

---

RCSL No: 52-AA972  
Edition: February, 1983  
Author: Poul Mølgaard

---

Title:

RC3502 Reference Manual

---

---

**Keywords:**

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

---

**Abstract:**

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

(360 printed pages).

---

Copyright © 1983, A/S Regnecentralen af 1979  
RC Computer A/S  
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

|                                             |    |
|---------------------------------------------|----|
| 1. INTRODUCTION .....                       | 1  |
| 1.1 Processing Unit .....                   | 1  |
| 1.1.1 Working Registers .....               | 1  |
| 1.1.2 Interruption System .....             | 1  |
| 1.1.3 Instruction Set .....                 | 2  |
| 1.1.4 Input/Output .....                    | 3  |
| 1.1.5 System Start-Up .....                 | 3  |
| 1.2 Debugging System .....                  | 4  |
| 1.3 Summary of the Hardware System .....    | 4  |
| 1.3.1 Basic Physical Unit .....             | 4  |
| 1.3.2 Additional Hardware Modules .....     | 4  |
| 1.3.3 Processing Unit Architecture .....    | 5  |
| 1.3.4 Processing Unit Instruction Set ..... | 6  |
| 2. DESIGN CONSIDERATIONS .....              | 8  |
| 2.1 Memory Addressing .....                 | 8  |
| 2.2 Register Structure .....                | 8  |
| 2.3 Instruction Set .....                   | 8  |
| 2.4 Input/Output Structure .....            | 9  |
| 2.5 Modularity .....                        | 10 |
| 3. COMPUTER ORGANIZATION .....              | 12 |
| 3.1 Processing Unit Architecture .....      | 12 |
| 3.2 Backplane Bus .....                     | 15 |
| 3.3 Control Microprocessor .....            | 16 |
| 3.3.1 Variable Array .....                  | 17 |
| 3.3.2 Real-Time Clock .....                 | 18 |
| 3.3.3 Console (TTY) Communication .....     | 19 |
| 3.3.4 Watchdog .....                        | 19 |
| 3.3.5 Configuration .....                   | 19 |
| 3.4 Interruption System .....               | 20 |
| 3.5 Input/Output .....                      | 21 |
| 3.5.1 Serial Transfer .....                 | 22 |
| 3.5.2 DMA Transfer .....                    | 23 |
| 3.5.3 Dual-Port Memory .....                | 24 |
| 3.6 Register Array .....                    | 24 |
| 3.6.1 Working Registers .....               | 24 |

| TABLE OF CONTENTS (continued) |                                                     | PAGE |
|-------------------------------|-----------------------------------------------------|------|
| 3.6.2                         | Micromachine Working Area                           | 30   |
| 3.7                           | Data Formats and Structures                         | 34   |
| 3.7.1                         | Basic Formats and Structures                        | 34   |
| 3.7.2                         | Communication Structures                            | 37   |
| 3.7.3                         | The Process Incarnation Stack Descriptor            | 40   |
| 3.7.4                         | Structures Depending on Stack Oriented execution    | 41   |
| 3.7.5                         | Structures Deriving from Process Execution          | 42   |
| 3.7.6                         | Structures Deriving from Range Testing and Indexing | 44   |
| 3.8                           | Number Representation                               | 44   |
| 4.                            | BASE INSTRUCTION SET                                | 45   |
| 4.1.                          | Runtime Environment                                 | 45   |
| 4.2.                          | Terminology                                         | 48   |
| 4.2.1.                        | Operation and Parameter                             | 48   |
| 4.2.2.                        | Operand and Result                                  | 48   |
| 4.2.3.                        | Effective Address                                   | 48   |
| 4.2.4.                        | Address                                             | 48   |
| 4.2.5.                        | Size                                                | 49   |
| 4.2.6.                        | Addressing                                          | 49   |
| 4.3.                          | Functional Description                              | 51   |
| 4.3.1.                        | Structure                                           | 51   |
| 4.3.2.                        | Notation                                            | 52   |
| 4.3.3.                        | Routines                                            | 54   |
| 5.                            | RETRIEVAL OF A VALUE                                | 56   |
| 5.1.                          | Push Nonsense (Reserve Stack Space)                 | 56   |
| 5.1.1.                        | RENPB                                               | 56   |
| 5.1.2.                        | RENHB                                               | 57   |
| 5.2.                          | Push Constant                                       | 58   |
| 5.2.1.                        | RECHW                                               | 58   |
| 5.2.2.                        | RECHD or REAAD                                      | 59   |
| 5.3.                          | Push Address                                        | 60   |
| 5.3.1.                        | REAXD                                               | 60   |
| 5.3.2.                        | REARD                                               | 61   |
| 5.3.3.                        | REALD                                               | 62   |
| 5.3.4.                        | REAGD                                               | 63   |
| 5.3.5.                        | REARD                                               | 64   |



| TABLE OF CONTENTS (continued) | PAGE |
|-------------------------------|------|
| 5.3.6 REASD or UADHW .....    | 65   |
| 5.4 Push Operand .....        | 66   |
| 5.4.1 REV PW .....            | 66   |
| 5.4.2 REV PD .....            | 67   |
| 5.4.3 REV AB .....            | 68   |
| 5.4.4 REV AW .....            | 69   |
| 5.4.5 REV AD .....            | 70   |
| 5.4.6 REV AF .....            | 71   |
| 5.4.7 REV LB .....            | 72   |
| 5.4.8 REV LW .....            | 73   |
| 5.4.9 REV LD .....            | 74   |
| 5.4.10 REV LF .....           | 75   |
| 5.4.11 REV GB .....           | 76   |
| 5.4.12 REV GW .....           | 77   |
| 5.4.13 REV GD .....           | 78   |
| 5.4.14 REV GF .....           | 79   |
| 5.4.15 REV IB .....           | 80   |
| 5.4.16 REV IW .....           | 81   |
| 5.4.17 REV ID .....           | 82   |
| 5.4.18 REV IF .....           | 83   |
| 5.4.19 REV SB .....           | 84   |
| 5.4.20 REV SW .....           | 85   |
| 5.4.21 REV SD .....           | 86   |
| 5.4.22 REV SF .....           | 87   |
| 5.4.23 READB .....            | 88   |
| 5.4.24 READW .....            | 89   |
| 6. STORAGE OF A VALUE .....   | 90   |
| 6.1 Pop Garbage .....         | 90   |
| 6.1.1 STNHB .....             | 90   |
| 6.2 Pop Result .....          | 91   |
| 6.2.1 STVAB .....             | 91   |
| 6.2.2 STVAW .....             | 92   |
| 6.2.3 STVAD .....             | 93   |
| 6.2.4 STVAF .....             | 94   |
| 6.2.5 STVLB .....             | 96   |
| 6.2.6 STVLW .....             | 97   |

| TABLE OF CONTENTS (continued) |                                    | PAGE |
|-------------------------------|------------------------------------|------|
| 6.2.7                         | STVLD                              | 98   |
| 6.2.8                         | STVLF                              | 99   |
| 6.2.9                         | STVGB                              | 101  |
| 6.2.10                        | STVGW                              | 102  |
| 6.2.11                        | STVGD                              | 103  |
| 6.2.12                        | STVGF                              | 104  |
| 6.2.13                        | STVIB                              | 106  |
| 6.2.14                        | STVIW                              | 107  |
| 6.2.15                        | STVID                              | 108  |
| 6.2.16                        | STVIF                              | 109  |
| 6.2.17                        | STVSB                              | 111  |
| 6.2.18                        | STVSW                              | 112  |
| 6.2.19                        | STVSD                              | 113  |
| 6.2.20                        | STVSF                              | 114  |
| 6.3                           | Manipulation of Storage Areas      | 116  |
| 6.3.1                         | Move a Storage Area                | 116  |
| 6.3.2                         | Exchange Two Storage Addresses     | 120  |
| 6.3.3                         | Compare Two Storage Areas          | 122  |
| 6.3.4                         | Push a Storage Area                | 124  |
| 6.3.5                         | Pop a Storage Area                 | 126  |
| 7.                            | PROCEDURE CALL AND EXIT            | 128  |
| 7.1                           | Enter a Routine                    | 128  |
| 7.1.1                         | PCALS                              | 130  |
| 7.1.2                         | PCALD                              | 131  |
| 7.2                           | Exit from a Routine                | 132  |
| 7.2.1                         | PEXIT                              | 132  |
| 8.                            | JUMPS                              | 133  |
| 8.1                           | Unconditional Jumps                | 133  |
| 8.1.1                         | JMPHC                              | 133  |
| 8.1.2                         | JMPPD                              | 134  |
| 8.1.3                         | JMPRW                              | 135  |
| 8.2                           | Case Jump                          | 136  |
| 8.2.1                         | JMCHT                              | 136  |
| 8.3                           | Conditional Jumps with One Operand | 138  |
| 8.3.1                         | JMZEQ                              | 138  |

| TABLE OF CONTENTS (continued) |                              | PAGE |
|-------------------------------|------------------------------|------|
| 80                            | 8.3.2 JMZNE .....            | 139  |
| 88                            | 8.3.3 JMZLT .....            | 140  |
| 107                           | 8.3.4 JMZGT .....            | 141  |
| 108                           | 8.3.5 JMZLE .....            | 142  |
| 109                           | 8.3.6 JMZGE .....            | 143  |
| 109                           | .....                        |      |
| 109                           | 9. OPERATORS .....           | 144  |
| 109                           | 9.1. Monadic Operators ..... | 144  |
| 109                           | 9.1.1 NEG .....              | 144  |
| 109                           | 9.1.2 NOT .....              | 145  |
| 109                           | 9.1.3 ABS .....              | 146  |
| 109                           | 9.1.4 COMPL .....            | 147  |
| 109                           | 9.1.5 TNILL .....            | 148  |
| 109                           | 9.1.6 TOPEN .....            | 149  |
| 109                           | 9.1.7 TLOCK .....            | 150  |
| 109                           | 9.2. Dyadic Operators .....  | 151  |
| 109                           | 9.2.1 ADD .....              | 151  |
| 109                           | 9.2.2 SUB .....              | 152  |
| 109                           | 9.2.3 MUL .....              | 153  |
| 109                           | 9.2.4 DIV .....              | 154  |
|                               | 9.2.5 MOD .....              | 155  |
| 109                           | 9.2.6 AND .....              | 156  |
| 109                           | 9.2.7 OR .....               | 157  |
| 109                           | 9.2.8 SHC .....              | 158  |
| 109                           | 9.2.9 ULT .....              | 160  |
| 109                           | 9.2.10 EQ .....              | 161  |
| 109                           | 9.2.11 NE .....              | 162  |
|                               | 9.2.12 LT .....              | 163  |
| 109                           | 9.2.13 GT .....              | 164  |
| 109                           | 9.2.14 LE .....              | 165  |
| 109                           | 9.2.15 GE .....              | 166  |
| 109                           | 9.2.16 XOR .....             | 167  |
| 109                           | 9.2.17 UADD .....            | 168  |
| 109                           | 9.2.18 USUB .....            | 169  |
| 109                           | 9.2.19 UMUL .....            | 170  |
| 109                           | 9.2.20 UDIV .....            | 171  |
| 109                           | 9.2.21 UMOD .....            | 172  |

TABLE OF CONTENTS (continued)

|                                                                |     |
|----------------------------------------------------------------|-----|
| 9.2.22 MADD .....                                              | 173 |
| 9.2.23 MSUB .....                                              | 174 |
| 9.2.24 CRC16 .....                                             | 175 |
| 9.3 Operations on Sets .....                                   | 177 |
| 9.3.1 Construct a Set from a Subrange .....                    | 178 |
| 9.3.2 Operations on Sets Giving a Set as the Re-<br>sult ..... | 183 |
| 9.3.3 Comparison of Sets .....                                 | 189 |
| 9.3.4 Test for Membership of a Set .....                       | 196 |
| 10. INPUT/OUTPUT .....                                         | 198 |
| 10.1 Write Control .....                                       | 200 |
| 10.1.1 IOWC .....                                              | 200 |
| 10.2 Write Word .....                                          | 201 |
| 10.2.1 IOWW .....                                              | 201 |
| 10.3 General Output .....                                      | 203 |
| 10.3.1 IOGO .....                                              | 203 |
| 10.4 Read Status .....                                         | 205 |
| 10.4.1 IORS .....                                              | 205 |
| 10.5 Read Word .....                                           | 207 |
| 10.5.1 IORW .....                                              | 207 |
| 10.6 General Input .....                                       | 209 |
| 10.6.1 IOGI .....                                              | 209 |
| 10.7 Clear Current Interrupt .....                             | 211 |
| 10.7.1 IOCCI .....                                             | 211 |
| 10.8 Execute Next Instruction After Clearing Interrupt .....   | 212 |
| 10.8.1 IONCI .....                                             | 212 |
| 10.9 Get Current Device Address .....                          | 213 |
| 10.9.1 IOCDA .....                                             | 213 |
| 10.10 Initialize Block Transfer .....                          | 214 |
| 10.10.1 IOIBX .....                                            | 214 |
| 10.11 Write Block of Bytes .....                               | 216 |
| 10.11.1 IOWBB .....                                            | 216 |
| 10.12 Write Block of Bytes and Clear .....                     | 220 |
| 10.12.1 IOWBBC .....                                           | 220 |
| 10.13 Write Block Of Words .....                               | 223 |
| 10.13.1 IOWBW .....                                            | 223 |
| 10.14 Read Block of Words and Clear .....                      | 226 |
| 10.14.1 IORBWC .....                                           | 226 |

| TABLE OF CONTENTS (continued)                 | PAGE |
|-----------------------------------------------|------|
| 10.15 Read Block of Bytes .....               | 229  |
| 10.15.1 IORBB .....                           | 230  |
| 10.16 Read Block of Bytes and Clear .....     | 233  |
| 10.16.1 IORBEC .....                          | 233  |
| 10.17 Read Block of Words .....               | 236  |
| 10.17.1 IORBW .....                           | 236  |
| 10.18 Write Block of Words and Clear .....    | 239  |
| 10.18.1 IOWBW .....                           | 239  |
| 11. MONITOR CONTROL AND SYNCHRONIZATION ..... | 242  |
| 11.1 Signal .....                             | 249  |
| 11.1.1 CSIGN .....                            | 250  |
| 11.1.2 CRELE .....                            | 251  |
| 11.2 Wait .....                               | 252  |
| 11.2.1 CSENS .....                            | 252  |
| 11.2.2 CWAIT .....                            | 253  |
| 11.2.3 CWTAC .....                            | 254  |
| 11.3 Link/Unlink .....                        | 255  |
| 11.3.1 CLLST .....                            | 255  |
| 11.3.2 CSKIP .....                            | 256  |
| 11.4 Select Level .....                       | 257  |
| 11.4.1 CSELL .....                            | 257  |
| 11.5 Stop .....                               | 259  |
| 11.5.1 CSTOP .....                            | 259  |
| 11.6 Multiple Wait .....                      | 261  |
| 11.6.1 MCIS .....                             | 261  |
| 11.6.2 MCIT .....                             | 262  |
| 11.6.3 MCIST .....                            | 263  |
| 11.6.4 MWI .....                              | 264  |
| 11.6.5 MWT .....                              | 265  |
| 11.6.6 MWIS .....                             | 266  |
| 11.6.7 MWIT .....                             | 267  |
| 11.6.8 MWST .....                             | 268  |
| 11.6.9 MWIST .....                            | 269  |
| 11.6.10 MWTAC .....                           | 270  |

| TABLE OF CONTENTS (continued) |                                                                | PAGE |
|-------------------------------|----------------------------------------------------------------|------|
| 11.7                          | Time .....                                                     | 272  |
| 11.7.1                        | MTIME .....                                                    | 272  |
| 11.8                          | Schedule Next Incarnation to Run .....                         | 274  |
| 11.8.1                        | SCHED .....                                                    | 274  |
| 11.9                          | Driver Activation .....                                        | 275  |
| 11.9.1                        | CSTDR .....                                                    | 275  |
| 11.9.2                        | CSLEV .....                                                    | 276  |
| 11.10                         | Register Array Operations .....                                | 277  |
| 11.10.1                       | CRGET .....                                                    | 277  |
| 11.10.2                       | CRPUT .....                                                    | 278  |
| 11.10.3                       | CRRAM .....                                                    | 279  |
| 11.10.4                       | CWRAM .....                                                    | 280  |
| 11.10.5                       | CGREG .....                                                    | 281  |
| 11.10.6                       | MBTES .....                                                    | 282  |
| 11.10.7                       | MBSET .....                                                    | 283  |
| 11.11                         | Push and Pop .....                                             | 284  |
| 11.11.1                       | LPUSH .....                                                    | 284  |
| 11.11.2                       | LPOP .....                                                     | 286  |
| 11.12                         | Lock Support .....                                             | 288  |
| 11.12.1                       | LLOCK .....                                                    | 288  |
| 11.12.2                       | LRESE .....                                                    | 289  |
| 12.                           | INDEXING AN ARRAY .....                                        | 290  |
| 12.1                          | Range Test .....                                               | 290  |
| 12.1.1                        | INTRS .....                                                    | 290  |
| 12.2                          | Indexing an Array .....                                        | 292  |
| 12.2.1                        | INDEX .....                                                    | 293  |
| 12.3                          | Push an Element of a Packed Array .....                        | 295  |
| 12.3.1                        | INPRS .....                                                    | 296  |
| 12.4                          | Pop and Store a Value in an Element of a Packed<br>Array ..... | 298  |
| 12.4.1                        | INFSS .....                                                    | 299  |
| 13.                           | MISCELLANEOUS .....                                            | 301  |
| 13.1                          | No Operation .....                                             | 301  |
| 13.1.1                        | MNOOP .....                                                    | 301  |
| 13.2                          | Exception .....                                                | 302  |
| 13.2.1                        | MXEPT .....                                                    | 302  |

| TABLE OF CONTENTS (continued)                   | PAGE |
|-------------------------------------------------|------|
| 13.3 Trap .....                                 | 303  |
| 13.3.1 TRAPE .....                              | 303  |
| 13.3.2 TRAPR .....                              | 304  |
| 14. INSTRUCTION FETCH .....                     | 305  |
| 15. AUTOLOAD .....                              | 308  |
| 15.1 Initialization .....                       | 309  |
| 16. SWITCHES AND INDICATORS .....               | 310  |
| 16.1 Operator Control Panel .....               | 310  |
| 16.2 Processor Front Panel .....                | 311  |
| 16.2.1 Switches .....                           | 312  |
| 16.2.2 Indicators .....                         | 313  |
| 16.2.3 Jack .....                               | 315  |
| 16.3 Power Supply .....                         | 315  |
| 17. DEBUG CONSOLE .....                         | 317  |
| 17.1 Activation of the Console .....            | 317  |
| 17.2 Display Commands .....                     | 317  |
| 17.3 Control Commands .....                     | 318  |
| 17.4 Command Parameters .....                   | 319  |
| 17.5 Connection of the Console .....            | 320  |
| 18. ACTUAL INSTRUCTION SET .....                | 321  |
| 19. INSTRUCTION EXECUTION TIMES .....           | 324  |
| <br><u>APPENDIX:</u>                            |      |
| A. INSTRUCTION TABLES .....                     | 341  |
| A.1 Instructions Listed by Operation Code ..... | 341  |
| A.2 Instructions Listed by Name .....           | 343  |
| A.3 Instructions, Section Reference .....       | 345  |

[illegible]

100% 100% 100% 100% 100% 100% 100% 100% 100% 100%



## 1. INTRODUCTION

1.

