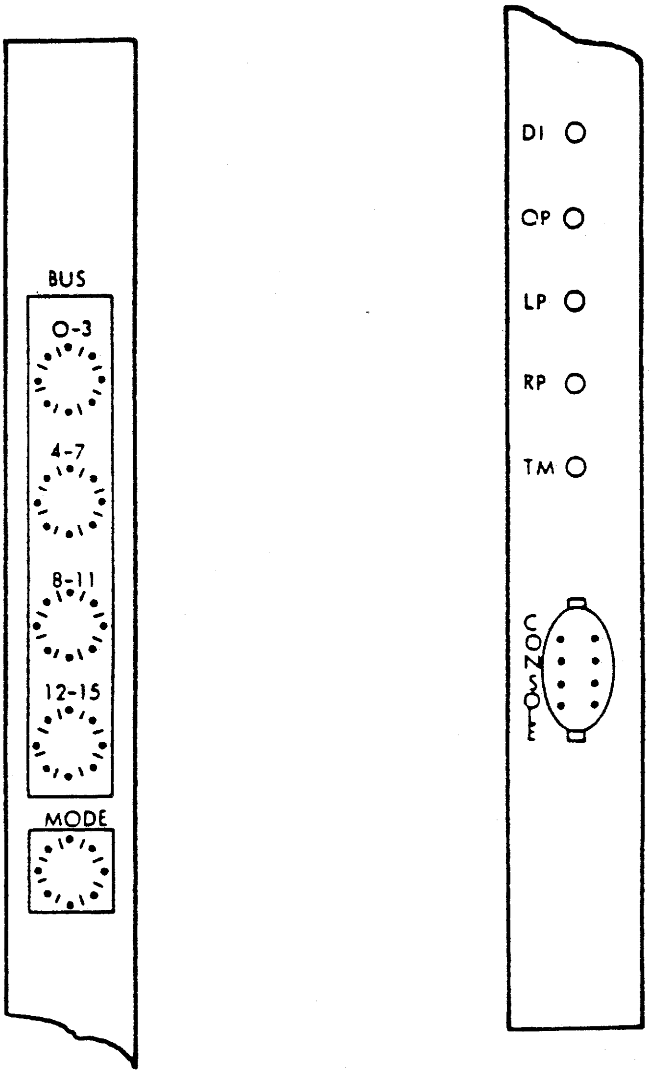


Figure 32: Processor front panel, switches and indicators.

15.2.1 Switches 15.2.1

All of the switches are rotary switches with 16 positions, indicated by the hexadecimal numbers 0 to F. The switches are set by means of a screwdriver.

15.2.1.1 Bus Switches 15.2.1.1

The four switches marked BUS are used to supply the processor with data. There is a switch for bits 0 to 3, 4 to 7, 8 to 11 and 12 to 15.

15.2.1.2 Mode Switch 15.2.1.2

The switch marked MODE is used to control the baud rate for the console and the execution of the built-in test programs (subsection 15.2.2).

If the mode switch is equal to, or greater than 8, the console is locked to Terminal-mode (T-mode), i.e. the console will not switch to Debug-mode (D-mode) by activating the BELL key (CTRL G). The mode switch is only read after power restart.

Settings		Baud Rate	Execution Mode
0	(8)	300 bps	run test, loop
1	(9)	1200 bps	run test, loop
2	(A)	300 bps	skip test
3	(B)	1200 bps	skip test
4	(C)	300 bps	run test, no loop
5	(D)	1200 bps	run test, no loop
6	(E)	300 bps	skip test
7	(F)	1200 bps	skip test

Test Program Execution Modes

run test — The test programs are executed whenever the autoload button is pressed.

skip test	The test programs are not executed.
loop	The test programs are executed in an endless loop.
no loop	The test programs are executed once.

15.2.2 Indicators

DI Disables Interrupt
This lamp, when lit, indicates that the processor is running in the disabled interrupt mode.

OP Operating
This lamp, when lit, indicates that the processor is running normally; when it is extinguished, the processor has stopped.

LP Left Parity Error
This lamp, when lit, indicates that a parity error has been detected during a memory read in the left byte. The lamp can be extinguished only by autoloading.

RP Right Parity Error
This lamp, when lit, indicates that a parity error has been detected during a memory read in the right byte. The lamp can be extinguished only by autoloading.

TM Test Mode
This lamp, when lit, indicates that the processor is executing the built-in test programs. The current program is indicated by the DI, OP, LP, RP and TM lamps. TM representing the least significant bit of the program number.

If an error is detected by a test program, one of the following messages is displayed on the console:

1 8085 Communication Test
Message: ERR01 <dummy><dummy><dummy>

Y5D gives 6 bytes of transmitted data.
Y70 gives 6 bytes of received data.

3 Interrupt Test

Message: No message.

Test microprogrammed interrupt of control microprocessor. RP and TM are lit.

5 Working Register Address Test

Message: ERR02 <address><errdata><04>
OK data = address.

7 Working Register Data Test

Message: ERR03 <address><errdate><sub>

```
sub = 01: if lsb(address) = 0 then OK data = AAAA
                                   else OK data = 5555
sub = 02: if lsb(address) = 0 then OK data = 5555
                                   else OK data = AAAA
```

9 Memory Address Test

Message: ERR04 <address><errdata><sub>
Y40 gives error module.