The present publication is the Reference Manual for the RC3502 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).

### 1.1 Processing Unit

1.1

The processing unit contains a 16-bit ALU. The processor can address up to 4M bytes of memory. The processing unit has 124 interruption levels. Each interruption level is connected to a register set, thereby supporting very fast context switching. 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 Working Registers

1.1.1

The processing unit has 124 register sets. A register set occupies eight 16-bit words. Each 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. Each register set is permanently associated with a specific interruption level.

#### 1.1.2 Interruption System

1.1.2

Because each interruption level is connected to a register set, the time required by the processing unit to perform context switching is reduced to the connection of a new register set to the control and arithmetic unit. Each time an instruction is ex-

ecuted, the register set associated with the interrupt signal on the level with the highest priority is selected as the current register set, and the instruction pointed out by this register set is executed as the next instruction.

### 1.1.3 Instruction Set

1.1.3

The analysis of real-time control applications which led to the extensions incorporated in Real-Time Pascal also influenced the design of the instruction set for the processing unit. The design objective was to ensure that operations which are time-critical in such applications would be supported efficiently by the instruction set. These operations include data manipulation, communication between process incarnations, procedure entry and exit, and context switching (incarnation scheduling).

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

Here 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 may access the address space by cycle stealing via the backplane bus or may be equipped with a dual ported memory which is part of the address space. In the latter case the transfer to memory may be performed without interfering with the processing unit.

#### 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 autoloading programs from an external device controlled by switches, and includes programs residing in other PROM modules.

## 1.2 Debugging System

1.2

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

The switches control the autoloading 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.

The connection is controlled by a microprocessor, which 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.

## 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

- o 16-bit processor with 64K bytes of RAM and 8K bytes of EPROM memory on 3 circuit boards.
- o Backplane bus, power supply, and crate with 16 free circuit-board slots for additional hardware modules.

### 1.3.2 Additional Hardware Modules

1.3.2

- o 64K bytes of RAM and 16K bytes of EPROM memory.

- o 64K bytes of EPROM memory.
- o 4-line synchronous HDLC controller (max. 64K bps).
- o 8-line asynchronous multiplexer (max. 9600 bps).
- o 8-channel I/O module. For connection of peripheral devices and front-end processor adapter (for RC8000 computer series).
- o Video display controller (for RC850 terminal series).
- o Multibus adapter. For connection of RC3502 processing units.

### 1.3.3 Processing Unit Architecture

1.3.3

- o 16-bit arithmetic-logic unit (ALU) built around four AM2901A bit-slice chips.
- o Dynamic MOS memory with a cycle time of 480 ns.
- o Up to 4M bytes of directly addressable memory. The basic memory unit is an 8-bit byte. The smallest addressable quantity is a byte.
- o 124 sets of working registers. Each register set is permanently associated with an interruption level.
- o Active process incarnations are scheduled in the processing unit as follows:

Class I: High priority. Interrupt handler class. Managed by the hardware interrupt priority mechanism.

Class II: Medium priority. Coroutine class. Internal priority within the class and round robin for a given priority.

Class III: Low priority. Time-slice class. Time-sliced round-robin algorithm with built-in priority.

- 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.

#### 1.3.4 Processing Unit Instruction Set

1.3.4

- o Stack-oriented instruction set.
- o Arithmetic operations with twos complements.
- o 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).
- o Base Instruction Set comprises 177 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; monitor control and synchronization, including signal and wait operations on queue semaphores; and indexing of arrays.
- o I/O instructions include: read status, write control, read/write word, initialize block transfer, read/write block of bytes/words, and clear current interrupt.

- o Addressing: direct or relative to the instruction pointer, local stack frame, global stack frame, an intermediate stack frame, or address on the stack.

## 2. DESIGN CONSIDERATIONS

2.

### 2.1 Memory Addressing

2.1

The memory of the RC3502 is directly addressable. The smallest addressable quantity is an 8-bit byte. In order to reach the total address space, an address consists of two 16-bit words.

### 2.2 Register Structure

2.2

The RC3502 contains a number of 16-bit words in a 1K register array. This array is organized as a number of register sets.

The execution of code in the RC3502 is closely related to the stack processing of procedural languages. The register set must therefore support the stack structure as it is represented in memory.

Beyond a number of pointers associated with the stack and the program code in memory, the designers considered representing the top of the stack in the register set in order to save memory accesses. Closer analysis indicated, however, that the cost of administering the contemplated "cashe" register would exceed the saving. The idea of a stack representation in the register sets of the RC3502 was therefore abandoned.

Each register set occupies eight 16-bit words. There are 124 register sets. Each register set corresponds to an interruption level.

The many register sets and interruption levels provide advantages, such as greater efficiency, in the performance of context switching and input/output.

### 2.3 Instruction Set

2.3

An RC3502 machine instruction consists in principle of an operation part and a parameter list part.



The operation part occupies an 8-bit byte. This is followed by zero, one, or more parameters, depending on the instruction specified by the operation part. Each parameter may occupy an 8-bit byte, a 16-bit word, or two 16-bit words, depending on the operation code.

The instruction set is designed to support Real-Time Pascal and similar procedure-oriented high-level programming languages, i.e. the instruction set specifically supports stack processing. On the other hand, support is also provided for the basic runtime system, so that linking and unlinking, for example, are efficient. Finally, the instruction set supports operations necessitated by the underlying hardware (e.g. initialization of register sets).

The instruction set is divided into a Base Instruction Set, or BIS, and an Actual Instruction Set, or AIS.

The BIS specifies the functionality of the set, and contains the machine instructions in their longest form.

The AIS contains, in addition, a number of machine instructions each of which corresponds functionally to an instruction in the BIS, but has a shorter form (fewer parameters).

The AIS is constructed on the basis of statistical analyses of the generated code in the BIS from a large number of programs.

## 2.4 Input/Output Structure

2.4

Particular emphasis was placed on the design of an efficient and versatile input/output system, because the RC3502, like most minicomputers, is used in real-time control applications, where I/O capability has a profound influence on total system efficiency.

The input/output system supports character-oriented as well as block-oriented devices. Moreover, the RC3502, equipped with in-

telligent device controllers, facilitates direct multibuffered memory I/O, independent of program control when first initiated, and thereby supports input/output at high rates.

The RC3502 has a large number of I/O channels, on which any of these operations can be performed, depending on the devices connected.

Character and block operations are under program control, and whenever a character or block is ready for transfer, an interrupt signal is generated. Each interrupt signal is associated with a register set. The direct connection between the interrupt signal and a register set minimizes context switching time. The I/O channels with interrupt signals and register sets may be arranged so that high-speed channels have priority over low-speed channels.

The register set associated with a certain interruption level has priority over register sets associated with lower interruption levels. Therefore, processing of input and generation of output should be performed at the lowest level so as not to block the I/O system.

Intelligent input/output controllers based on microprocessors step up throughput in RC3502 systems by relieving the RC3502 processor of time-consuming control tasks. In such a configuration, all communication between the RC3502 processor and the controller is handled through a section of dual-port memory that resides in the controller subsystem. This setup allows more efficient transfer of large blocks of data from the I/O device to the system without contention over access to the backplane bus. It also simplifies interprocessor communication because the subsystem controller appears to the RC3502 as an additional RAM board.

## 2.5      Modularity

2.5

The RC3502 is constructed in a modular fashion. Upgrading in the form of additional memory capacity, controllers, and multiplexers

is accomplished by inserting the relevant modules in the 16 free positions in the RC3502 module crate.

The technology employed is based on contemporary state-of-the-art electronics, including fast bit-slice chips and 16K memory chips.

The crate used in the RC3502 is a standardized mechanical unit, prepared for double Europe boards (DIN NORM 41494). This ensures stable deliveries, proven design, and great flexibility.

### 3. COMPUTER ORGANIZATION

3.

#### 3.1 Processing Unit Architecture

3.1

For details, refer to fig. 1, microprogram structure and fig. 2, internal bus sources and destinations.

The RC3502 processing unit, or PU, is built around an internal data bus (bus 0-15) as shown in fig. 2. This bus is interfaced to the backplane bus, working registers, control microprocessor, and register ALU, a 16-bit arithmetic-logic unit based on four AM2901A CPU bit-slice chips.

The flow on the internal data bus is controlled from the RC3502 microprogram, which implements the various features of the processing unit, and is physically contained in a read-only memory of 2048 60-bit words. The execution time per microinstruction is 217 nanoseconds.

A word in the microinstruction contains the following information:

| <u>Bits</u> | <u>Interpretation</u>                |
|-------------|--------------------------------------|
| 00-03       | microprogram sequencer command       |
| 04-07       | condition select                     |
| 08-15       | register ALU register select         |
| 16-26       | register ALU                         |
| 27-28       | carry load control                   |
| 29-31       | working register address select      |
| 32-37       | bus source and destination control   |
| 38-49       | micromachine manipulation operations |
| 50-59       | next address field                   |

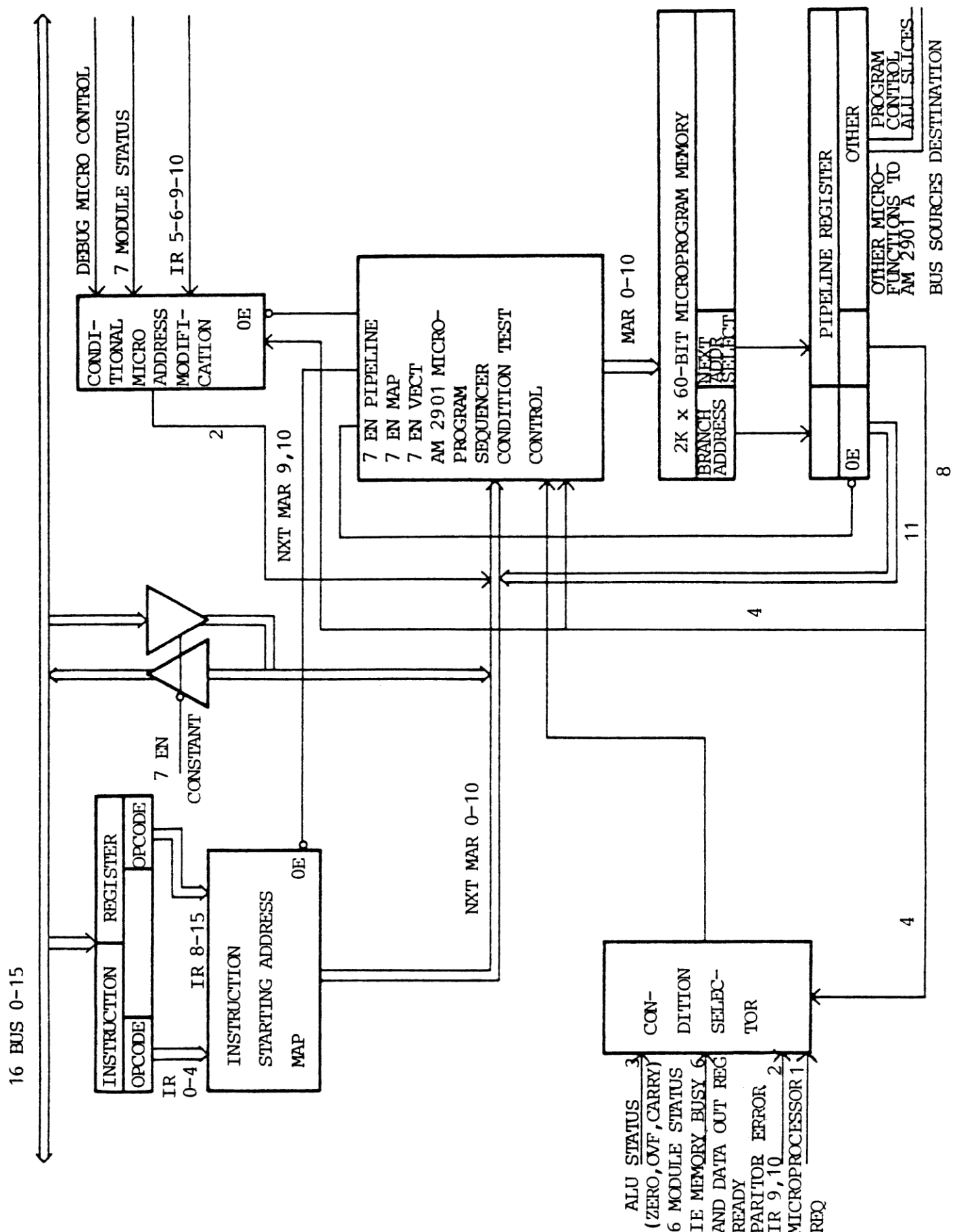


Figure 1: Microprogram Structure.

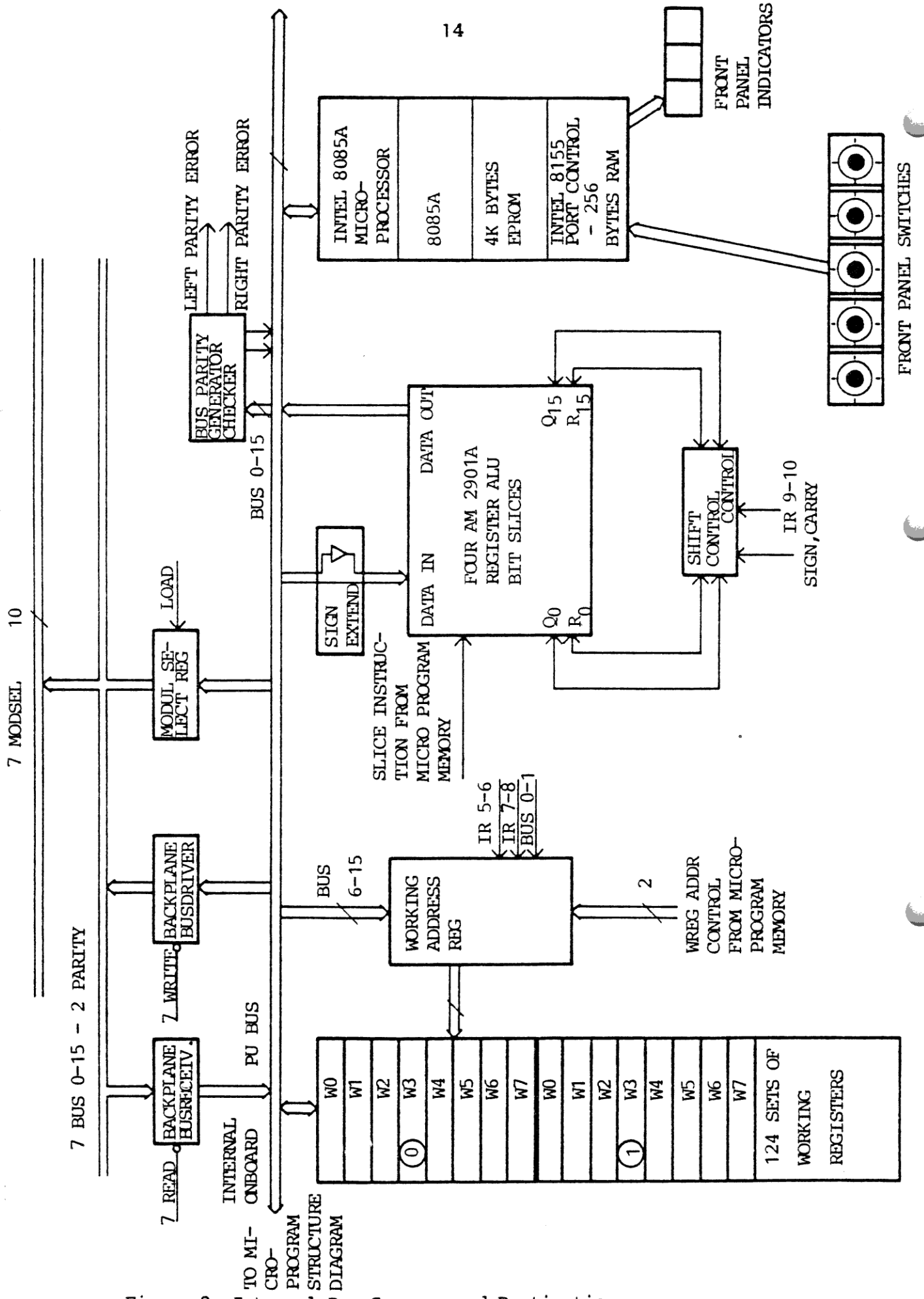


Figure 2: Internal Bus Sources and Destinations.

For details, refer to fig. 3, backplane bus structure.

The backplane bus is used to interconnect the various kinds of modules in the RC3502 computer family.

This bus is normally controlled from the processing unit, for communication between the latter and memory or I/O devices, but there are facilities on the bus which enables other masters to request its use. When the processing unit acknowledges such a request, the bus becomes available to the requesting device.

The backplane bus contains 16 data lines and 8 module address lines. The latter are used to select a source or destination register in a memory module or an I/O device. The data transfer is initiated by a READ or WRITE pulse.

As the backplane bus is asynchronous, the processing unit can extend READ and WRITE until the addressed device acknowledges the operation by XACK.

When the processing unit reads or writes in memory, the address is sent in one backplane bus cycle and the data in the next.

The transfer rate on the backplane bus is 5 MHz.