The test will read both by means of word and by byte read. In the latter case OK data is the byte contents of the address read by byte read.

```
sub = 02: right parity error = R
      04: left  parity error = L
      06: left and right parity error = LR
      41: dataerror
      43: dataerror + R
      45: dataerror + L
      47: dataerror + LR
```

B Memory Data Test

Message: ERR05 <address><errdata><sub>
Y40 gives error module.

sub = 02: right parity error = R
04: left parity error = L
06: left and right parity error = LR
40: dataerror, okdata = AAAA in addr 0000
41: dataerror, okdata = 5555
42: dataerror + R, okdata = AAAA
43: dataerror + R, okdata = 5555
44: dataerror + L, okdata = AAAA
45: dataerror + L, okdata = 5555
46: dataerror + LR, okdata = AAAA
47: dataerror + LR, okdata = 5555

The test will write alternating AAAA,5555.

D <u>Internal Interrupt Test</u>

Message: ERR06 <low,high><errdata><04>

low = byte with lowest interrupt
high = byte with highest interrupt

OK data = high.

F <u>Schedule Test</u>

Message: ERR07 <param1><errdata><sub>

sub = 01: no external interrupt
02: maperror, okdata = 07FF
03: external interrupt or missing "interrupt chain terminator"
04: maperror, okdata = 0007
06: coroutine error, okdata = param1
07: medium priority error, okdata = param1
08: low priority error, okdata = param1
09: high FF, okdata = 000F
0A: medium FF, okdata = 0017
0B: low FF, okdata = 001F

11 Interrupt Map Test

Message: ERR08 <address><errdata><04>

okdata = address.

13 Prefetch Test

Message: ERR09 <address><errdata><sub>

sub = 01: load of ICD, okdata = 5555

02: load of ICD, okdata = AAAA

03: nxtbyte read ICD,okdata = AAAB

04: nxtword read ICD,okdata = AAAD

05: nxtbyte read ICD,okdata = address

06: read of nxtbyte, okdata = address and OFF

07: ICD, okdata = address

08: nxtword even ICD,okdata = address

09: nxtword, okdata = address

0A: odd addr, okdata = address

15 Register Stack Test

Message: ERR0A <param1><errdata><sub>

sub = 01: not stack limit

02: stack limit

03: size error, okdata = 0016

04: stack limit

05: stack limit

06: data error, okdata = param1+7

08: stack limit

09: stack limit

0A: data error, okdata = param1+7

15.3 Power Supply 15.3

The power supply POW204 is supplied with the following controls:

POWER: Circuit breaker, lit when power on.

POWER OK: Indicator which is illuminated during power ok condition.

POWER FAILURE:

OVER-TEMPERATURE:

OVER-VOLTAGE: Error indicators which are illuminated after an error condition. These indicators are reset after activating the circuit-breaker, or after activating the RESET push-button.

RESET: Push-button for manual generation of an autoload signal and a reset of the error indicators.

15.4 Connection of the Console 15.4

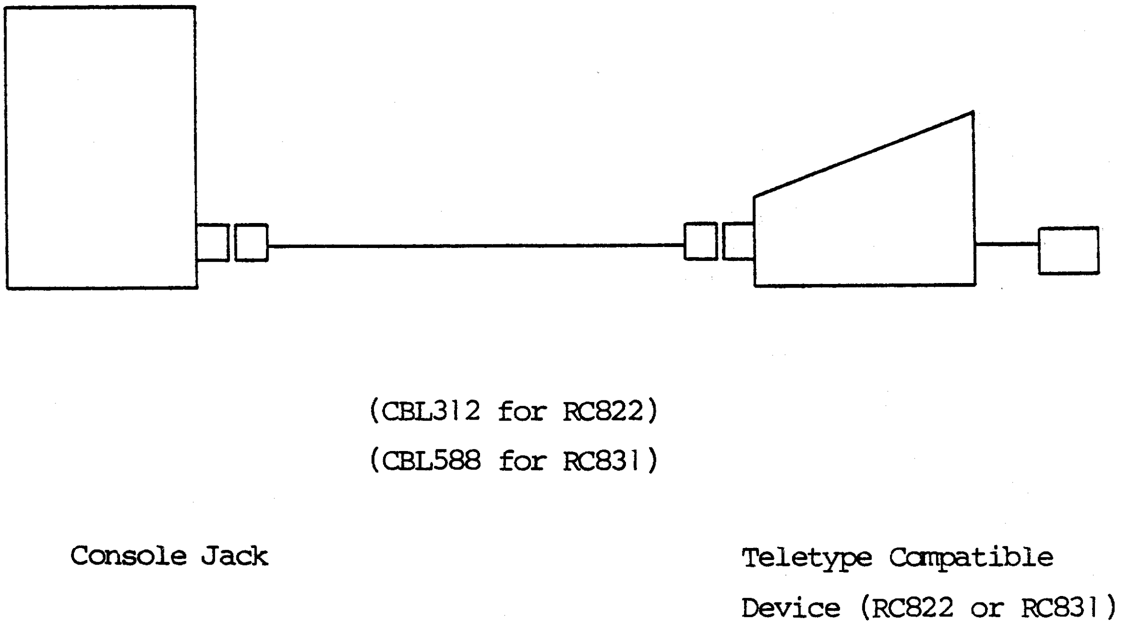


Figure 33: Connection of the Console.

16. DEBUG CONSOLE 16.

The debug console can be in one of two possible modes: debug mode (D mode) or terminal mode (T mode). A switch between the two modes takes place when the BELL key (CTRL and G) is pressed.

16.1 Activation of the Console 16.1

If the MODE switch (subsection 15.2.1.2) is set in the range 0 to 7, the debug console can be activated at any time by pressing the BELL key (CTRL and G) without stopping instruction execution in the processor.

16.2 Display Commands 16.2

Display commands cause the display of eight words of data. The following display commands are available:

M <addr> Modify Memory
Displays the contents of the 8 memory words starting at <addr>.

W <regset> Modify Working Registers
Displays the contents of the 8 working registers comprising registerset <regset>.

P <regset> Modify Register Stack
Didplays the contents of the register stack associated registerset <regset>. At most 8 register stack elements are displayed.

L <level> Modify Working Registers
Displays the level number, the registerset and the contents of the 8 working registers comprising the registerset connected to <level>.

Y <yaddr> Modify Control Microprocessor RAM

Displays the contents of the 8 control microprocessor RAM bytes starting at <yaddr>.

Display commands are executed

When a display command is entered, one can now modify the displayed data by entering new data in the same positions on the following line. Pressing the space bar will move the cursor one position to the right.

When a P command is terminated (by CR, +, or -) the cursor position defines the number of register stack elements. If the number has been changed, a # is displayed. Note that a cursor on the first position does not empty the register stack. This is done by the # key (see later).

A display command is terminated by pressing one of the following keys:

CR The CR key terminates the current display command. The P command terminates with a number sign marking the last element in the register stack. The console will await the next command.

+ The + key terminates the current display command and executes a display command for the succeeding 8 words (M), 8 bytes (Y), up to 8 elements (P), or the 8 registers on the succeeding level (W).

- The - key terminates the current display command and executes a display command for the preceding 8 words (M), 8 bytes (Y), up to 8 elements (P), or the 8 registers on the preceding level (W).

ESC The ESC key terminates the current display command, but no data modification takes place in M, W, P and L commands. The text <ESC> is displayed. The console will await the next command.

16.3 Control Commands

The following control commands are available:

R Run

The processor will start instruction execution.

S Instruction Step

The processor will execute one instruction, stop, and display the current levelno, the registerset, and the contents of the 8 working registers, and reactivate the console.

S <steps> Multi-Instruction Step

The processor will execute <steps> instructions, stop and reactivate the console.

T <testno> Single Selftest

The processor will execute a single selftest, in a loop mode, according to the following table. If testno is chosen as C1-D5, then there will be no errormessage. The test can be terminated by use of the ESC key.Errorno + info are explained in 15.2.2.

testno	testno		
without	with	err	
mess	mess	no	test
00C1	0081	01	fifo test
00C3	0083	-	7.5 interrupt test
00C5	0085	02	W-register address test
00C7	0087	03	W-register data test
00C9	0089	04	memory address test
00CB	008B	05	memory data test
00CD	008D	06	internal intr. test
00CF	008F	07	schedule test
00D1	0091	08	intmap test
00D3	0093	09	prefetch test
00D5	0095	0A	register stack test

16.4 Command Parameters

All numbers entered or displayed are hexadecimal.

At any time the entering of an empty command (i.e. pressing the CR key) will cause the previous command to be repeated.

An address (<addr>) is entered using one of the following formats:

```
    <base> : <disp>
or
           : <disp>
```

<base> is the leftmost 8 bits of the 24-bit address.

<disp> is the displacement within the selected memory module, i.e. the rightmost 16 bits of the address.

If the second format (: <disp>) is used, the last entered address base will be echoed and used.

17. ACTUAL INSTRUCTION SET

The Base Instruction Set (BIS) described in chapters 4 to 13 has been extended with a number of encoded instructions to form the Actual Instruction Set (AIS) for the RC3502. The candidates for encoding were selected after comprehensive analyses of static and dynamic instruction frequency for large application systems. Every encoded instruction in the AIS has a single corresponding instruction in the BIS, from which it is encoded. In practice the encoding follows one of the two models:

- A 16-bit parameter may be expressed in 8 bits, and the encoding is done by introducing a new instruction (new operation code) occupying 8 bits less than its BIS form.

- A single 16-bit parameter value is so frequent that it may be expressed implicitly in the operation code for the encoded instruction.

In the description of encoded instructions, the following information is given:

- Symbolic name
- Hexadecimal operation code value
- Original BIS instruction and the parameter value interval
- Number of bytes occupied.

For the verbal, diagrammatic, and algorithmic descriptions, see the description of the original BIS instruction, which is functionally equivalent.

Symbolic Name	Op-Code Hex Value	Corresponding BIS Instruction and Parameter Value	Number of Bytes Occupied
INDH01	8D	INDH0,X, 1	3
INDH02	AD	INDH0,X, 2	3
INDH11	CD	INDH1,X, 1	3
INDH12	ED	INDH1,X, 2	3
REAGDS	CE	REAGD,X; X [0..255]	2
REALDS	CF	REALD,X; X [0..255]	2
REASD1	30	REASD1	1
REC0	88	RECHW,0	1
REC1	01	RECHW,1	1
REC10	0A	RECHW,10	1
REC11	0B	RECHW,11	1
REC12	0C	RECHW,12	1
REC13	0D	RECHW,13	1
REC14	0E	RECHW,14	1
REC15	0F	RECHW,15	1
REC2	02	RECHW,2	1
REC3	03	RECHW,3	1
REC4	04	RECHW,4	1
REC5	05	RECHW,5	1
REC6	06	RECHW,6	1
REC7	07	RECHW,7	1
REC8	08	RECHW,8	1
REC9	09	RECHW,9	1
RECHWS	C8	RECHW,X; X [0..255]	2
REVGAS	C9	REVGA,X; X [0..255]	2
REVGBS	89	REVGB,X; X [0..255]	2
REVGDS	E9	REVGD,X; X [0..255]	2
REVGWS	A9	REVGW,X; X [0..255]	2
RVLANS	DC	REVLA,X; X [-255..0]	2
REVLAS	CB	REVLA,X; X [0..255]	2
RVLBNS	DA	REVLB,X; X [-255..0]	2
REVLBS	8B	REVLB,X; X [0..255]	2
RVLDNS	DD	REVLD,X; X [-255..0]	2
REVLDS	EB	REVLD,X; X [0..255]	2
RVLWNS	DB	REVLW,X; X [-255..0]	2
REVLWS	AB	REVLW,X; X [0..255]	2
RVSA0	F4	REVSA,0	1
RVSA2	C3	REVSA,2	1
RVSA4	C5	REVSA,4	1
RVSA6	C7	REVSA,6	1
RVSB0	F0	REVSB,0	1
RVSB1	69	REVSB,1	1
RVSB2	83	REVSB,2	1
RVSB3	BB	REVSB,3	1
RVSB4	85	REVSB,4	1
RVSB5	BD	REVSB,5	1