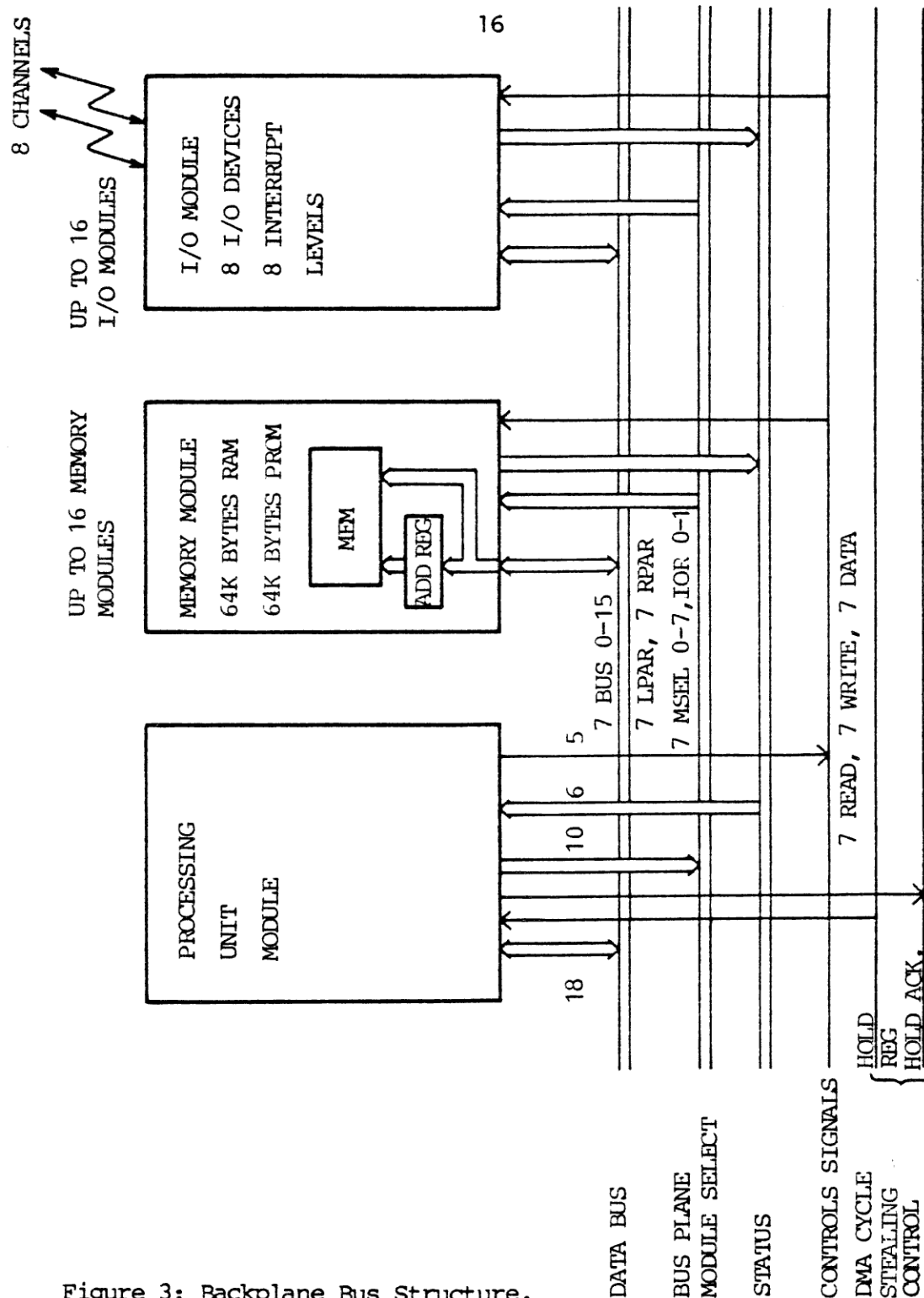


Figure 3: Backplane Bus Structure.

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 16 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 17.

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

The following subsections describe additional functions incorporated in the microprocessor system.

### 3.3.1 Variable Array

3.3.1

For details, refer to fig. 4, control microprocessor RAM layout.

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 section 17.2. 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.

|          |    |                               |
|----------|----|-------------------------------|
| 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  | unused                        |
|          |    |                               |
|          |    |                               |
|          |    |                               |
|          | 1B | RAM module<br>configuration   |
|          | 1C |                               |
|          | 1D |                               |
|          | 1E |                               |
|          | 1F | EPROM module<br>configuration |
|          | 20 | unused                        |
|          |    |                               |
|          |    |                               |
|          |    |                               |
|          |    |                               |
|          |    |                               |
|          |    |                               |
|          | 3F |                               |

Figure 4: Control Microprocessor RAM Layout.

### 3.3.2 Real-Time Clock

### 3.3.2

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.

3.3.3 Console (TTY) Communication

3.3.3

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 undefined.

3.3.4 Watchdog

3.3.4

This timer counts down from the value placed in RAM addr (6, 5), and if 0 is reached, the microprogram is commanded to execute an autoloader. 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.

3.3.5 Configuration

3.3.5

After power up, the microprocessor tests the current RC3502 hardware configuration for RAM modules and EPROM modules, and generates a configuration table in its own RAM area.

RAM Configuration

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

|     | MODULE NUMBER |   |   |   |   |   |   |   |
|-----|---------------|---|---|---|---|---|---|---|
| 1C: | 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1D: | 8             | 9 | A | B | C | D | E | F |

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 (1E, 1F) 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.

A program interruption feature permits automatic switching from the current program sequence to another program sequence in immediate response to specific external and internal events. The efficiency of an interruption system is directly related to the time required by the processing unit to perform such context switching. In the RC3502, therefore, a set of working registers is associated with each interruption level, so that context switching is reduced to the connection of a new register set to the control and arithmetic unit. This is illustrated by fig. 5, interruption system.

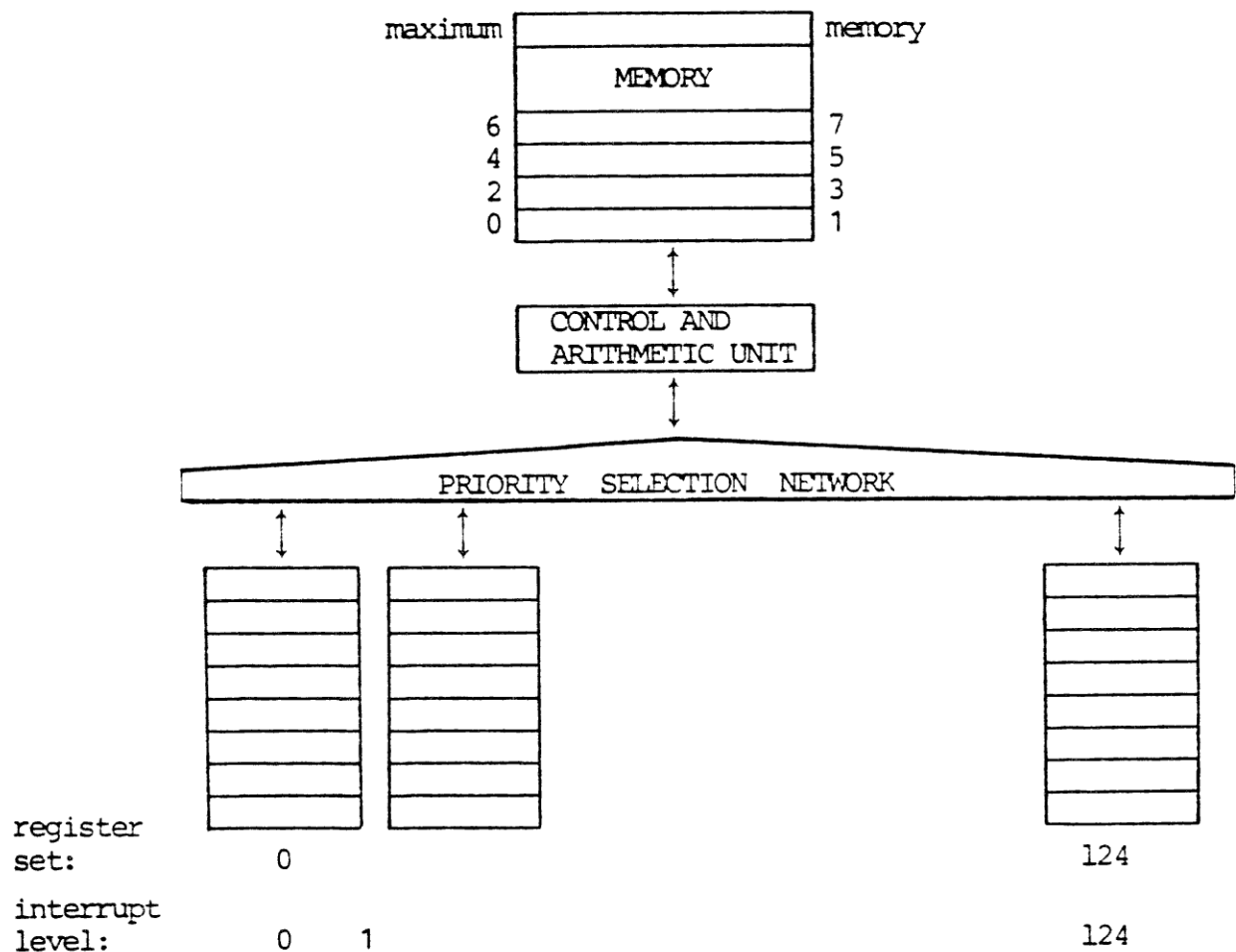


Figure 5: Interruption System.



### 3.5.1 Serial Transfer

3.5.1

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 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<br>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

|                 |   |   |   |   |                          |
|-----------------|---|---|---|---|--------------------------|
| Input<br>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.

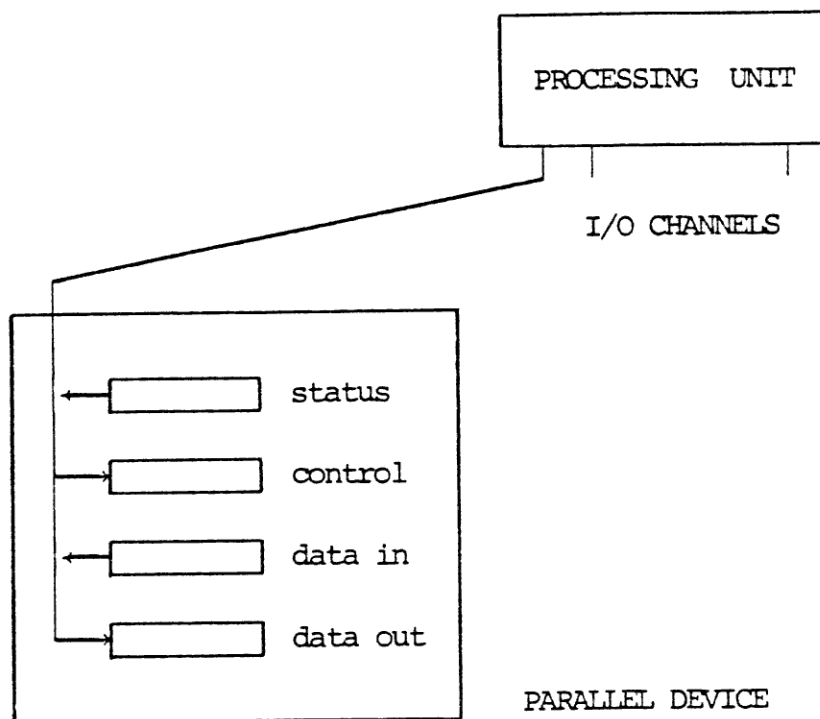


Figure 6: Device connected by I/O Modules.

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 instructions: 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.

### 3.5.2 DMA Transfer

### 3.5.2

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.

### 3.5.3 Dual-Port Memory

3.5.3

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.

### 3.6 Register Array

3.6

The RC3502 processing unit contains a register array of 1024 16-bit words, which is organized as a number of working register sets and an area that is used as a working memory by the micro-machine.

#### 3.6.1 Working Registers

3.6.1

The register array contains 124 sets of working registers. Each set comprises eight 16-bit words.

The first two bits of the 7th register in a register set define a state.

A register set is in one of three states:

- 1) Normal mode; when the register set is used to execute instructions except block input/output instructions.
- 2) Block I/O mode; when the register set is used to execute a block input/output instruction.
- 3) Dummy mode; when the register set is unused.



The structure of the register set in normal mode is as follows:

|    |                             |
|----|-----------------------------|
| LM | maximum stack displacement  |
| PS | flag bits                   |
| PB | global frame pointer base   |
| LU | last used byte pointer disp |
| SF | local frame pointer disp    |
| PR | global frame pointer disp   |
| IB | instruction pointer base    |
| IC | instruction pointer disp    |

The register set consists of a number of memory references, a word for administration of the stack overflow check, and a word containing various flag bits. The memory references are closely connected with the RC3502 implementation of the Real-Time Pascal program and process incarnation concepts.

There are three references to the process incarnation stack:

- global frame pointer (PB, PR)
- local frame pointer (PB, SF)
- last used byte pointer (PB, LU)

There is also a reference to the current instruction in the program being executed by the process incarnation:

- instruction pointer (IB, IC)

Each register set is associated with an interruption level. Level 0 is always set. Each time an instruction (or step in an interruptable instruction) is executed, the register set associated with the interruption level having the highest priority is selected as the current register set, and the instruction pointed out by this register set is executed as the next instruction (or step in an interruptable instruction) in the runtime environment defined by the register set, program, and process incarnation stack (see further chapter 4).

The LM word in the register set is used in conjunction with run-time checking of stack overflow, which is performed automatically by the microprogram. If the execution of an instruction will cause stack overflow, the process incarnation is activated instead in its exception routine (see section 13.2).

The PS word in the register set contains a number of flag bits. The structure of this word is as follows:

|   |   |   |   |   |   |   |   |   |   |    |     |    |    |     |     |
|---|---|---|---|---|---|---|---|---|---|----|-----|----|----|-----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11  | 12 | 13 | 14  | 15  |
| X | S | X | X | X | X | X | W | A | I | T  | RSM | X  | TO | EOI | OVF |

X Not used.

S Statistics (reserved).

WAIT Used by multiple wait instructions to define the cause of waiting and the stack contents.

bit 10: interrupt wait cause

bit 9: semaphore wait cause

bit 8: timer wait cause

bit 7: write control version of interrupt wait

RSM Resume. Used by set instructions to indicate that execution has been suspended by an interrupt with a higher priority and that the stack format is special.

TO Timeout. Indicates timeout and prohibits clear-interrupt functions in I/O instructions.

EOI End of information. Indicates whether the last I/O read/write data function received an EOI status from a device.

OVF Overflow. Used to control the arithmetic overflow exception. A 1 bit causes suppression of this exception.

In Real-Time Pascal notation, a register set is a variable of register\_set\_type defined by the following type declarations:

```
word = 0..65535;
```

```
bit = 0..1;
```

```
nibtype = PACKED RECORD      <* this type is used in normal mode *>
    mode : 0..3;             <* mode = 0 *>
    unused : 0..31;
    onebit : bit;            <* one *>
    memkind : bit;           <* zero for external memory,
                              one for internal *>
    module : 0..31;          <* module number *>
    zerobit : bit;           <* zero *>
END;
```

```
bibtype = PACKED RECORD      <* used in block I/O mode *>
    mode : 0..3;             <* mode = 2 *>
    unused : 0..15;
    const : 0..7;            <* const = 6 *>
    instr : 0..127;          <* instruction value *>
END;
```

```
dibtype = PACKED RECORD      <* used in dummy mode *>
    mode : 0..3;             <* mode = 3 *>
    rem : 0..16383;          <* rem = 16383 *>
END;
```

```
pstype = PACKET RECORD
    unused : bit;
    statistics : bit;
    unused : 0..31;
    wait : 0..15;
    resume : bit;
    unused : bit;
    to : bit;
    eoi : bit;
    ovf : bit;
END;
```

```

register_set_type = RECORD    <* this type is used in normal mode *>
    lm : word;
    ps : pstype;
    pb : basetype; <* see 3.7.1.1 and 4.2.4 *>
    lu : word;
    sf : word;
    pr : word;
    ib : nibtype;
    ic : word;
END;

```

### 3.6.1.2 Block I/O Mode

3.6.1.2

When a register set is in normal mode, and the register set IB, IC points at a block I/O instruction, the following changes are made in that set after instruction fetch:

1st, 2nd, 4th, 5th, 7th, and 8th registers are dumped.

1st register is changed to last + 1.

4th - - - - next addr disp - 1.

5th - - - - next addr base.

7th - - - - bib.

8th - - - - count.

The indication in bib shows that the register set is now in block I/O mode.

An interrupt to a level where the register set is in dummy mode will have the effect:

clear the interrupt

goto instruction, e.g. use of information on bib

execute the I/O code

if the I/O instruction is finished, reload the register set,  
i.e. the register set is set to normal mode again

fetch the next instruction

In Real-Time Pascal, the structure of a register set in block I/O mode notation is as follows:

```

register_set_type = RECORD
    top : word;
    ps : pstype;
    pb : basetype; < * see 3.7.1.1 and 4.2.4 * >
    nadrd : word;
    nadrb : word;
    pr : word;
    ib : bibtype;
    count : word;
END;
```

### 3.6.1.3 Dummy Mode

3.6.1.3

In Real-Time Pascal notation, the structure of a register set in dummy mode is as follows:

```

register_set_type = RECORD
    unused : word;
    unused : word;
    unused : word;
    unused : word;
    unused : word;
    unused : word;
    dib : dibtype;
    unused : word;
END;
```

After autoloading, all register sets are set to dummy mode.

It is also possible to set a register set to dummy mode during the run. A register set in dummy mode belonging to a specific level will perform no instructions. If an interrupt arrives to

such a level greater than zero, the effect will be:

clear the interrupt indication to that level

write the level in the dummy register "dummy intr last" (see subsection 3.6.2.4)

increment the counter in the dummy register "dummy intr last" (see subsection 3.6.2.4)

### 3.6.2 Micromachine Working Area

3.6.2

Besides the 124 register sets, there are 32 registers which are used as a micromachine working area.

The 32 registers may be seen as 4 register sets.

#### 3.6.2.1 Monitor Register Set

3.6.2.1

The address of the first register in this set in the W register array is 3E0. The structure of the register set is as follows:

```

3E0: actq base ; active queue base
3E1: actq disp ; active queue displacement
3E2: n
3E3: m          ; priority selectors
3E4: k
3E5: nxt        ; working register during scheduling
3E6: dlc msb
3E7: dlc lsb    ; dummy loop counter (32 bits)

```

This register set is used to select among coroutines and processes running at level 0.

actq base, actq disp points at an array of addresses of coroutines and processes. n, m, k, and nxt are pointers to the active

queue used to select the next process (see the instructions SCHED, CSTDR, and section 3.7).

dlc msb, dlc lsb is a 32-bit counter, which is incremented by one when the microprogrammed scheduler has found no process to start.

After autoload, the entire register set is initialized with ones. This forces the scheduler to perform a dummy loop. The scheduler is started when the monitor initializes all registers by means of the instruction OCPUT.

In the running system, the scheduler is initialized by SCHED. The use of the pointers is explained in section 3.7.

#### 3.6.2.2 COM8085 Register Set

3.6.2.2

This register set is used for communication between the control microprocessor and the micromachine. It is addressed as 3E8 through 3EF in the W register array.