Symbolic Name	Op-Code Hex Value	Corresponding BIS Instruction and Parameter Value	Number of Bytes Occupied
RVSB6	87	REVSB,6	1
RVSB7	BF	REVSB,7	1
RVSD0	F6	REVSD,0	1
RVSD2	E3	REVSD,2	1
RVSD4	E5	REVSD,4	1
RVSD6	E7	REVSD,6	1
RVSW0	F2	REVSW,0	1
RVSW2	A3	REVSW,2	1
RVSW4	A5	REVSW,4	1
RVSW6	A7	REVSW,6	1
STVLAS	CA	STVLA,X; X [0..255]	2
STVLBS	8A	STVLB,X; X [0..255]	2
STVLDS	EA	STVLD,X; X [0..255]	2
STVLWS	AA	STVLW,X; X [0..255]	2
SVSA0	F5	STVSA,0	1
SVSA2	C2	STVSA,2	1
SVSA4	C4	STVSA,4	1
SVSA6	C6	STVSA,6	1
SVSB0	F1	STVSB,0	1
SVSB1	99	STVSB,1	1
SVSB2	82	STVSB,2	1
SVSB3	9B	STVSB,3	1
SVSB4	84	STVSB,4	1
SVSB5	9D	STVSB,5	1
SVSB6	86	STVSB,6	1
SVSB7	9F	STVSB,7	1
SVSW0	F3	STVSW,0	1
SVSW2	A2	STVSW,2	1
SVSW4	A4	STVSW,4	1
SVSW6	A6	STVSW,6	1
UADHW1	30	UADHW,1	1

18. INSTRUCTION EXECUTION TIMES

The measured execution times of the RC3502/2 machine instructions are based on an 18.432 MHz CPU clock, high-speed working registers, and the absence of DMA controllers stealing CPU cycles. The instruction times are measured using MEM204. The execution time includes a non-interrupt instruction fetch, with normal microinstruction flow.

Some of the execution times may vary, e.g. arguments may start in an odd or even address or there may be a differing number of ones in multiplication. An average value is used in these cases. No context shift is included. For each context set shift add 1.6 μs.

The following instructions are interruptable:

IORBB IORBBC IORBW IORBWC
IOWBB IOWBBC IOWBW IOWBWC
MOVEB MOVEBS MOVEG REVSM
SCHED SETAD SETCR SETDI
SETEQ SETIN SETRE SETSB
SETSP SETST SETUN STCEA

The following abbreviations are employed:

SH	shiftout + shiftin time (controller dependent)
W	number of words
Weq	number of equal words before difference
Wsb	number of subset word
Wsp	number of superset word
B	number of bytes
Beq	number of equal bytes before difference
L	last bit number in a field instruction bit; 0 is most significant bit

R	static procedure level difference
SS	words in registerstack
I	additional time for each interrupt exclusiv registerset shift.
µsec.	microsecond(s)

Instruction	µsec.
ABS	
< 0	1.5
≥ 0	1.1
ADD	1.3
AND	1.1
CEXCH	10.2
CGREG	0.8
CINWQ	
empty queue	2.0
non empty queue	2.5
COMPL	1.1
COUTWQ	
no in queue	2.0
1 in queue	3.0
> 1 in queue	4.2
CRC16	6.5
CRELE	CSIGN + 3.3
CRGET	1.5
CRPUT	1.5
CRRAM	2.0
CSELL	
level = 0, new level = 0	5.0
level = 0, new level > 0	6.2
level > 0, new level = 0	5.9
level > 0, new level > 0	7.1
CSENS	
open semaphore	
going passive	10.1
staying open	13.0
passive and closed semaphore	4.0
CSIGN	
open semaphore	14.8
passive semaphore	10.8
closed semaphore startlevel = 0	
going passive	9.6
staying closed	
next incarnation has a registerset	9.9
next incarnation has no registerset	10.9

Instruction	µsec.
closed semaphore startlevel > 0	
going passive	10.2
staying closed	
next incarnation has a registerset	10.4
next incarnation has no registerset	11.4
closed semaphore incarnation has no registerset	
going passive	15.3
staying closed	
next incarnation has a registerset	16.5
next incarnation has no registerset	17.5
CSLEV	1.5
CSTART	
level = 0	2.8
level > 0	3.5
CSTOP	
not in semaphore	6.7
in semaphore	6.7 + unchain
CWAIT	
open semaphore	
going passive	10.1
staying open	13.0
passive semaphore	14.2
closed semaphore	
first and last incarnation has a registerset	14.3
first incarnation has a registerset last incarnation has no registerset	15.4
first incarnation has no registerset last incarnation has a registerset	16.5
first incarnation has no registerset last incarnation has no registerset	17.9