The contents of the register set are:

|               |                                                                |
|---------------|----------------------------------------------------------------|
| 3E8: fifo56   | ; 6-byte FIFO buffer                                           |
| 3E9: fifo34   | ; used to save parameters in                                   |
| 3EA: fifo12   | ; communication with the<br>control microprocessor             |
| 3EB: COW      | ; 2-byte buffer used by the<br>instruction CWRAM               |
| 3EC: dummy    | ; unused 16-bit register                                       |
| 3ED: messcode | ; code used after parity error<br>message                      |
| 3EE: messbase | ; base and displacement                                        |
| 3EF: messdisp | ; of word with parity error,<br>to the control microprocessor. |

3.6.2.3 Working Register Set

3.6.2.3

This register set is used as a working area by the micromachine. Its use varies according to the instructions that use it. The registers are addressed as 3F0 through 3F7 in the W register array. The process synchronizing instructions use the registers as follows:

|            |                       |
|------------|-----------------------|
| 3F0: ldbb  | ; last base b         |
| 3F1: shptr | ; addr ref variable   |
| 3F2: ldbb  | ; last base a         |
| 3F3: share | ; addr mess           |
| 3F4: ldbb  | ; last displacement b |
| 3F5: shptr | ; addr ref variable   |
| 3F6: ldbb  | ; last displacement a |
| 3F7: share | ; addr mess           |

The registers are defined only within one instruction.

3.6.2.4 Dummy Register Set

3.6.2.4

The addresses of this register set in the W register array are 3F8 through 3FF, and the use of it is as follows:

|                        |                                                                          |
|------------------------|--------------------------------------------------------------------------|
| 3F8: breakp base       | ; address used in breakpoint mode                                        |
| 3F9: breakp disp       |                                                                          |
| 3FA: puerrmsk          |                                                                          |
| 3FB: breakp mode       | ; breakpoint mode                                                        |
| 3FC: dummy intr last   | ; save of level in dummy mode<br>after interrupt                         |
| 3FD: dummy intr count  | ; count of interrupt on level<br>in dummy mode                           |
| 3FE: parity error base | ; address with parity error                                              |
| 3FF: parity error disp | ; saved when the instruction<br>READB or READW detects a<br>parity error |



3.6.2.5 Breakpoint Mode

3.6.2.5

When the machine is in stop mode (after an "s" command to the control microprocessor, see section 17.3), the dummy register set can be used as a breakpoint facility:

if breakp mode (W3FB) is set to a value  $> 0$  then the micromachine will do:

```
repeat
    exec one instruction
    value := value - 1
until value = 0
```

It is also possible to set breakp base, breakp disp to point at an address by using W3F8, W3F9 from the control microprocessor; then:

if breakp mode  $< 0$  then

```
repeat
    exec one instruction
until instruction pointer base = breakp base AND
    instruction pointer disp = breakp disp
```

In this case it is possible:

- 1) either to reset breakpoint mode by resetting breakp mode
- 2) or to execute one or more instructions by means of the "s" command:

then

```
repeat
    exec one instruction
until instruction pointer base = breakp base AND
    instruction pointer disp = breakp disp
```

The tested instruction pointer base and instruction pointer disp is the instruction pointer at the level that is currently running.

### 3.7 Data Formats and Structures

3.7

#### 3.7.1 Basic Formats and Structures

3.7.1

The RC3502 has a number of basic data formats with corresponding data structures on which the instructions operate.

Fundamental to all structures are the quantities 8-bit byte, 16-bit word, and 32-bit double.

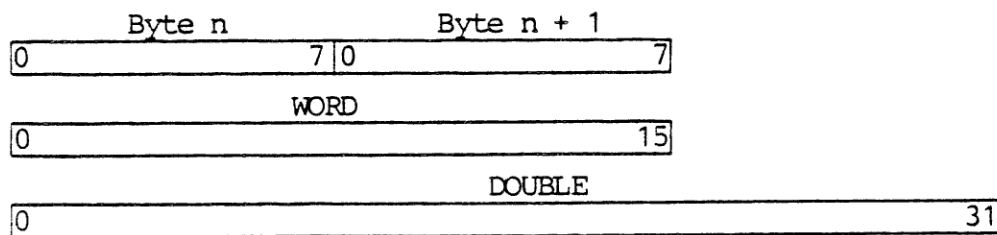


Figure 7: Byte, Word, and Double.

## 3.7.1.1 Addressing

3.7.1.1

Addressing of the memory is done by means of byte addresses, which can be regarded as being numbered in succession from zero. An address occupies two 16-bit words, starting at an even address (a physical word address) and with the following structure:

Address:

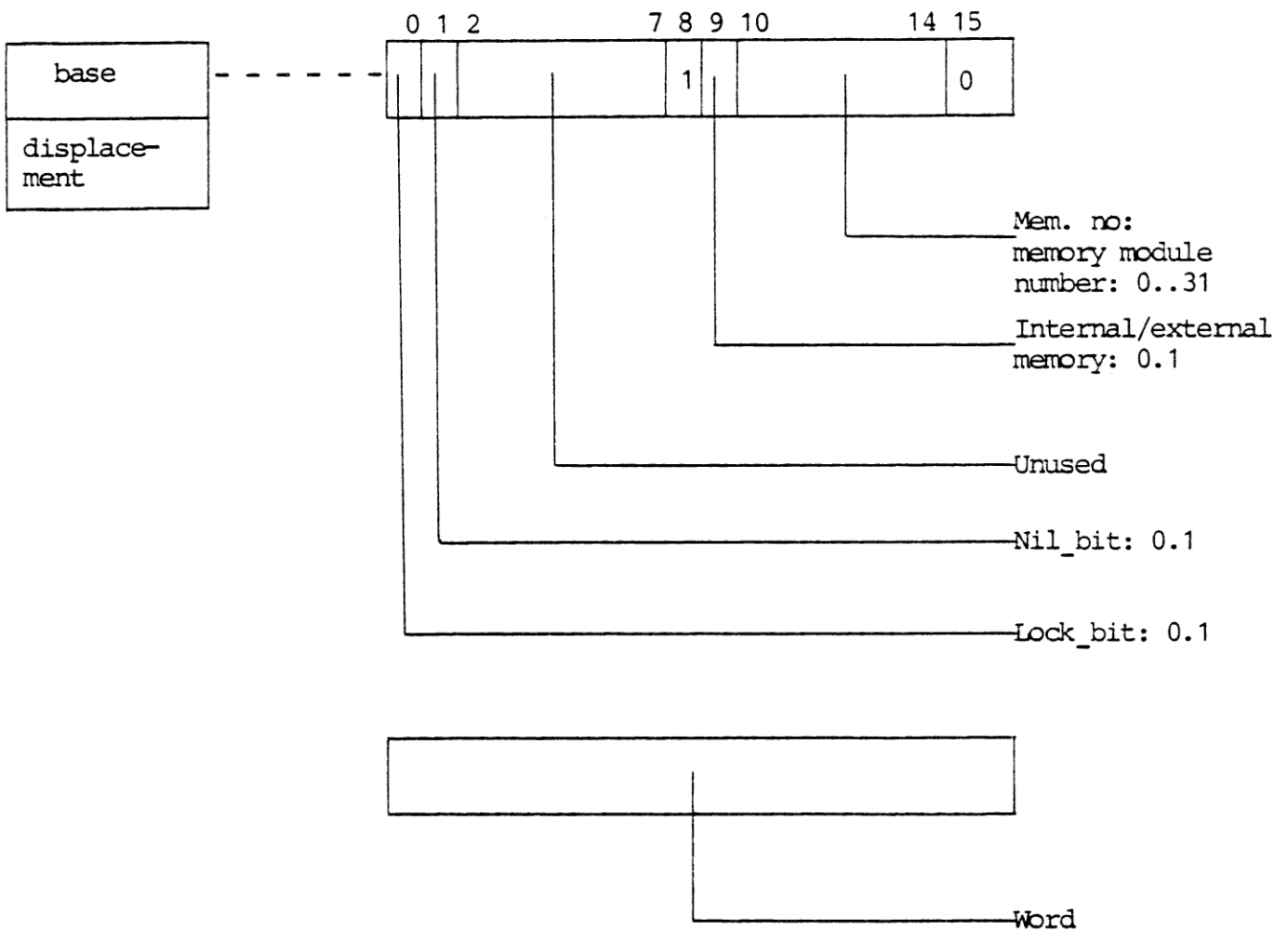


Figure 8: Memory Address.

Nil\_bit and lock\_bit are status bits, used by some of the instructions to ensure data security.

Data structures are addressed by the byte with the lowest address, except the following data structures (system types), which are allocated in the memory to start at an even byte address:

- Incarnation descriptors are addressed by odd byte addresses (see subsection 3.7.3.1).
- Activation records are addressed by odd byte addresses (see subsection 3.7.4).

### 3.7.1.2 Chain and Queue Structures

3.7.1.2

In the RC3502 there are two fundamental chain structures. In connection with queue semaphores and active queues a number of addresses with the following structure are used:

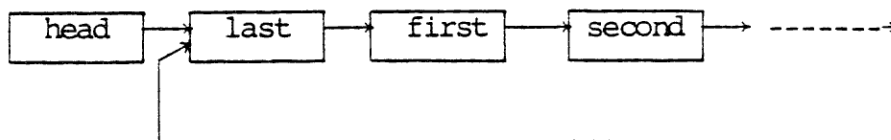


Figure 9: Chain Structure.

Elements are inserted and removed as in a queue (FIFO).

A chain has the following states:

- Empty: Nil-bit in the head has the value 1;
- Chain of messages: Even byte address and nil-bit = 0;
- Chain of incarnations: Odd byte address and nil-bit = 0;

Also used is a single chained linear list of elements containing a pointer field. The offset to the pointer field is chain dependent.

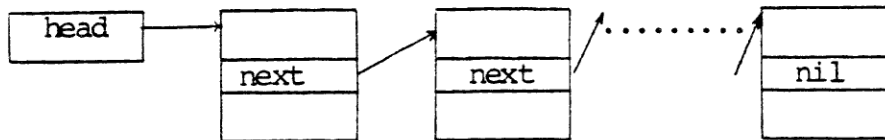


Figure 10: Single Chained Linear List.

### 3.7.1.3 Additional Structures

3.7.1.3

Additional data structures derived from the following functional characteristics of the implementation of Real-Time Pascal on the RC3502:

- Communication
- Dynamic process structure
- Stack oriented execution
- Process execution
- Range testing and indexing

The following gives a short description of the additional structures known by the microprogram (i.e. firmware and hardware).

### 3.7.2 Communication Structures

3.7.2

Process incarnations communicate by means of message passing supported by queue semaphores.

#### 3.7.2.1 Messages

3.7.2.1

A message consists of a message header and a data part (possibly

empty). The format is shown below:

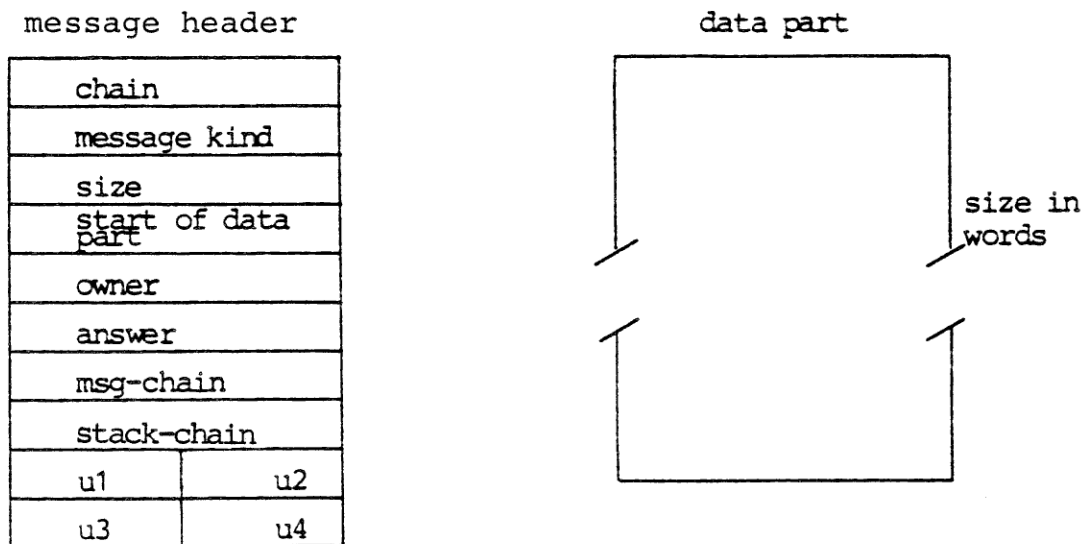


Figure 11: A Message.

The format and meaning of the individual fields are explained by the following Real-Time Pascal notation:

message\_header =

RECORD

```

    chain: addr          ; (* used for chaining the message into
                           semaphore queues *)
    message_kind: word   ; (* either a data message with allocated
                           size  $\geq 0$  or a channel message *)
    size: word           ; (* size in words of the data part *)
    start: addr          ; (* data part address *)
    owner: addr          ; (* address of the owner semaphore *)
    answer: addr         ; (* address of the answer semaphore *)
    message_chain: addr  ; (* chain field for the chain of messages
                           allocated to a process_incarnation *)
    stack_chain: addr    ; (* chain of stacked messages, maintained
                           by push and pop instructions *)
    u1,u2,u3,u4: byte    ; (* user parameters *)
END;    (* message header *)

```

With messages as elements, a stack structure of messages can be formed. (The stack\_chain field is used). The instructions LPUSH and LPOP are used to manipulate this stack (see section 11.12).

### 3.7.2.2 Semaphores

A semaphore is actually a chain head with the status:

- Passive: Empty queue head.
- Open: The chain is a queue of messages.
- Locked: The chain is a queue of waiting process incarnations.

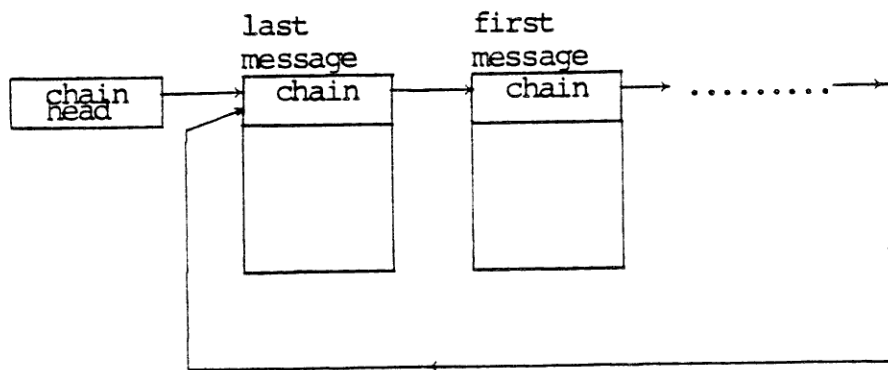


Figure 12: A Semaphore (Open State).

### 3.7.3 The Process Incarnation Stack Descriptor

3.7.3

Several incarnations of a process can be executed. These incarnations are fully dynamic and controlled by a number of data structures, of which the following are so fundamental that single instructions operate on them.

#### 3.7.3.1 Process Incarnation Stack

3.7.3.1

The execution of 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

|                   |                                                                                            |
|-------------------|--------------------------------------------------------------------------------------------|
| chain: addr       | ; (* chain field for semaphores and active queues *)                                       |
| not used: byte    | ;                                                                                          |
| level: byte       | ; (* current interruption level for the incarnation *)                                     |
| incstate: integer | ; (* state may be start, stop *)                                                           |
| msg_waited: addr  | ; (* address of reference; defined only when the incarnation is waiting for a semaphore *) |



```

activequeue: addr      ; (* address of active queue in which
                        ; an incarnation is chained when ac-
                        ; tive *)
chainhead: addr        ; (* address of the queue in which the
                        ; incarnation is currently chained *)
exception_mask: integer ; (* exception mode when a runtime ex-
                        ; ception occurs *)
exception_point: addr  ; (* address of the exception routine
                        ; to be called if an exception
                        ; occurs *)
exic: addr             ; (* instruction pointer base and dis-
                        ; placement stored here prior to en-
                        ; tering the exception routine *)
dumplm: word           ; (* dump location for lm register *)
dumpps: word           ; (* dump location for ps register *)
dumplu: word           ; (* dump location for lu register *)
dumpsf: word           ; (* dump location for sf register *)
entry_point: addr      ; (* dump location for instruction
                        ; pointer *)
timer: integer         ; (* timeout counter used for timeout
                        ; interrupt service *)
maxstack: word         ; (* maximum value of the lu register *)
END;

```

#### 3.7.4 Structures Depending on Stack Oriented Execution

3.7.4

The RC3502 is a pure stack machine, in which processes are executed in connection with an evaluation stack for each incarnation. This evaluation stack is, in fact, placed in continuation of the incarnation descriptor. When a process incarnation is active, the current register set will contain three pointers to the incarnation stack:

- Global frame pointer (odd address), which points out the start of the incarnation descriptor.
- Local frame pointer (odd address), which points out the start of the current frame (i.e. routine) in which expressions are evaluated (activation record).

- Last used pointer, which points out the last used byte on the evaluation stack.

These structures are further described in chapters 4 and 7.

### 3.7.5 Structures Deriving from Process Execution

3.7.5

Each time an instruction has been executed (under certain circumstances, steps in an instruction), the next instruction to be executed is selected. This may involve a switch to a new process to be executed.

The scheduling of process incarnations is done by hardware in three classes with mutual priority:

- Class I: Scheduling by interrupt. The interruption levels are with mutual priority. The incarnations are executed on levels greater than 0.
- Class II: Scheduling of incarnations with internal priority and round robin for a given priority. Class II incarnations are executed on interruption level = 0 (as coroutines).
- Class III: Scheduling of incarnations with internal priority and time-sliced round-robin for a given priority. Class III incarnations are executed on interruption level 0.

Class I scheduling does not cause further data structures. Class II and class III scheduling take place as a co-operation between the SCHED instruction, the communication instructions (signal, wait ...), the instruction fetch, and the contents of the monitor register set.

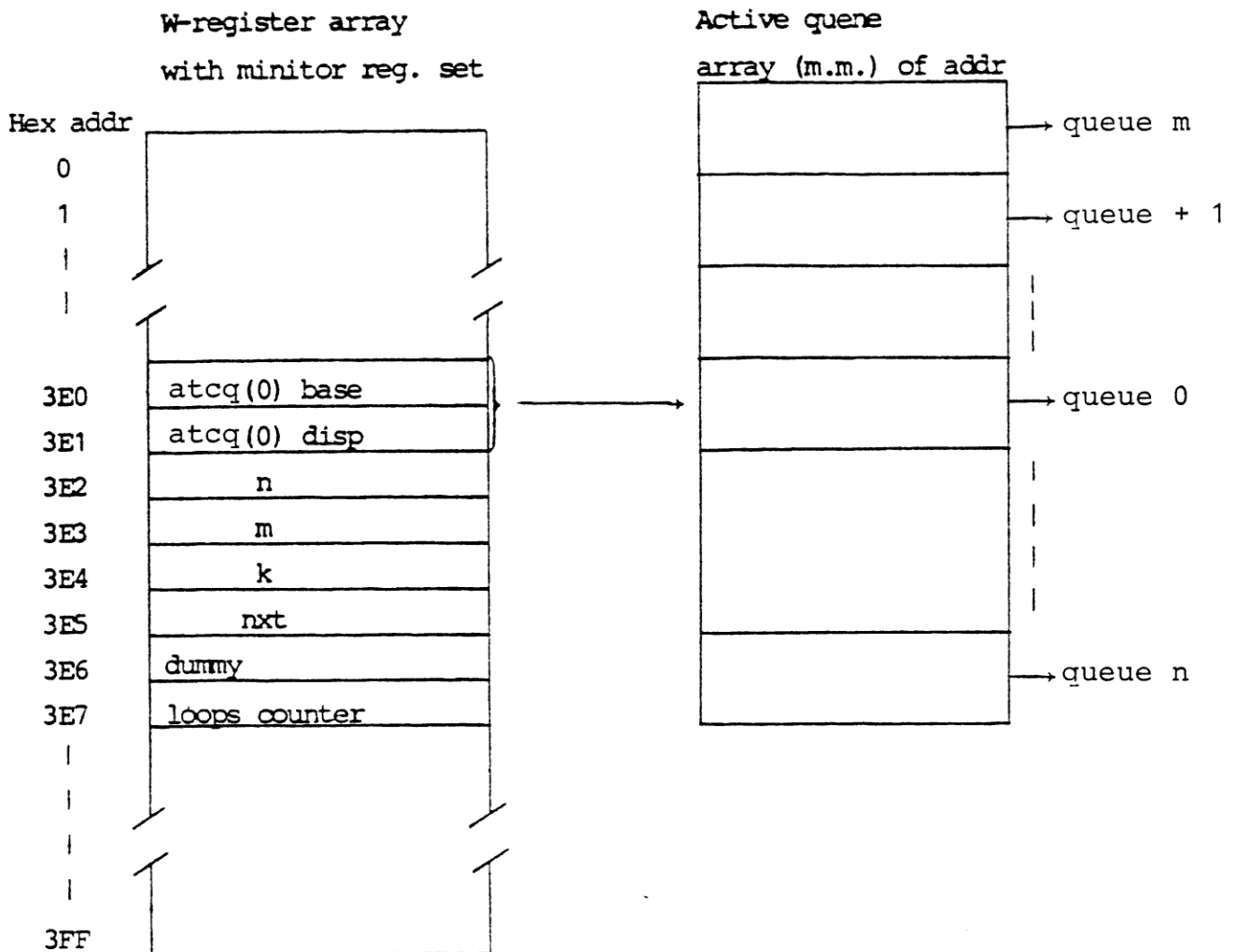


Figure 13: Scheduling Structures.

k in the monitor register set defines that the array of active queues is to be scanned in the following sequence to find the next incarnation:

$n, n-1, n-2, \dots, 0, k, -1, -2, \dots, k+1, k-1, \dots, m$

nxt is a variable used to point out where one is in the above sequence. nxt is reset to n before scheduling. By letting the operating system vary k after the expiration of each time-slice period, one obtains a time-sliced class III scheduling, where priority group  $i$  ( $m \leq i < 0$ ) is guaranteed scheduling at least each  $2^i$  time-slice period.

Queue  $n, n-1, \dots, 0$  is used in the class II scheduling (coroutines). Dummy looping takes place if  $n = -1$  or if  $nxt = m-1$ . Dummy looping will imply incrementation of the dummy loop counter, which should be regarded as an unsigned 32-bit integer. Consult chapter 11 for a description of the instruction SCHED.

3.7.6 Structures Deriving from Range Testing and Indexing

3.7.6

In connection with subrange testing, array indexing, and operations on packed arrays (i.e. load, store), three data structures have been defined:

- Range descriptor (INTRS)
- Dope vector (INDEX)
- Packed dope vector (INPRS, INPSS)

These structures are described in chapter 12.

3.8 Number Representation

3.8

The RC3502 has signed, unsigned, and modulo 64K arithmetic on 16-bit words. Used unsigned or modulo 64K, a word is regarded as an integer in the range:

$$0 \leq \text{unsigned integer word} \leq 65535$$

Used signed, a word is regarded as an integer in the range:

$$-32768 \leq \text{integer word} \leq 32767$$

Signed arithmetic is in twos complement form with bit 0 as the sign bit, with 0 for positive integers and 1 for negative integers.

Range violation as a consequence of signed or unsigned arithmetic operations will result in an exception indicating arithmetic overflow/underflow.

Modulo 64K arithmetic operations will not cause such an exception.

#### 4. BASE INSTRUCTION SET

4.

As described in section 2.3, the Base Instruction Set, or BIS, specifies the functionality of the instruction set for the RC3502. The BIS contains redundancy in the sense that a sequence of instructions may have a single instruction as its substitute.

#### 4.1 Runtime Environment

4.1

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).

3. Local objects

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

To facilitate access to objects in the stack, the register sets of all process incarnations include the following three registers:

1. Global frame pointer (GF)

This pointer contains the address of the second byte of the stack frame at the bottom of the stack (i.e. the stack frame which belongs to the outermost block level of the process). The first part of this stack frame is the incarnation descriptor.

2. Local frame pointer (LF)

This pointer contains the address of the second byte of the anonymous parameters in the stack frame at the top of the stack (i.e. the stack frame which belongs to the latest uncompleted routine call of the process).

3. Last used byte pointer (LP)

This pointer contains the address of the last used byte in the stack.

Expressions are evaluated on top of the stack of frames.

The stack for a process incarnation may be illustrated as follows:

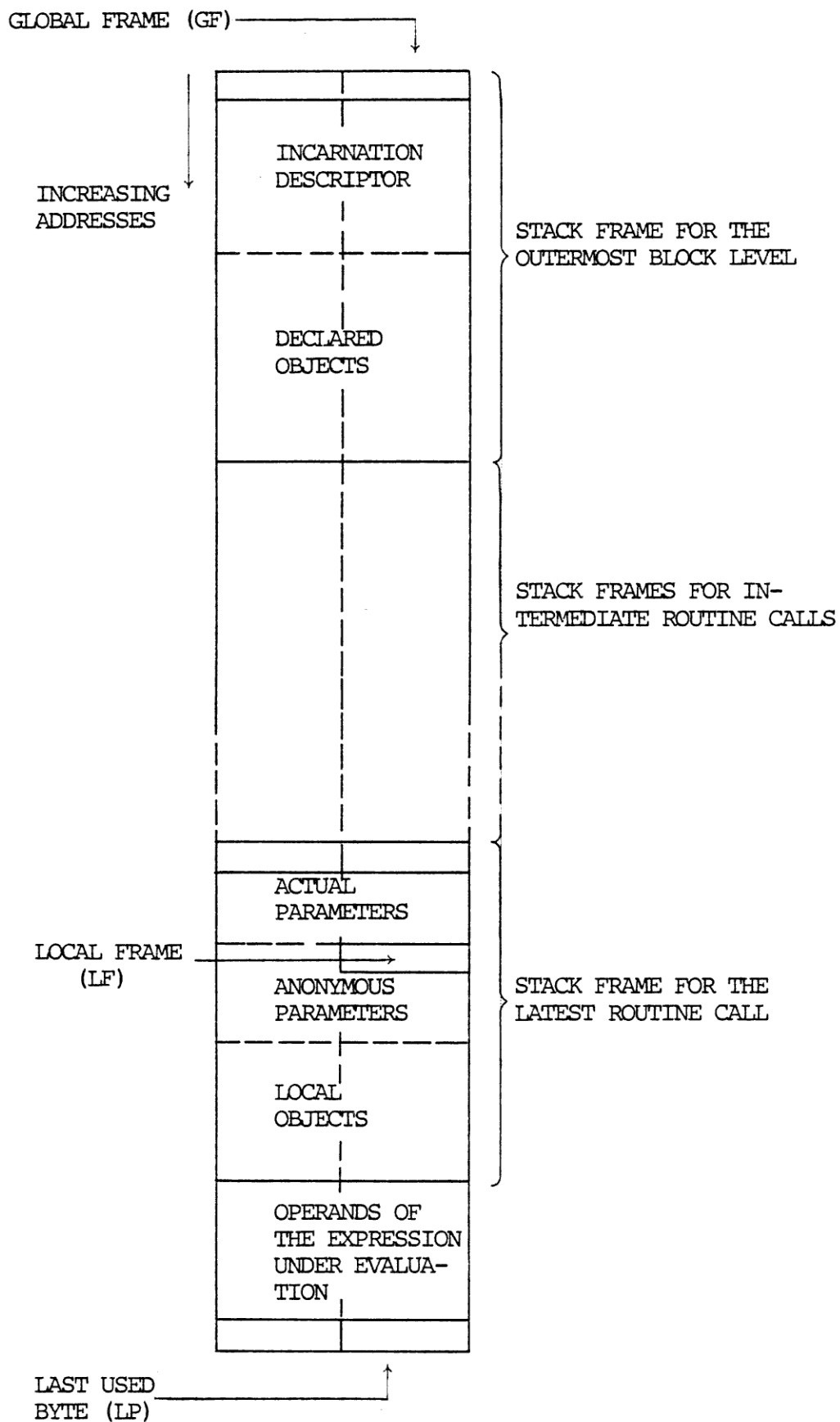


Figure 14: Process Incarnation Stack.

## 4.2 Terminology

4.2

### 4.2.1 Operation and Parameter

4.2.1

An instruction consists of an operation (an 8-bit byte) and zero, one, or more parameters. A parameter may follow the operation code or be placed elsewhere, e.g. on the top of the stack.

### 4.2.2 Operand and Result

4.2.2

The operation works on operands, and when the instruction is executed, a result may be produced.

### 4.2.3 Effective Address

4.2.3

Operands may be retrieved in several different ways, but fundamental to them all is the obtaining of an effective address to point out the operand.

### 4.2.4 Address

4.2.4

An address, or ADDR, is a byte address, which is defined in Real-Time Pascal notation by

TYPE

```

    bit  = 0..1;
    word = 0..65535;
    addr = RECORD
        base : basetype;
        disp : word;
    END;
```

basetype = PACKED RECORD

```

    lockbit : bit;
    nilbit  : bit;
    unused  : 0..63;
    onebit  : bit;      (* always one *)
    memkind : bit;      (* 0 for external memory *)
                          (* 1 for internal memory *)
    module  : 0..31;
    nullbit : bit;      (* always zero *)
END;
```



#### 4.2.5 Size

4.2.5

The size of an operand (or a result) may be one of the following:

- BYTE, which occupies one byte (eight bits).
- WORD, which occupies two bytes. A WORD is addressed by the byte with the smallest address. The first byte need not have an even address.
- DOUBLE, which occupies four bytes. A DOUBLE is addressed by the first byte in its first word (i.e. the word with the smallest address). The first byte need not have an even address.
- FIELD, which occupies a number of bits within a word. A FIELD is specified by a parameter of one byte, which follows the parameters used to describe the access path for the operand (or the result). This byte is interpreted as two 4-bit quantities, specifying, respectively, the first and the last bit occupied by the field.

#### 4.2.6 Addressing

4.2.6

Several different schemes are used to obtain the effective address in order to retrieve an operand (or store a result).

- Direct operand

The operand is placed immediately after the operation code, and the effective address is obtained by means of the instruction pointer (IP).

- Operand on top of the stack

The operand is placed on the top of the stack, and the effective address is obtained by means of the last used byte pointer (LP).

- Direct address

The effective address is formed by the four bytes following the operation code, interpreted as an ADDR.

- Relative to the instruction pointer

The two bytes following the operation code are interpreted as an unsigned integer. The effective address is obtained by adding this integer to the displacement field (ip.disp) of the instruction pointer (IP). A carry is not transferred to the base field (ip.base), but a 16-bit wraparound is used.

- Relative to the local frame

A number (i) of bytes (e.g. two) following the operation code are interpreted as an unsigned integer. The effective address is obtained by adding this integer to the displacement field (lf.disp) of the local frame pointer (LF). A carry is ignored.

- Relative to the global frame

A number (i) of bytes (e.g. two) following the operation code are interpreted as an unsigned integer. The effective address is obtained by adding this integer to the displacement field (gf.disp) of the global frame pointer (GF). A carry is ignored.

- Relative to an intermediate frame

The byte that follows the operation code is interpreted as a level difference, i.e. the difference between the static level of the routine which is executing the instruction and the static level of the routine in which the operand is declared as an object. This difference is used to retrieve a value, called the intermediate frame pointer, which is a pointer to the stack frame in which the object is allocated.

A number (i) of succeeding bytes (e.g. two) are interpreted as an unsigned integer. The effective address is obtained by adding this integer to the displacement field (if.disp) of the intermediate frame pointer (IF). A carry is ignored.

If the level difference is zero, the intermediate frame pointer (IF) will have the same contents as the local frame pointer (LF).

- Relative to an address on top of the stack

The four bytes on the top of the stack are removed and interpreted as an ADDR.

A number (i) of bytes (e.g. two) following the operation code are interpreted as an unsigned integer. The effective address is obtained by adding this integer to the displacement field (addr.disp) of the address removed from the stack. A carry is ignored.

### 4.3 Functional Description

4.3

#### 4.3.1 Structure

4.3.1

Chapters 5 to 13 contain a functional description of the Base Instruction Set. Each of these chapters is divided into one or more groups of instructions with uniform basic functions. For each such group the following information is given:

- The basic function of the instructions in the group.
- A verbal description of the instructions in the group.
- A diagrammatic description of each instruction and its function.
- An algorithmic description on the macro level of each instruction and its function.

Diagrammatic Descriptions

Abbreviations are used for most of the addressing schemes explained in subsection 4.2.6, and certain letters are used in the symbolic names of the instructions to indicate how the effective address of the operand is obtained.

| Addressing Scheme                    | Diagram | Fourth Letter of Name |
|--------------------------------------|---------|-----------------------|
| direct operand:                      | (none)  | H                     |
| operand on top of stack              | (none)  | P                     |
| direct address:                      | $DA_i$  | A                     |
| relative to instruction pointer:     | $IP_i$  | R                     |
| relative to local frame:             | $LF_i$  | L                     |
| relative to global frame:            | $GF_i$  | G                     |
| relative to intermediate frame:      | $IF_i$  | I                     |
| relative to address on top of stack: | $SA_i$  | S                     |
| address of top of stack:             | (none)  | X                     |

Algorithmic Descriptions

The addresses which are fundamental to the instructions are abbreviated as follows:

```

ip : addr; (* instruction pointer *)
lp : addr; (* last used byte pointer *)
lf : addr; (* local frame pointer *)
gf : addr; (* global frame pointer *)
ms : addr; (* maxstack (last byte in allocated stack) *)

```

## Arithmetic

Operations on byte pointers (i.e. ADDR's) are performed as if they were 32-bit integers, but the operations are performed modulo 64K, i.e. only the lower 16 bits (the displacement fields) are involved.

In the algorithms, the memory is accessed in three different ways:

### 1) Accessed as bytes

The symbolic name "membyte" is used with the interpretation

membyte = ARRAY (addr) OF byte

### 2) Accessed as words

The symbolic name "memword" is used with the interpretation

memword = ARRAY (addr) OF word

when the memory is accessed in word mode and the address of the first byte takes either an even or an odd value.

The symbolic name "mem" is used with the interpretation

mem = ARRAY (addr) OF word

when the memory is accessed in word mode, but the address of the first byte is always even.

The process incarnation stack is a part of memory, starting at an even byte address and always occupying an even number of bytes during evaluation on top of the stack. To facilitate the reading of the notation,

stack (lp - 1)

is used as a pseudonym for

mem (lp - 1).

Note that the stack pointers are always odd, and the operands on top of the stack are always accessed in "mem" mode by the micro-program.

3) Accessed as doubles

The symbolic name "memaddr" is used with the interpretation

memaddr = ARRAY (addr) OF addr

when the memory is accessed in double word mode, and the address of the first byte in the first word takes either an even or an odd value.

When an operand of size BYTE is moved to a destination of size WORD, the operand is converted to size WORD with ZERO extension. In the algorithms, the operator "word" is used to indicate the conversion.

4.3.3 Routines

4.3.3

In the following chapters some common routines are used in the algorithmic description.

4.3.3.1 Exception Routine

4.3.3.1

This routine performs an unconditional jump to the program point defined in the exception point field of the incarnation descriptor. Before the jump, the multiple-wait flags are cleared in the ps register of the register set and the maxstack pointer reset to the value defined in the incarnation descriptor.

```

PROCEDURE exception(cause: integer);
BEGIN
  mem(gf+exception_mask):= cause;
  memaddr(gf+exic):= ip;
  ip:= memaddr(gf+exception_point);
  ms.disp:= mem(gf+maxstack);
  ps.wait:= 0;
  GOTO fetch
END; (* exception *)

```

4.3.3.2 Get Intermediate Frame Routine

4.3.3.2

This routine is used to retrieve the address of the intermediate frame. The level difference between the static levels is used.

```
FUNCTION get_if(level: byte): addr;  
VAR  
    res: addr;  
BEGIN  
    res:= lf;  
    WHILE level > 0 DO  
        BEGIN  
            res:= memaddr(res-1);  
            level:= level-1;  
        END;  
    get_if:= res  
END; (* get_if *)
```

## 5. RETRIEVAL OF A VALUE

5.

### 5.1 Push Nonsense (Reserve Stack Space)

5.1

The operand is retrieved, and a result which occupies  $\langle \text{operand} \rangle$  bytes is pushed on the stack. The contents of the result are undefined.