Instruction	µsec.
CWRAM	5.7
DIV	
result >= 0	14.5
result < 0	14.8
EQ	
equal signed	1.5
different signed	1.7
GE	
equal signed	1.5
different signed	1.7
GT	
equal signed	1.5
different signed	1.7
INDEX	
elementsize < 256	7.1
elementsize >= 256	9.3
INDH0	4.8
INDH01	1.6
INDH02	1.6
INDH1	5.0
INDH11	1.8
INDH12	1.8
INPDV	15.0
INTH0	1.6
INTPA	4.2
INTRS	3.9
IOCCI	2.0
IOCDA	4.5
IOGI	4.5 + SH
IOGO	3.0
IOIBX	8.6
IONCI	1.7
IORBB	(6.0 + SH) * B
last step if eoi	8.0 + SH
last step count = 0	5.8 + SH

Instruction	µsec.
IORBBC	(6.0 + SH) * B
last step if eoi	5.2 + SH
last step count = 0	7.4 + SH
IORBW	
destination address even	(6.0 + SH) * W
last step if eoi	8.0 + SH
last step count = 0	5.8 + SH
destination address odd	(7.0 + SH) * W
last step if eoi	9.0 + SH
last step count = 0	6.8 + SH
IORBWC	
destination address even	(6.0 + SH) * W
last step if eoi	5.2 + SH
last step count = 0	7.4 + SH
destination address odd	(7.0 + SH) * W
last step if eoi	5.2 + SH
last step count = 0	7.4 + SH
IORS	3.8 + SH
IORSC	
status match	4.6 + SH
status no match	6.4 + SH
IORW	4.4 + SH
IOWBB	5.7 * B
last step	7.7
IOWBBC	5.7 * B
last step	7.2
IOWBW	
source address even	5.7 * W
last step	7.7
source address odd	6.8 * W
last step	8.8
IOWBWC	
source address even	5.7 * W
last step	7.2
source address odd	6.8 * W
last step	8.3
IOWC	2.9
IOWW	3.0
JMCHT	10.0
otherwise	9.6

Instruction	µsec.
JMPGA	6.1
JMPHC	4.2
JMPRW	3.4
JMZEQ	
jump	3.4
no jump	1.7
JMZGE	
jump	3.4
no jump	1.7
JMZGT	
jump	3.4
no jump	1.7
JMZLE	
jump	3.4
no jump	1.7
JMZLT	
jump	3.4
no jump	1.7
JMZNE	
jump	3.4
no jump	1.7
LE	
equal signed	1.5
different signed	1.7
LLOCK	11.8
LPOP	
description clear	15.2
no description clear	13.6
LPUSH	
description copy	19.4
no description copy	13.5
LRELE	3.4
LRESE	3.6
LT	
equal signed	1.5
different signed	1.7
MADD	1.7
MBSET	1.7
MBTES	1.9

Instruction	μsec.
MCIS	MWIS + 2.4
MCIST	MWIST + 2.4
MCIT	MWIT + 2.4
MHALT	
level = 0	5.0
level > 0	5.9
MMUL	15.4
MNOOP	0.7
MOD	
result < 0	14.8
result >= 0	14.5
MOVEB	6.1 + 3.0 * B + 6.6 I
MOVEBS	6.1 + 3.2 * B + 6.6 I
MOVEG	5.3 + 2.4 * W + 6.2 I
MRECHA	
candicate not found	3.0
candidate found	
not in semaphore queue	
has a registerset	5.9
in semaphore queue	
incarnation in queue	11.9
first incarnation in queue	
with more than 1 incarnation	
second has a registerset	11.5
second has no registerset	12.5
last incarnation in queue	
with more than 1 incarnation	
first incarnation has a registerset	
previous incarnation has a registerset	12.2
first incarnation has a registerset	
previous incarnation has no registerset	13.2
first incarnation has no registerset	
previous incarnation has a registerset	13.4

Instruction	µsec.
first incarnation has no registerset previous incarnation has no registerset	14.4
neither first nor last incarnation in queue	
previous incarnation has a registerset next incarnation has a registerset	10.0
previous incarnation has a registerset next incarnation has no registerset	12.0
previous incarnation has no registerset next incarnation has a registerset	12.0
previous incarnation has no registerset next incarnation has no registerset	14.1
MREST	4.1 + SS * 0.9
MSTST	3.0 + SS * 0.9
MSUB	1.7
MTIME	
for each incarnation	
own.timer > 1	5.4
own.timer <= 0	3.8
own.timer = 1, no semaphore level = 0	7.3
own.timer = 1, semaphore level = 0	7.3 + unchain
own.timer = 1, no semaphore level > 0	7.8
own.timer = 1, semaphore level > 0	7.8 + unchain
own.timer = 1, no semaphore no registerset	10.2
own.timer, semaphore no registerset	10.2 + unchain
MTRH	8.4
MTRS	8.1

Instruction	µsec.
MUL	6.7
MWI	8.8
MWIS	
activated by interrupt	0.8 + CWAIT + unchain
activated by message	CWAIT
MWIST	
activated by timer	1.5 + CWAIT
activated by interrupt	2.8 + CWAIT + unchain
activated by semaphore	2.0 + CWAIT
MWIT	
activated by timer	8.2
activated by interrupt	10.6 + unchain
MWST	
activated by timer	1.5 + CWAIT
activated by semaphore	2.0 + CWAIT
MWT	8.2
NE	1.5
NEG	1.2
NOT	1.1
OR	1.1
PCALLS	10.4 + 0.9 * SS
PCALLS0	10.4 + 0.9 * SS
PCALLS1	11.5 + 0.9 * SS
PEXIT	6.1
REAAD	2.0
READB	2.6
READW	2.6
REAGD	2.6
REAGDS	2.1
REAID	2.1 + 0.9 * R
REAISD	1.9 + 0.9 * R
REALD	2.6
REALDS	2.1
REARD	2.6