#### 5.1.1 RENPB

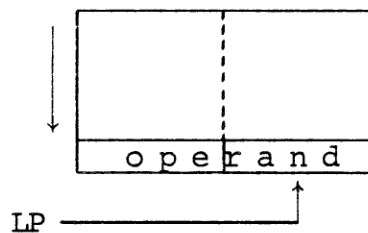
5.1.1

Value:  $8E_{\text{Hex}}$ 

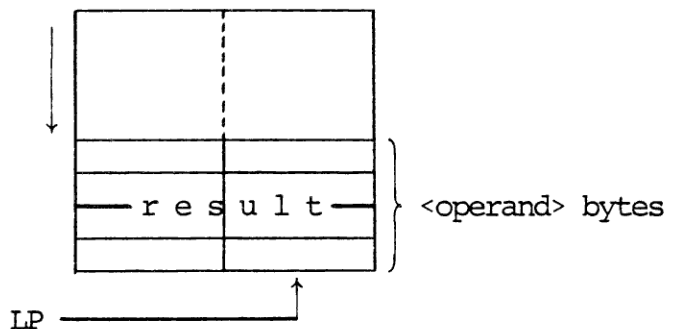
REtrieve Nonsense via P(ush Down List) Bytes

IP  $\rightarrow$  RENPB

STACK BEFORE:



STACK AFTER:



(\* RENPB \*)

VAR

operand : word;

BEGIN

operand := stack(lp-1);

IF odd(operand) THEN exception(odd operand);

IF lp - 2 > ms - operand THEN exception(stack overflow);

lp := lp - 2 + operand;

ip := ip + 1

END; (\* RENPB \*)



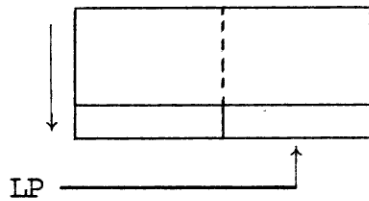
5.1.2 RENHB

5.1.2

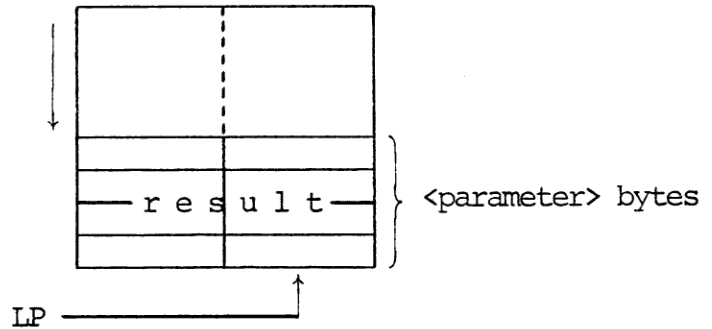
REtrieve Nonsense H(ere) ByteValue: 8F<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| RENHB | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



(\* RENHB \*)

VAR

parameter : word;

BEGIN

parameter := memword(ip+1);

IF odd(parameter) THEN exception(odd parameter);

IF lp &gt; ms - parameter THEN exception(stack overflow);

lp := lp + parameter;

ip := ip + 3

END; (\* RENHB \*)

5.2 Push Constant

5.2

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

5.2.1 RECHW

5.2.1

Retrieve Constant H(ere) Word

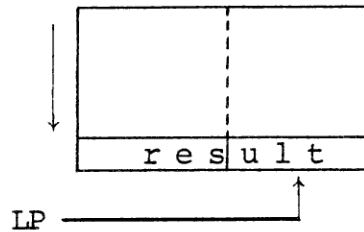
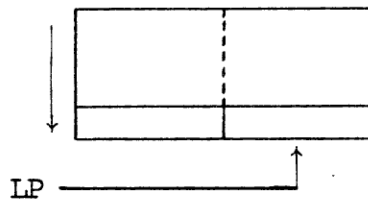
Value:  $AB_{Hex}$

IP → 

|       |           |
|-------|-----------|
| RECHW | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:



(\* RECHW \*)

BEGIN

IF lp > ms - 2 THEN exception(stack overflow);

stack(lp+1) := memword(ip+1);

lp := lp + 2

ip := ip + 3

END; (\* RECHW \*)

## 5.2.2 RECHD or REAAD

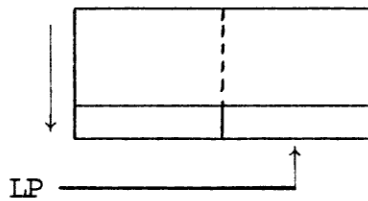
5.2.2

REtrieve Constant H(ere) Double      Value:  $E8_{Hex}$   
REtrieve Address A(bsolute) Double  
 (Two names for the same instruction)

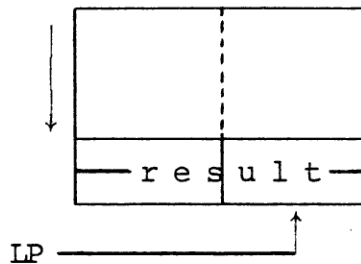
IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| RECHD | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:



STACK AFTER:



(\* RECHD or REAAD \*)

BEGIN

```

  IF lp > ms - 4 THEN exception(stack overflow);
  stack(lp+1) := memword(ip+1);
  stack(lp+3) := memword(ip+3);
  lp := lp + 4;
  ip := ip + 5
END; (* RECHD or REAAD *)
```

5.3 Push Address

5.3

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

5.3.1 REAXD

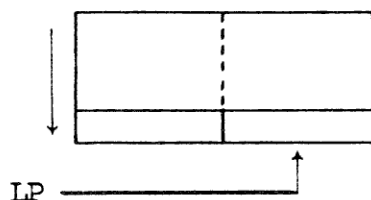
5.3.1

REtrieve Address X (path) Double

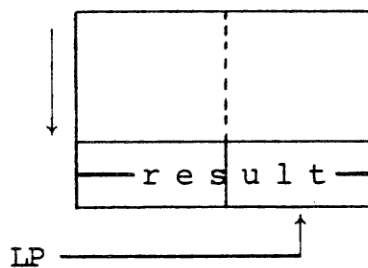
Value:  $FD_{Hex}$

IP → REAXD

STACK BEFORE:



STACK AFTER:



(\* REAXD \*)

BEGIN

```

  IF lp > ms - 4 THEN exception(stack overflow);
  stack(lp+1) := lp.base;
  stack(lp+3) := lp.disp;
  lp := lp + 4;
  ip := ip + 1
END;    (* REAXD *)
```

## 5.3.2 REARD

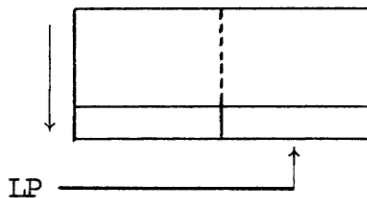
5.3.2

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

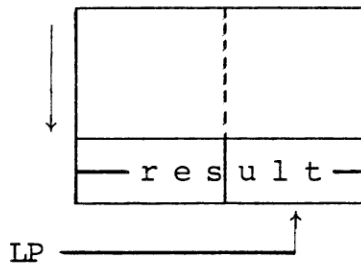
IP → 

|       |           |
|-------|-----------|
| REARD | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



address  
expression  
used:  $IP_2$

The result is the address denoted by  $IP + \text{parameter}$ .

(\* REARD \*)

VAR

param : word;

BEGIN

IF  $lp > ms - 4$  THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := ip.base;

stack(lp+3) := ip.disp + param;

lp := lp + 4;

ip := ip + 3

END; (\* REARD \*)

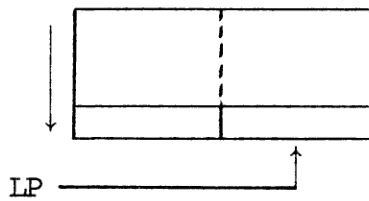
5.3.3 REALD

5.3.3

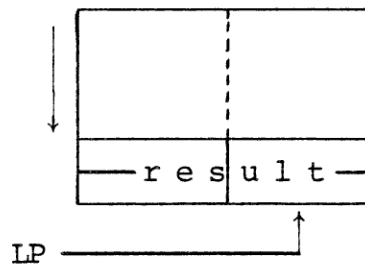
REtrieve Address L(ocal) DoubleValue: E4<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| REALD | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:

address  
expression  
used: LF<sub>2</sub>

(\* REALD \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := lf.base;

stack(lp+3) := lf.disp + param;

lp := lp + 4;

ip := ip + 3

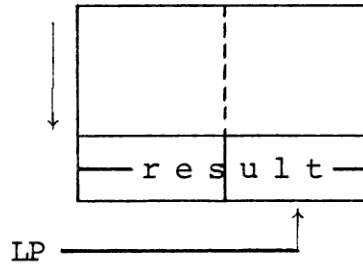
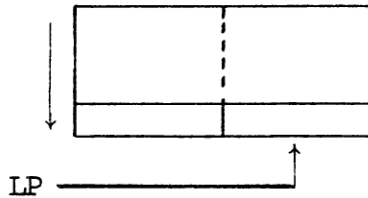
END; (\* REALD \*)

REtrieve Address G(lobal) DoubleValue:  $EO_{Hex}$ IP → 

|       |           |
|-------|-----------|
| REAGD | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:

address  
expression  
used:  $GF_2$ 

(\* REAGD \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := gf.base;

stack(lp+3) := gf.disp + param;

lp := lp + 4;

ip := ip + 3

END; (\* REAGD \*)

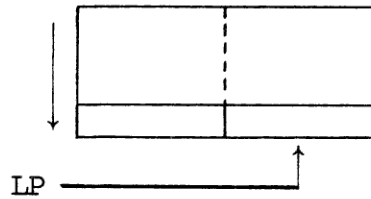
5.3.5 READ

5.3.5

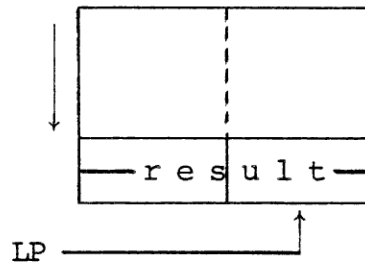
Value: E2<sub>Hex</sub>REtrieve Address I(ntermediate) DoubleIP → 

|      |         |             |
|------|---------|-------------|
| READ | param 1 | parameter 2 |
|------|---------|-------------|

STACK BEFORE:



STACK AFTER:

address  
expression  
used: IF<sub>2</sub>

(\* READ \*)

VAR

param1 : byte;

param2 : word;

FUNCTION get\_if (level : byte) : addr;

EXTERNAL;    (\* to retrieve address of intermediate frame \*)

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow);

param1 := membyte(ip+1);

param2 := memword(ip+2);

stack(lp+1) := get\_if(param1).base;

stack(lp+3) := get\_if(param1).disp + param2;

lp := lp + 4;

ip := ip + 4

END;       (\* READ \*)



## 5.3.6 REASD or UADHW

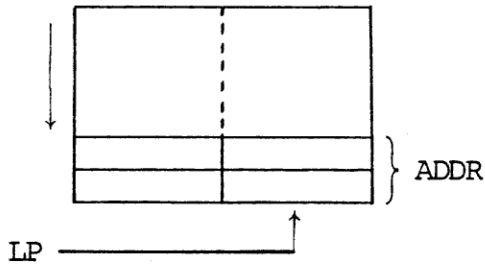
5.3.6

REtrieve Address via S(tack) Double Value: E6<sub>Hex</sub>  
Unsigned ADd H(ere) Word

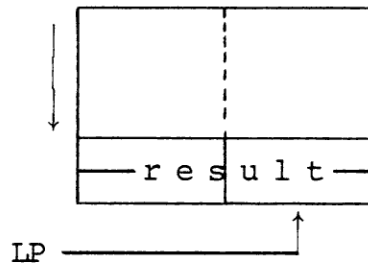
IP → 

|       |           |
|-------|-----------|
| REASD | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



address  
expression  
used: SA<sub>2</sub>

(\* REASD or UADHW \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

stack(lp-1) := stack(lp-1) + param;

ip := ip + 3;

END; (\* REASD or UADHW \*)

5.4 Push Operand

5.4

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

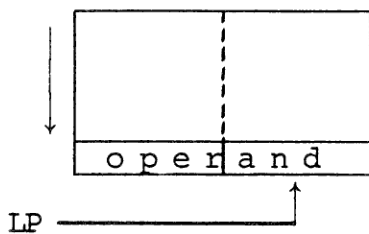
5.4.1 REVPW

5.4.1

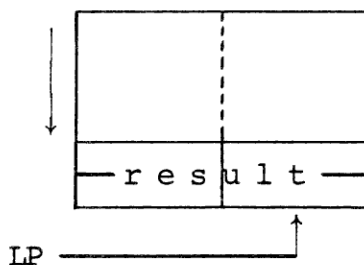
REtrieve Value P(ush Down List) Word Value: AF<sub>Hex</sub>

IP → REVPW

STACK BEFORE:



STACK AFTER:



The WORD on the top of the stack is doubled.

(\* REVPW \*)

BEGIN

```

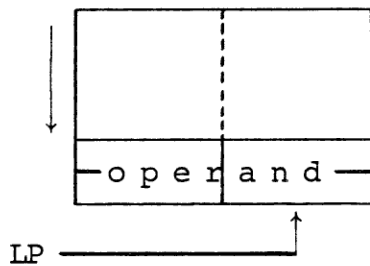
  IF lp > ms - 2 THEN exception(stack overflow);
  stack(lp+1) := stack(lp-1);
  lp := lp + 2;
  ip := ip + 1
END;      (* REVPW *)
```

5.4.2 REVPD

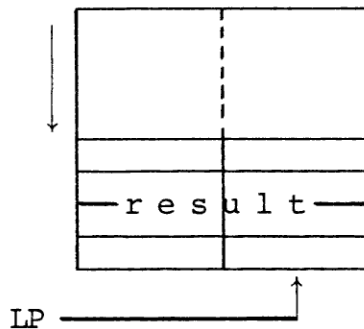
5.4.2

REtrieve Value P(ush Down List) Double Value: EF<sub>Hex</sub>IP → REVPD

STACK BEFORE:



STACK AFTER:



The DOUBLE on  
the top of  
the stack is  
doubled.

(\* REVPD \*)

BEGIN

```

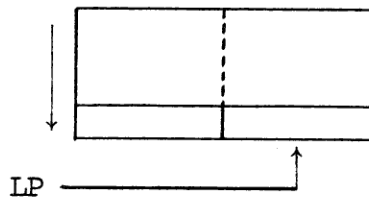
  IF lp > ms - 4 THEN exception(stack overflow);
  stack(lp+1) := stack(lp-3);
  stack(lp+3) := stack(lp-1);
  lp := lp + 4;
  ip := ip + 1
END;      (* REVPD *)

```

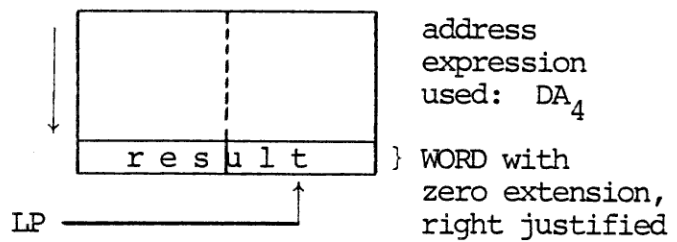
REtrieve Value A(bsolute) ByteValue: 9B<sub>Hex</sub>IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| REVAB | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:



STACK AFTER:



(\* REVAB \*)

VAR

param : addr;

BEGIN

IF lp &gt; ms - 2 THEN exception(stack overflow);

param.base := memword(ip+1);

param.disp := memword(ip+3);

stack(lp+1) := word(membyte(param));

lp := lp + 2;

ip := ip + 5

END; (\* REVAB \*)

## 5.4.4 REVAW

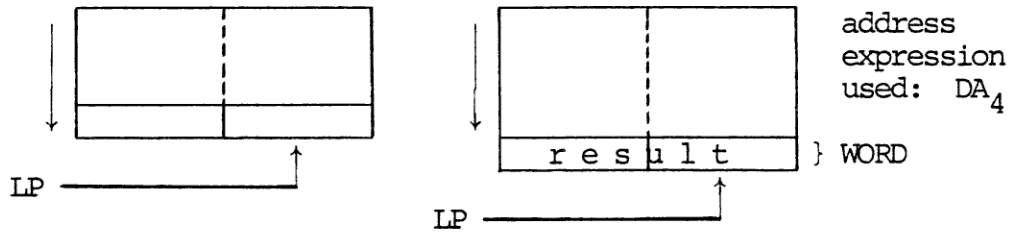
5.4.4

REtrieve Value A(bsolute) WordValue: BB<sub>Hex</sub>IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| REVAW | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:

STACK AFTER:



(\* REVAW \*)

VAR

param : addr;

BEGIN

IF lp &gt; ms - 2 THEN exception(stack overflow);

param.base := memword(ip+1);

param.disp := memword(ip+3);

stack(lp+1) := memword(param);

lp := lp + 2;

ip := ip + 5

END; (\* REVAW \*)

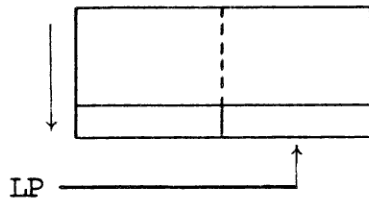
5.4.5 REVAD

5.4.5

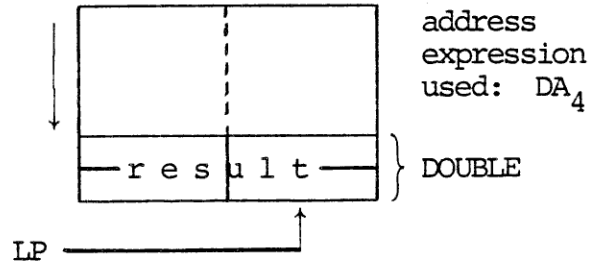
REtrieve Value A(bsolute) DoubleValue:  $FB_{Hex}$ IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| REVAD | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:



STACK AFTER:



(\* REVAD \*)

VAR

param : addr;

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow);

param.base := memword(ip+1);

param.disp := memword(ip+3);

stack(lp+1) := memword(param);

stack(lp+3) := memword(param+2);

lp := lp + 4;

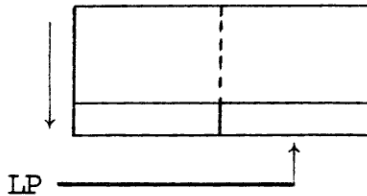
ip := ip + 5

END; (\* REVAD \*)

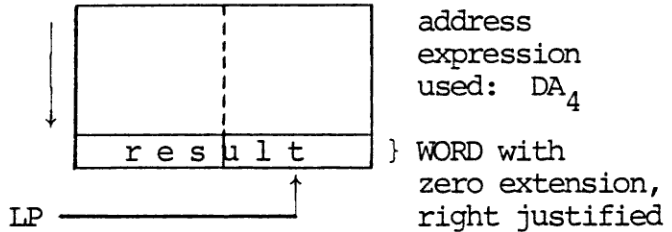
REtrieve Value A(bsolute) FieldValue: DB<sub>Hex</sub>IP → 

|       |   |   |   |   |   |   |   |   |   |   |         |
|-------|---|---|---|---|---|---|---|---|---|---|---------|
| REVAF | p | a | r | a | m | e | t | e | r | 1 | param 2 |
|-------|---|---|---|---|---|---|---|---|---|---|---------|

STACK BEFORE:



STACK AFTER:



(\* REvAF \*)

TYPE

```

fieldtype = PACKED RECORD
    firstbit : 0..15;
    lastbit  : 0..15;
END;
```

VAR

```

param1 : addr;
param2 : fieldtype;
```

BEGIN

```

    param2 := membyte(ip+5);
    IF lp > ms - 2 THEN exception(stack overflow);
    IF param2.firstbit > param2.lastbit THEN exception(field error);
    param1.base := memword(ip+1);
    param1.disp := memword(ip+3);
    stack(lp+1) := (memword(param1) DIV (2**(15-param2.lastbit)))
                   MOD (2**(param2.lastbit-param2.firstbit+1));
    lp := lp + 2;
    ip := ip + 6
END;    (* REvAF *)
```