Instruction	µsec.
REASD	1.6
REASD1	0.9
REAXD	3.1 + 0.9 * SS
RECHD	2.6
RECHW	1.4
REC0..REC15	0.7
RECHWS	0.8
RENHB	3.1 + 0.9 * SS
RENPB	2.9 + 0.9 * SS
RESTA	3.3 + 0.9 * SS + 0.9 * param
RESTC	3.5 + 0.9 * SS
REVGA	
even address	4.4
odd address	3.7
REVGAS	
even address	3.9
odd address	3.2
REVGB	3.0
REVGBS	2.3
REVGD	
even address	4.4
odd address	5.8
REVGDS	
even address	3.9
odd address	5.3
REVGW	
even address	3.0
odd address	4.2
REVGWS	
even address	2.3
odd address	3.6
REVLA	
even address	4.4
odd address	3.7
REVLAS	
even address	3.9
odd address	3.2

Instruction	µsec.
REVLB	3.0
REVLBS	2.3
REVLD	
even address	4.4
odd address	5.8
REVLDS	
even address	3.9
odd address	5.3
REVLW	
even address	3.0
odd address	4.2
REVLWS	
even address	2.3
odd address	3.6
REVPD	1.7
REVPW	1.1
REVSA	
even address	4.4
odd address	3.7
REVSB	2.9
REVSD	
even address	4.6
odd address	6.0
REVSF	
even address	4.9 + 0.2 * (15-L)
odd address	4.0 + 0.2 * (15-L)
REVSM	6.8 + SS * 0.9 + 2.4 * W +6.15 * I
REVSW	
even address	2.9
odd address	4.2
RVLANS	
even address	3.9
odd address	3.2
RVLBNS	2.3
RVLDNS	
even address	3.9
odd address	5.3
RVLWNS	
even address	2.3
odd address	3.6

Instruction	µsec.
RVSA0	
even address	3.5
odd address	2.8
RVSA2, RVSA4, RVSA6	
even address	3.7
odd address	3.0
RVSB0	2.0
RVSB1, RVSB2, RVSB3, RVSB4 RVSB5, RVSB6, RVSB7	2.2
RVSD0	
even address	3.9
odd address	5.3
RVSD2, RVSD4, RVSD6	
even address	4.1
odd address	5.5
RVSW0	
even address	2.0
odd address	3.3
RVSW2, RVSW4, RVSW6	
even address	2.2
odd address	3.5
SETAD	
size equal	3.0
set too small	5.7 + 1.2 * W + 0.9 * SS + 5.0I
set too big	4.5 + 2.4 * W + 5.0 * I
SETATM	6.5
SETCR	
1 word in set	10.0 + 0.9 * SS
>1 word in set	10.8 + 0.9 * SS + 1.2 * W + 5.0 * I
SETDI	10.5 + 3.0 * W + 5.0 * I
SETEQ	5.2 + 2.4 * Weq + 5.0 * I
SETIN	10.5 + 3.0 * W + 5.0 * I
SETRE	7.4 + 0.9 * SS + 2.4 * W + 6.2 * I
SETSB	5.2 + 2.4 * Wsb + 5.0 * I
SETSP	5.2 + 2.4 * Wsp + 5.0 * I
SETST	7.0 + 0.9 * SS + 2.4 * W + 6.2 * I
SETTM	8.0
SETUN	10.5 + 3.0 * W + 5.0 * I

Instruction	µsec.
SHC	3.5
SHC8	0.9
STCEA	2.7 + 2.8 * Beq + 6.3 * I
STNHB	2.5 + 0.9 * SS
STSTC	2.4 + 0.9 * SS
STVGA	
even address	5.4
odd address	3.8
STVGB	2.9
STVGD	
even address	3.8
odd address	6.4
STVGW	
even address	2.9
odd address	4.1
STVLA	
even address	5.4
odd address	3.8
STVLAS	
even address	5.0
odd address	3.4
STVLB	2.9
STVLBS	2.4
STVLD	
even address	3.8
odd address	6.4
STVLDS	
even address	3.3
odd address	5.9
STVLW	
even address	2.9
odd address	4.1
STVLWS	
even address	2.4
odd address	3.6
STVSA	
even address	5.6
odd address	4.0
STVSB	3.0

Instruction	µsec.
STVSD	
even address	4.0
odd address	6.6
STVSF	
even address	6.2 + (15-L) * 0.2
odd address	8.3 + (15-L) * 0.2
STVSW	
even address	3.0
odd address	4.2
SVSA0	
even address	4.9
odd address	3.3
SVSA2, SVSA4, SVSA6	
even address	5.1
odd address	3.5
SVSB0	2.2
SVSB1, SVSB2, SVSB3, SVSB4, SVSB5, SVSB6, SVSB7	2.4
SVSW0	
even address	2.2
odd address	3.3
SVSW2, SVSW4, SVSW6	
even address	2.4
odd address	3.5
SUB	1.3
TEQAD	1.7
TLOCK	3.2
TNILL	2.5
TOPEN	3.0
UADD	1.5
UADHW	1.6
UADHW1	0.9
UDIV	13.7
ULT	1.5
UMOD	13.7
UMUL	15.3

Instruction	µsec.
unchain	
1 incarnation in queue	
with registerset	5.9
without registerset	12.5
unchain first incarnation from queue with more than 1 incarnation	
with registerset	
second in queue has registerset	5.5
second in queue has no registerset	6.5
without registerset	
second in queue has registerset	11.9
second in queue has no registerset	12.9
unchain last incarnation from queue with more than 1 incarnation	
with registerset	
first and previous incarnation has a registerset	6.2
first incarnation has a registerset, previous incarnation has no registerset	7.2
first incarnation has no registerset, previous incarnation has a registerset	7.4
first incarnation has no registerset, previous incarnation has no registerset	8.4
without registerset	
first incarnation has a registerset, previous incarnation has a registerset	12.8
first incarnation has a registerset, previous incarnation has no registerset	13.8
first incarnation has no registerset, previous incarnation has a registerset	14.0
first incarnation has no registerset, previous incarnation has no registerset	15.0
unchain incarnation neither first nor last	
with registerset	
previous incarnation has a registerset, next incarnation has a registerset	4.0

Instruction	µsec.
previous incarnation has a registerset, next incarnation has no registerset	6.0
previous incarnation has no registerset, next incarnation has a registerset	6.0
previous incarnation has no registerset, next incarnation has no registerset	8.1
without registerset	
previous incarnation has a registerset, next incarnation has a registerset	5.8
previous incarnation has a registerset, next incarnation has no registerset	7.8
previous incarnation has a registerset, next incarnation has a registerset	7.8
previous incarnation has no registerset, next incarnation has no registerset	9.9
USUB	1.5
XOR	1.1

A. REFERENCES

[1] RCSL No 52-AA1177:
CPU 212 - CPU 219, Technical Manual
August 1983, Allan Bjørn Kristensen

[2] RCSL No 52-AA1167:
RC3502/2 Real-Time Pascal, Reference Manual
August 1983, Bo Bagger Laursen

[3] RCSL No 52-AA964:
PASCAL80, Report
January 1980, Jørgen Staunstrup

[4] RCSL No 42-i1539:
PASCAL80, User's Guide
October 1980, Jan Bardino

[5] RCSL No 52-AA1195:
RC3502/2 Operating Guide
September 1983, Bo Bagger Laursen

B. INSTRUCTION TABLES B.

B.1 Instructions Listed by Operation Code B.1

For each instruction the following is given: operation code (hexadecimal), symbolic name, and number of bytes in each parameter (if any).

```
00: except
01: rec1
02: rec2
03: rec3
04: rec4
05: rec5
06: rec6
07: rec7
08: rec8
09: rec9
0a: rec10
0b: rec11
0c: rec12
0d: rec13
0e: rec14
0f: rec15
10: csign
11: crele       1
12: movebs
13: cinwq
14: coutwq
15: csens
16: lrele
17: mtime
18: mhalt
19: except
1a: csell
1b: cstop
1c: cstart
1d: iocci
1e: cslev
1f: cgreg
20: mwi
21: iowc
22: iors
23: iorw
24: ioww
25: iogo
26: iogi
27: ionci
28: mrest
29: mstst
2a: iocda

2b: iofbx
2c: mtrh        1 1
2d: iorsc
2e: mtrs
2f: mnoop
30: uadhwl
30: reasdl *
31: ult
32: eq
33: ne
34: lt
35: gt
36: le
37: ge
38: indh0       2 2
39: indh1       2 2
3a: topen
3b: tlock
3c: teqad
3d: tnill
3e: mmul
3f: madd
40: cwait
41: msub
42: uadd
43: usub
44: add
45: sub
46: umul
47: udiv
48: umod
49: mul
4a: div
4b: mod
4c: and
4d: or
4e: xor
4f: crc16
50: neg
51: abs
52: compl
53: shc
54: shc8

55: not
56: setcr
57: settm
58: setatm
59: seteq
5a: setsb
5b: setsp
5c: setun
5d: setin
5e: setdi
5f: setad
60: mwis
61: mcis
62: jmzeq       2
63: jmzne       2
64: jmzlt       2
65: jmzgt       2
66: jmzle       2
67: jmzge       2
68: jmprw       2
69: jmphc       3
6a: jmpga       2
6b: jmcht       3
6c: intrs
6d: index
6e: intpa
6f: inpdv
70: iorbbc
71: iorbb
72: iowbbc
73: iowbb
74: iorbwc
75: iorbw
76: iowbwc
77: iowbw
78: pcals0      3
79: pcals1      3
7a: pcals       3
7b: pexit       2
7c: lpush
7d: lpop
7e: lrese
7f: llock       2

80: mwt
81: mrecha
82: svsb2
83: rvsb2
84: svsb4
85: rvsb4
86: svsb6
87: rvsb6
88: rec0
89: revgbs      1
8a: stvlbs      1
8b: revlbs      1
8c: stnhb       2
8d: indh01      2
8e: renpb
8f: renhb       2
90: readb
91: crget
92: stvgb       2
93: revgb       2
94: setre
95: resta       1
96: stvlb       2
97: revlb       2
98: stvsb       2
99: svsb1
9a: stvsw       2
9b: svsb3
9c: stvsa       2
9d: svsb5
9e: stvsd       2
9f: svsb7
a0: mwit
a1: mcit
a2: svsw2
a3: rvsw2
a4: svsw4
a5: rvsw4
a6: svsw6
a7: rvsw6
a8: rechw       2
a9: revgws      1
aa: stvlws      1

ab: revlws      1
ac: moveb
ad: indh02      2
ae: moveq
af: revpw
b0: readw
b1: crput
b2: stvgw       2
b3: revgw       2
b4: reagd       2
b5: reaisd      1
b6: stvlw       2
b7: revlw       2
b8: revsb       2
b9: rvsb1
ba: revsw       2
bb: rvsb3
bc: revsa       2
bd: rvsb5
be: revsd       2
bf: rvsb7
c0: mwst
c1: inth0       2
c2: svsa2
c3: rvsa2
c4: svsa4
c5: rvsa4
c6: svsa6
c7: rvsa6
c8: rechws      1
c9: revgas      1
ca: stvlas      1
cb: revlas      1
cc: revsm
cd: indh11      2
ce: reagds      1
cf: realds      1
d0: mbset       2
d1: mbtes       2
d2: stvga       2
d3: revga       2
d4: ststc
d5: restc

d6: stvla       2
d7: revla       2
d8: stvsf
d9: revsf
da: rvlbns      1
db: rvlwns      1
dc: rvlans      1
dd: rvldns      1
de: crram
df: cwram
e0: mwist
e1: mcist
e2: reaid       1
e3: rvsd2
e4: reald       2
e5: rvsd4
e6: uadhw       2
e6: reasd     * 2
e7: rvsd6
e8: rechd       2 2
e9: revgds      1
ea: stvlds      1
eb: revlds      1
ec: setst
ed: indh12      2
ee: stcea
ef: revpd
f0: rvsb0
f1: svsb0
f2: rvsw0
f3: svsw0
f4: rvsa0
f5: svsa0
f6: rvsd0
f7: revld       2
f8: stvld       2
f9: revgd       2
fa: stvgd       2
fb: reaad       3
fc: reard       2
fd: reaxd
fe: cexch
ff: except
```