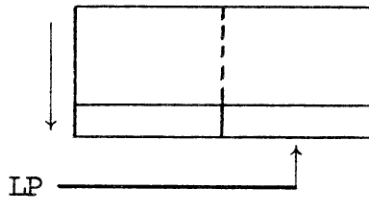
5.4.7 REVLB

5.4.7

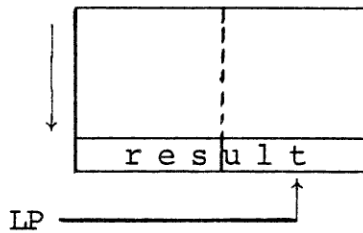
REtrieve Value L(ocal) ByteValue: 97<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| REVLB | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:

address  
expression  
used:  $LF_2$ } WORD with  
zero extension,  
right justified

(\* REVLB \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 2 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := word(membyte(lf+param));

lp := lp + 2;

ip := ip + 3

END; (\* REVLB \*)



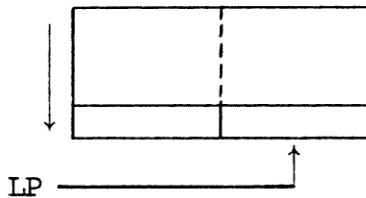
5.4.8 REVLW

5.4.8

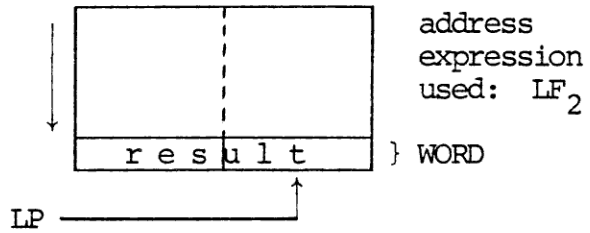
REtrieve Value L(ocal) WordValue: B7<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| REVLW | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



(\* REVLW \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 2 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := memword(lf+param);

lp := lp + 2;

ip := ip + 3

END; (\* REVLW \*)

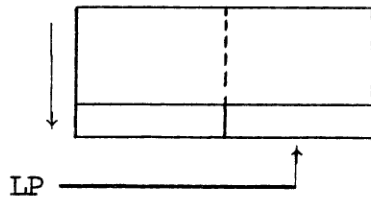
5.4.9 REVLD

5.4.9

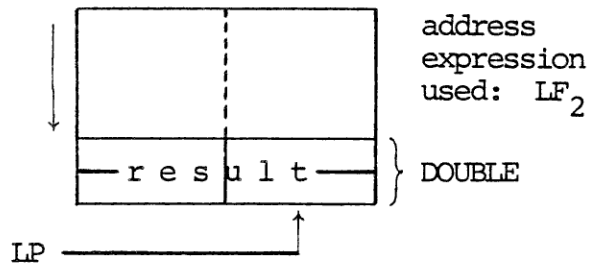
REtrieve Value L(ocal) DoubleValue: F7<sub>Hex</sub>IP → 

|      |           |
|------|-----------|
| REVL | parameter |
|------|-----------|

STACK BEFORE:



STACK AFTER:



(\* REVL D \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := memword(lf+param);

stack(lp+3) := memword(lf+param+2);

lp := lp + 4;

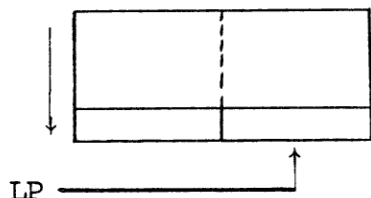
ip := ip + 3

END; (\* REVL D \*)

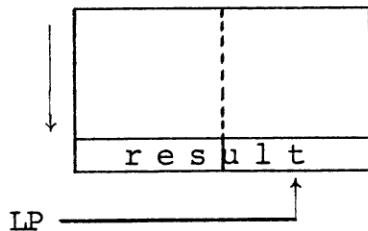
REtrieve Value L(ocal) FieldValue: D7<sub>Hex</sub>IP → 

|      |             |         |
|------|-------------|---------|
| REVL | parameter 1 | param 2 |
|------|-------------|---------|

STACK BEFORE:



STACK AFTER:

address  
expression  
used: LF<sub>2</sub>} WORD with  
zero extension,  
right justified

(\* REVL F \*)

TYPE

```

    fieldtype = PACKED RECORD
        firstbit : 0..15;
        lastbit  : 0..15;
    END;

```

VAR

```

    param1 : word;
    param2 : fieldtype;

```

BEGIN

```

    param2 := membyte(ip+3);
    IF lp > ms - 2 THEN exception(stack overflow);
    IF param2.firstbit > param2.lastbit THEN exception(field error);
    param1 := memword(ip+1);
    stack(lp+1) := (memword(lf+param1) DIV (2**(15-param2.lastbit))
                   MOD (2**(param2.lastbit-param2.firstbit+1)));

```

```

    lp := lp + 2;

```

```

    ip := ip + 4

```

```

END;      (* REVL F *)

```

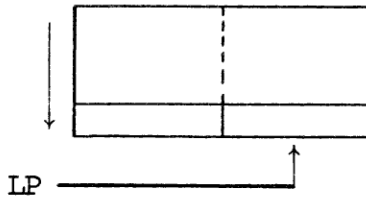
5.4.11 REVGB

5.4.11

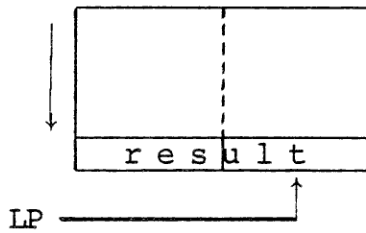
REtrieve Value G(lobal) ByteValue: 93<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| REVGB | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:

address  
expression  
used: GF<sub>2</sub>} WORD with  
zero extension,  
right justified

(\* REVGB \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 2 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := word(membyte(gf+param));

lp := lp + 2;

ip := ip + 3

END; (\* REVGB \*)

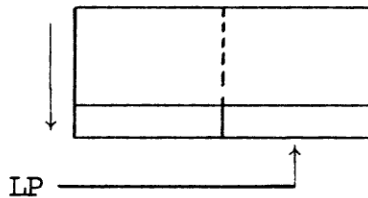
5.4.12 REVGW

5.4.12

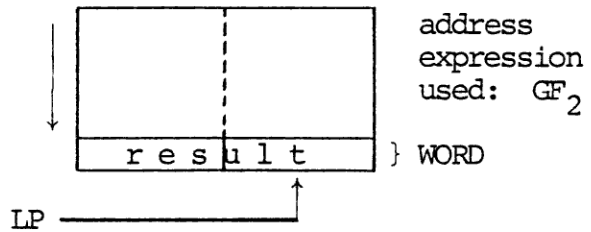
REtrieve Value G(lobal) WordValue: B3<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| REVGW | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



(\* REVGW \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 2 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := memword(gf+param);

lp := lp + 2;

ip := ip + 3

END; (\* REVGW \*)

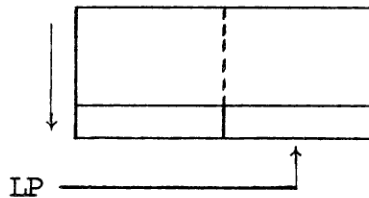
5.4.13 REVGD

5.4.13

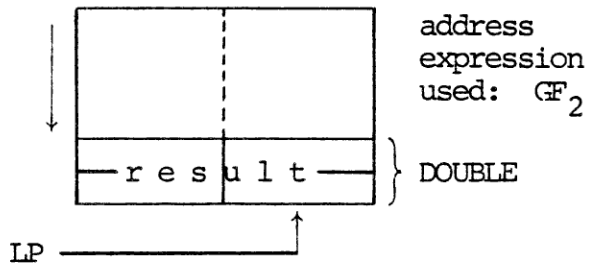
REtrieve Value G(lobal) DoubleValue: F3<sub>Hex</sub>IP → 

|      |           |
|------|-----------|
| REVG | parameter |
|------|-----------|

STACK BEFORE:



STACK AFTER:



(\* REVG \*)

VAR

param : word;

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow);

param := memword(ip+1);

stack(lp+1) := memword(gf+param);

stack(lp+3) := memword(gf+param+2);

lp := lp + 4;

ip := ip + 3

END; (\* REVG \*)

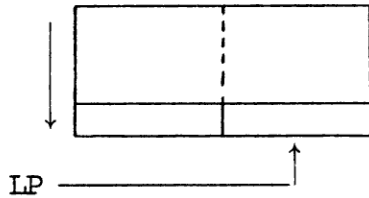
5.4.14 REVGF

5.4.14

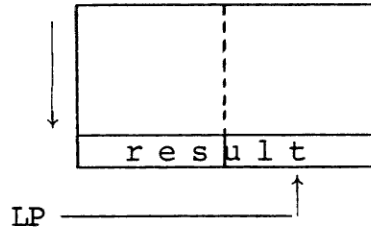
REtrieve Value G(lobal) FieldValue:  $D3_{Hex}$ IP → 

|      |             |         |
|------|-------------|---------|
| REVG | parameter 1 | param 2 |
|------|-------------|---------|

STACK BEFORE:



STACK AFTER:

address  
expression  
used:  $GF_2$ } WORD with  
zero extension,  
right justified

(\* REVG \*)

TYPE

```

    fieldtype = PACKED RECORD
        firstbit : 0..15;
        lastbit  : 0..15;
    END;

```

VAR

```

    param1 : word;
    param2 : fieldtype;

```

BEGIN

```

    param2 := membyte(ip+3);
    IF lp > ms - 2 THEN exception(stack overflow);
    IF param2.firstbit > param2.lastbit THEN exception(field error);
    param1 := memword(ip+1);
    stack(lp+1) := (memword(gf+param1) DIV (2**(15-param2.lastbit)))
                   MOD (2**(param2.lastbit-param2.firstbit+1));

```

lp := lp + 2;

ip := ip + 4

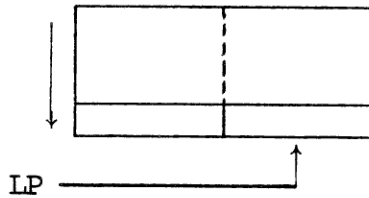
END; (\* REVG \*)

REtrieve Value I(ntermediate) Byte      Value: 95<sub>Hex</sub>

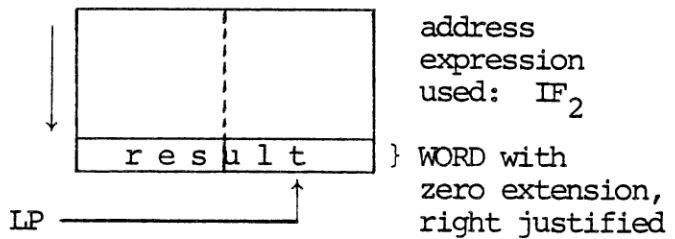
IP → 

|       |         |             |
|-------|---------|-------------|
| REVIB | param 1 | parameter 2 |
|-------|---------|-------------|

STACK BEFORE:



STACK AFTER:



(\* REVIB \*)

VAR

    param1 : byte;

    param2 : word;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL;    (\* to retrieve address of intermediate frame \*)

BEGIN

    IF lp > ms - 2 THEN exception(stack overflow);

    param1 := membyte(ip+1);

    param2 := memword(ip+2);

    stack(lp+1) := word(membyte(get\_if(param1)+param2));

    lp := lp + 2;

    ip := ip + 4

END;            (\* REVIB \*)

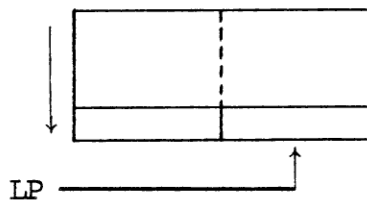


REtrieve Value I(ntermediate) Word      Value: B5<sub>Hex</sub>

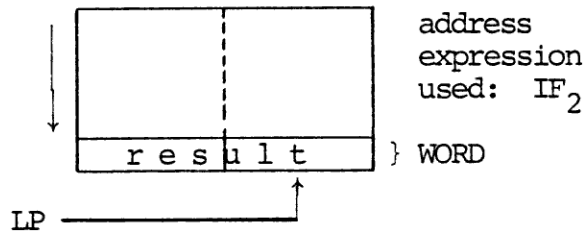
IP → 

|       |         |             |
|-------|---------|-------------|
| REVIW | param 1 | parameter 2 |
|-------|---------|-------------|

STACK BEFORE:



STACK AFTER:



(\* REVIW \*)

VAR

    param1 : byte;

    param2 : word;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL;    (\* to retrieve address of intermediate frame \*)

BEGIN

    IF lp > ms - 2 THEN exception(stack overflow);

    param1 := membyte(ip+1);

    param2 := memword(ip+2);

    stack(lp+1) := memword(get\_if(param1)+param2);

    lp := lp + 2;

    ip := ip + 4

END;            (\* REVIW \*)

## 5.4.17 REVID

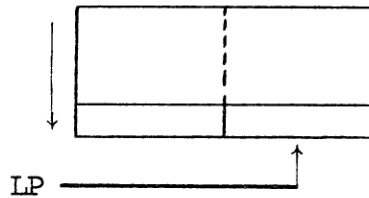
5.4.17

REtrieve Value I(ntermediate) Double Value: F5<sub>Hex</sub>

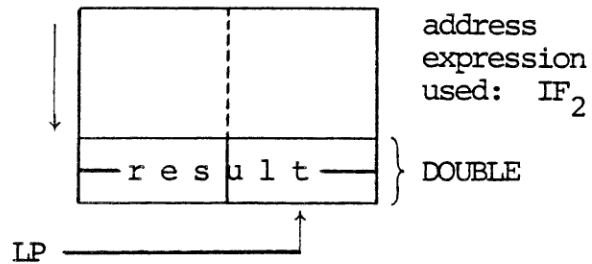
IP → 

|       |         |             |
|-------|---------|-------------|
| REVID | param 1 | parameter 2 |
|-------|---------|-------------|

STACK BEFORE:



STACK AFTER:



(\* REVID \*)

VAR

    param1 : byte;

    param2 : word;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL;    (\* to retrieve address of intermediate frame \*)

BEGIN

    IF lp > ms - 4 THEN exception(stack overflow);

    param1 := membyte(ip+1);

    param2 := memword(ip+2);

    stack(lp+1) := memword(get\_if(param1)+param2);

    stack(lp+3) := memword(get\_if(param1)+param2+2);

    lp := lp + 4;

    ip := ip + 4

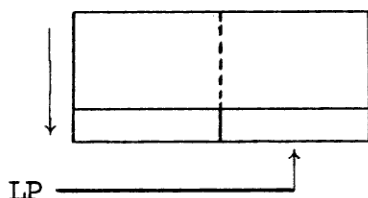
END;           (\* REVID \*)

REtrieve Value I(ntermediate) Field    Value: D5<sub>Hex</sub>

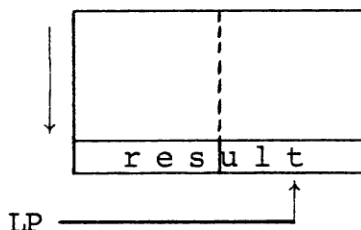
IP → 

|       |         |             |         |
|-------|---------|-------------|---------|
| REVIF | param 1 | parameter 2 | param 3 |
|-------|---------|-------------|---------|

STACK BEFORE:



STACK AFTER:



address  
expression  
used: IF<sub>2</sub>

} WORD with  
zero extension,  
right justified

(\* REVIF \*)

TYPE

```
fieldtype = PACKED RECORD
    firstbit : 0..15;
    lastbit  : 0..15;
END;
```

VAR

```
param1 : byte;
param2 : word;
param3 : fieldtype;
```

FUNCTION get\_if(level : byte) : addr;

EXTERNAL; (\* to retrieve address of intermediate frame \*)

BEGIN

```
param3 := membyte(ip+4);
IF lp > ms - 2 THEN exception(stack overflow);
IF param3.firstbit > param3.lastbit THEN exception(field error);
param1 := membyte(ip+1);
param2 := memword(ip+2);
stack(lp+1) := (memword(get_if(param1)+param2) DIV (2**(15-param3.lastbit)))
                MOD (2**(param3.lastbit-param3.firstbit+1));
```

```
lp := lp + 2;
```

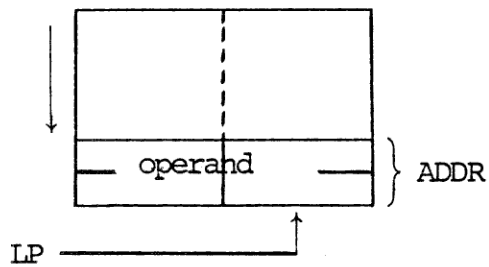
```
ip := ip + 5
```

END; (\* REVIF \*)

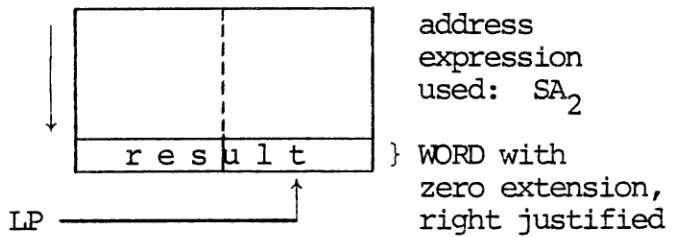
REtrieve Value S(tack) ByteValue: 99<sub>Hex</sub>IP → 

|       |             |
|-------|-------------|
| REVSb | parameter 1 |
|-------|-------------|

STACK BEFORE:



STACK AFTER:



(\* REVSb \*)

VAR

 param1 : word;  
 operand : addr;

BEGIN

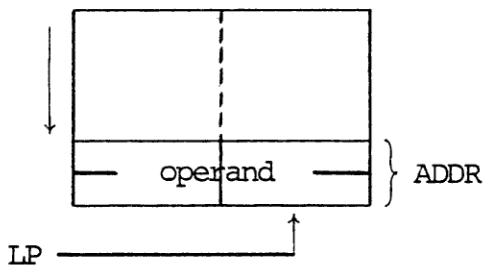
 param1 := memword(ip+1);  
 operand.base := stack(lp-3);  
 operand.disp := stack(lp-1);  
 IF operand.base.nilbit THEN exception(nil exception);  
 stack(lp-3) := word(membyte(operand+param1));  
 lp := lp - 2;  
 ip := ip + 3

END; (\* REVSb \*)

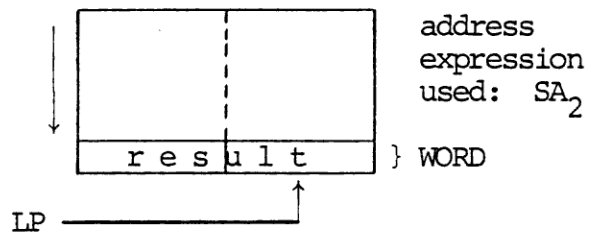
REtrieve Value S(tack) WordValue: B9<sub>Hex</sub>IP → 

|       |             |
|-------|-------------|
| REVSU | parameter 1 |
|-------|-------------|

STACK BEFORE:



STACK AFTER:



(\* REVSU \*)

VAR

 param1 : word;  
 operand : addr;

BEGIN

```

  param1 := memword(ip+1);
  operand.base := stack(lp-3);
  operand.disp := stack(lp-1);
  IF operand.base.nilbit THEN exception (nil exception);
  stack(lp-3) := memword(operand+param1);
  lp := lp - 2;
  ip := ip + 3

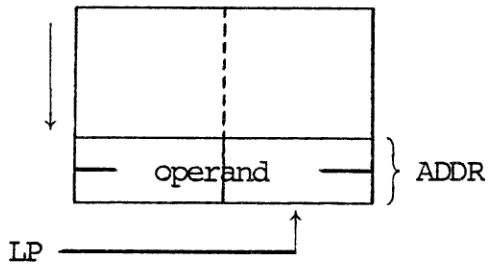
```

END; (\* REVSU \*)

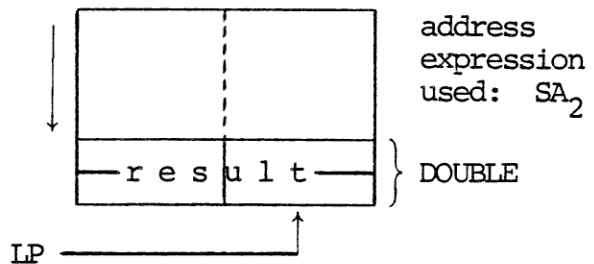
REtrieve Value S(tack) DoubleValue: F9<sub>Hex</sub>IP → 

|       |             |
|-------|-------------|
| REVSD | parameter 1 |
|-------|-------------|

STACK BEFORE:



STACK AFTER:



(\* REVSD \*)

VAR

 param1 : word;  
 operand : addr;

BEGIN

 param1 := memword(ip+1);  
 operand.base := stack(lp-3);  
 operand.disp := stack(lp-1);  
 IF operand.base.nilbit THEN exception (nil exception);  
 stack(lp-3) := memword(operand+param1);  
 stack(lp-1) := memword(operand+param1+2);

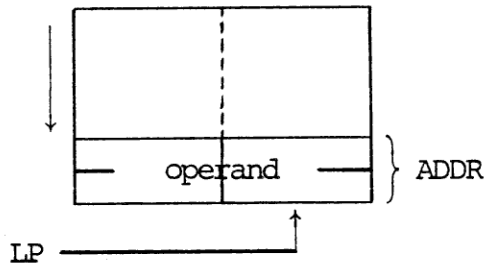
ip := ip + 3

END; (\* REVSD \*)

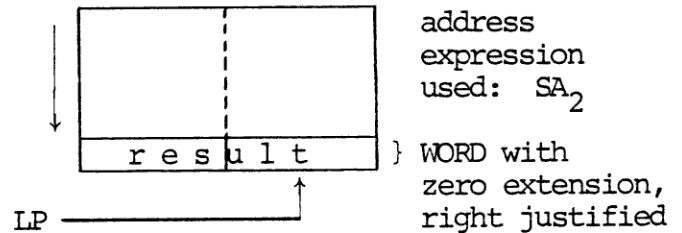
REtrieve Value S(tack) FieldValue: D9<sub>Hex</sub>IP → 

|       |             |         |
|-------|-------------|---------|
| REVSF | parameter 1 | param 2 |
|-------|-------------|---------|

STACK BEFORE:



STACK AFTER:



(\* REVSF \*)

TYPE

```

fieldtype = PACKED RECORD
    firstbit : 0..15;
    lastbit  : 0..15;
END;
```

VAR

```

param1 : word;
param2 : fieldtype;
operand : addr;
```

BEGIN

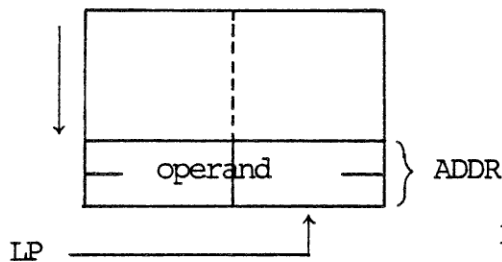
```

    param2 := membyte(ip+3);
    IF param2.firstbit > param2.lastbit THEN exception(field error);
    param1 := memword(ip+1);
    operand.base := stack(lp-3);
    operand.disp := stack(lp-1);
    IF operand.base.nilbit THEN exception (nil exception);
    stack(lp-3) := (memword(operand)+param1) DIV (2**(15-param2.lastbit)))
                  MOD (2**(param2.lastbit-param2.firstbit+1));

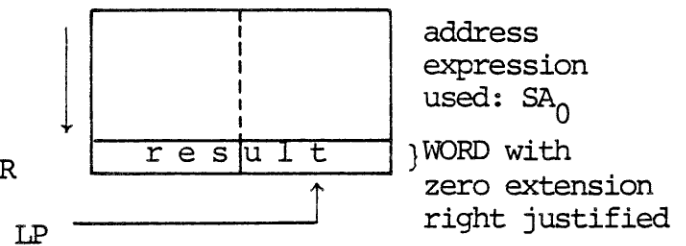
    lp := lp - 2;
    ip := ip + 4
END;    (* REVSF *)
```

READ ByteValue: 90<sub>Hex</sub>IP → READB

STACK BEFORE:



STACK AFTER:



If no module is selected, the instruction will return a zero as the result. If a parity error occurs, the standard parity error procedure is suppressed. The actual result of the reading is delivered, besides the value of 'operand' in the dummy register set. The processing unit continues in run mode.

(\* READB \*)

VAR

```
operand : addr;
result  : word;
```

BEGIN

```
operand.base := stack(lp-3);
operand.disp := stack(lp-1);
result := 0;
result := word(membyte(operand));
```

IF 'parity error' THEN

WITH dummy\_regset.parity\_error\_address DO

BEGIN

base := operand.base;

disp := operand.disp

END;

stack(lp-3) := result;

lp := lp - 2;

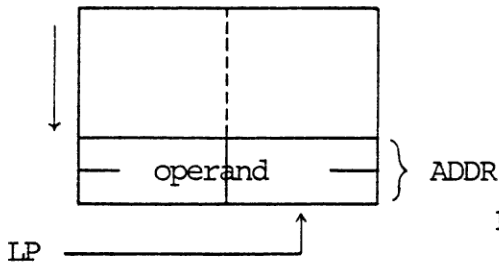
ip := ip + 1

END; (\* READB \*)

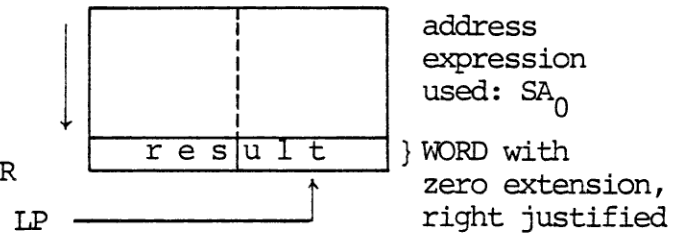


READ WordValue: B0<sub>Hex</sub>IP → READW

STACK BEFORE:



STACK AFTER:



If no module is selected, the instruction will return a zero as the result. If a parity error occurs, the standard parity error procedure is suppressed. The actual result of the reading is delivered, besides the value of 'operand' in the dummy register set. The processing unit continues in run mode.

(\* READW \*)

VAR

```
operand : addr;
result  : word;
```

BEGIN

```
operand.base := stack(lp-3);
operand.disp := stack(lp-1) AND (-2);
result := 0;
result := memword(operand);
IF 'parity error' THEN
  BEGIN
    IF 'right parity error' THEN
      operand.disp := operand.disp + 1;
      WITH dummy_regset.parity_error_address DO
        BEGIN
          base := operand.base;
          disp := operand.disp
        END
      END;
    END;
  END;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
```

END; (\* READW \*)

## 6. STORAGE OF A VALUE

6.

### 6.1 Pop Garbage

6.1

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

#### 6.1.1 STNHB

6.1.1

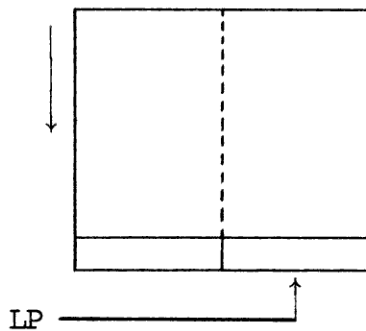
Store Nonsense H(ere) Byte

Value: 8C<sub>Hex</sub>

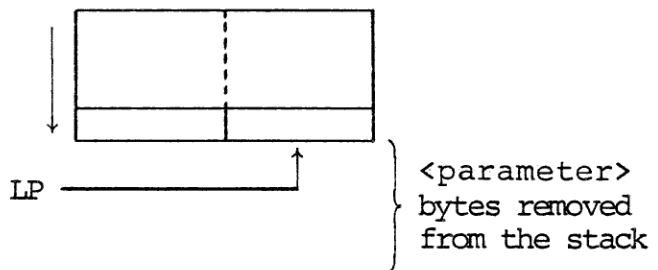
IP → 

|       |           |
|-------|-----------|
| STNHB | parameter |
|-------|-----------|

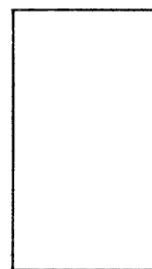
STACK BEFORE:



STACK AFTER:



MEMORY:



No result is stored.

(\* STNHB \*)

VAR

parameter : word;

BEGIN

parameter := memword(ip+1);

IF odd(parameter) THEN exception(odd parameter)

lp := lp - parameter;

ip := ip + 3

END; (\* STNHB \*)

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 STVAB

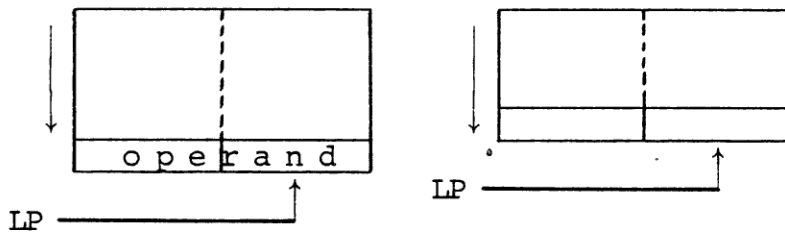
6.2.1

Store Value A(bsolute) ByteValue: 9A<sub>Hex</sub>IP → 

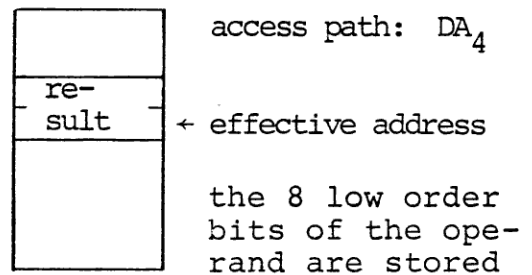
|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| STVAB | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:

STACK AFTER:



MEMORY:



(\* STVAB \*)

VAR

param : addr;

BEGIN

param.base := memword(ip+1)

param.disp := memword(ip+3);

membyte(param) := stack(lp-1) AND 255;

lp := lp - 2;

ip := ip + 5;

END; (\* STVAB \*)

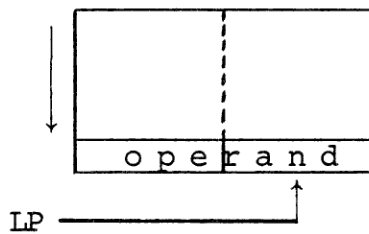
## 6.2.2 STVAW

6.2.2

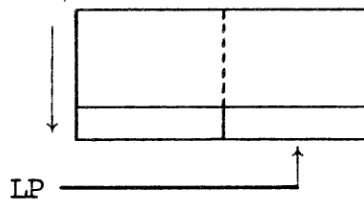
Store Value Absolute) WordValue: BA<sub>Hex</sub>IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| STVAW | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

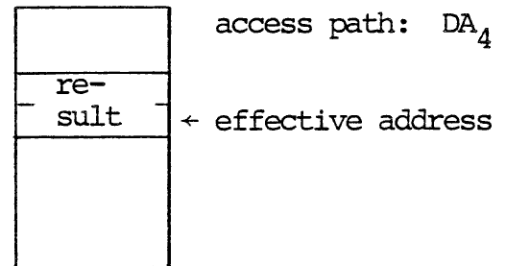
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVAW \*)

VAR

param : addr;

BEGIN

param.base := memword(ip+1);

param.disp := memword(ip+3);

memword(param) := stack(lp-1);

lp := lp - 2;

ip := ip + 5;

END; (\* STVAW \*)

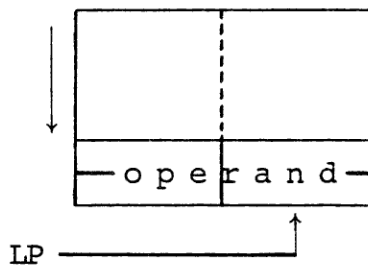
## 6.2.3 STVAD

6.2.3

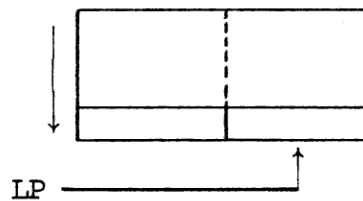
Store Value Absolute) DoubleValue: FA<sub>Hex</sub>IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| STVAD | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

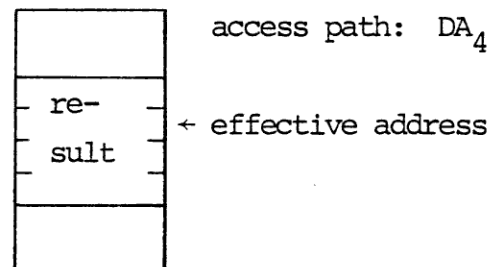
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVAD \*)

VAR

param : addr;

BEGIN

param.base := memword(ip+1)

param.disp := memword(ip+3);

memword(param) := stack(lp-3);

memword(param+2) := stack(lp-1);

lp := lp - 4;

ip := ip + 5;

END;     (\* STVAD \*)

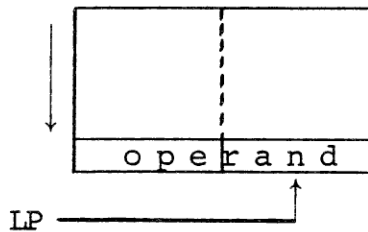
## 6.2.4 STVAF

6.2.4

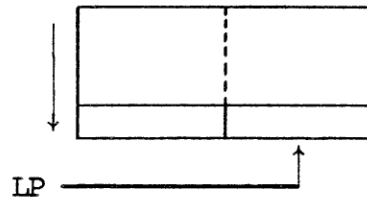
Store Value Absolute) FieldValue:  $DA_{Hex}$ IP → 

|       |   |   |   |   |   |   |   |   |   |   |         |
|-------|---|---|---|---|---|---|---|---|---|---|---------|
| STVAF | p | a | r | a | m | e | t | e | r | 1 | param 2 |
|-------|---|---|---|---|---|---|---|---|---|---|---------|

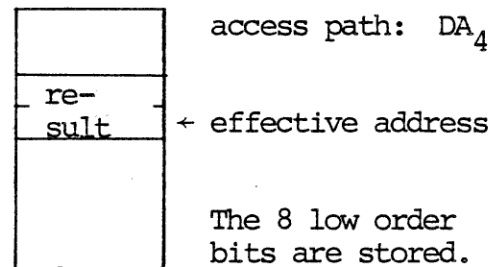
STACK BEFORE:



STACK AFTER:



MEMORY:



```

(* STVAF *)

TYPE
  fieldtype = PACKED RECORD
    firstbit : 0..15;
    lastbit  : 0..15;
  END;

VAR
  param1 : addr;
  param2 : fieldtype;
  operand : word;
  result  : word;
  mask    : word;

BEGIN
  param1.base := memword(ip+1);
  param1.disp := memword(ip+3);
  param2 := membyte(ip+5);
  WITH param2 DO
    BEGIN
      IF firstbit > lastbit THEN
        exception(field error);
      operand := stack(lp-1);
      mask := 1 shift(lastbit+1-firstbit)-1;
      IF mask < operand THEN (* unsigned *)
        exception(field overflow);
      mask := mask shift(15-lastbit);
      operand := operand shift(15-lastbit)
    END;
    result := memword(param1);
    result := result AND (NOT mask)
    memword(param1) := result OR operand;
    lp := lp - 2;
    ip := ip + 6
  END; (* STVAF *)

```

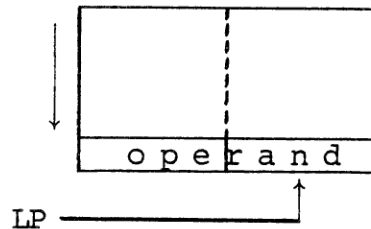
## 6.2.5 STVLB

6.2.5

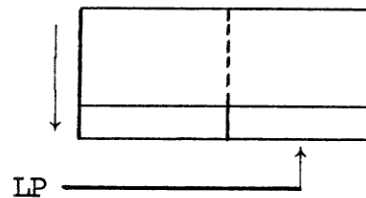
Store Value L(ocal) ByteValue: 96<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| STVLB | parameter |
|-------|-----------|

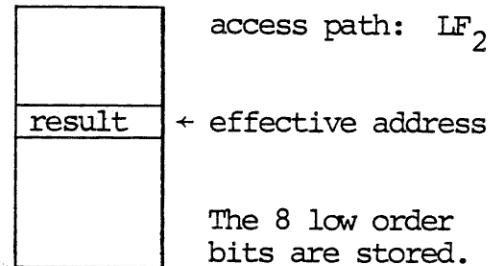
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVLB \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

membyte(lf+param) := stack(lp-1) AND 255;

lp := lp - 2;

ip := ip + 3

END; (\* STVLB \*)



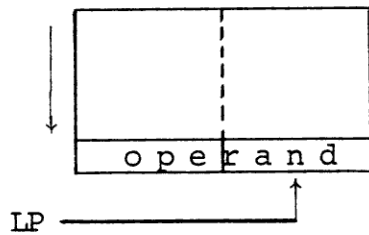
## 6.2.6 STVLW

6.2.6

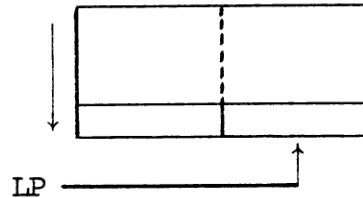
Store Value L(ocal) WordValue: B6<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| STVLW | parameter |
|-------|-----------|