B.2 Instructions Listed by Name

abs	51	ioibx	2b	movebs	12	rec4	04	rvsa6	c7	stvla	d6
add	44	ionci	27	moveg	ae	rec5	05	revsb	b8	stvlas	ca
and	4c	iorbb	71	mrecha	81	rec6	06	rvsb0	f0	stvlb	96
cexch	fe	iorbbc	70	mrest	28	rec7	07	rvsb1	b9	stvlbs	8a
cgreg	1f	iorbw	75	mstst	29	rec8	08	rvsb2	83	stvld	f8
cinwq	13	iorbwc	74	msub	41	rec9	09	rvsb3	bb	stvlds	ea
compl	52	iors	22	mtime	17	rec10	0a	rvsb4	85	stvlw	b6
coutwq	14	iorsc	2d	mtrh	2c	rec11	0b	rvsb5	bd	stvlws	aa
crc16	4f	iorw	23	mtrs	2e	rec12	0c	rvsb6	87	stvsa	9c
crele	11	iowbb	73	mul	49	rec13	0d	rvsb7	bf	svsa0	f5
crget	91	iowbbc	72	mwi	20	rec14	0e	revsd	be	svsa2	c2
crput	b1	iowbw	77	mwis	60	rec15	0f	rvsd0	f6	svsa4	c4
crram	de	iowbwc	76	mwist	e0	rechws	c8	rvsd2	e3	svsa6	c6
csell	1a	iowc	21	mwit	a0	renhb	8f	rvsd4	e5	stvsb	98
csens	15	ioww	24	mwst	c0	renpb	8e	rvsd6	e7	svsb0	f1
csign	10	jmcht	6b	mwt	80	resta	95	revsf	d9	svsb1	99
cslev	1e	jmpga	6a	ne	33	restc	d5	revsm	cc	svsb2	82
cstart	1c	jmphc	69	neg	50	revga	d3	revsw	ba	svsb3	9b
cstop	1b	jmprw	68	not	55	revgas	c9	rvsw0	f2	svsb4	84
cwait	40	jmzeq	62	or	4d	revgb	93	rvsw2	a3	svsb5	9d
cwram	df	jmzge	67	pcals	7a	revgbs	89	rvsw4	a5	svsb6	86
div	4a	jmzgt	65	pcals0	78	revgd	f9	rvsw6	a7	svsb7	9f
eq	32	jmzle	66	pcals1	79	revgds	e9	setad	5f	stvsd	9e
exception	00	jmzlt	64	pexit	7b	revgw	b3	setatm	58	stvsf	d8
exception	19	jmzne	63	reaad	fb	revgws	a9	setcr	56	stvsw	9a
exception	ff	le	36	readb	90	revla	d7	setdi	5e	svsw0	f3
ge	37	llock	7f	readw	b0	rvlans	dc	setea	59	svsw2	a2
gt	35	lpop	7d	reagd	b4	revlas	cb	setin	5d	svsw4.	a4
index	6d	lpush	7c	reagds	ce	revlb	97	setre	94	svsw6	a6
indh0	38	lrele	16	reaid	e2	rvlbns	da	setsb	5a	sub	45
indh01	8d	lrese	7e	reaisd	b5	revlbs	8b	setsp	5b	teqad	3c
indh02	ad	lt	34	reald	e4	revld	f7	setst	ec	tlock	3b
indh1	39	madd	3f	realds	cf	rvldns	dd	settm	57	tnill	3d
indh11	cd	mbset	d0	reard	fc	revlds	eb	setun	5c	topen	3a
indh12	ed	mbtes	d1	reasd	e6	revlw	b7	shc	53	uadd	42
inpdv	6f	mcis	61	reasd1	30	rvlwns	db	shc8	54	uadhw	e6
inth0	c1	mcist	e1	reaxd	fd	revlws	ab	stcea	ee	uedhw1	30
intpa	6e	mcit	a1	rechd	e8	revpd	ef	stnhb	8c	udiv	47
intrs	6c	mhalt	18	rechw	a8	revpw	af	ststc	d4	ult	31
iocci	1d	mmul	3e	rec0	88	revsa	bc	stvqa	d2	umod	48
iocda	2a	mnoop	2f	rec1	01	rvsa0	f4	stvab	92	umul	46
iogi	26	mod	4b	rec2	02	rvsa2	c3	stvqd	fa	usub	43
iogo	25	moveb	ac	rec3	03	rvsa4	c5	stvqw	b2	xor	4e

RETURN LETTER

Title: RC3502/2 Reference Manual RCSL No.: 52-AA1192

A/S Regnecentralen af 1979/RC Computer A/S 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
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark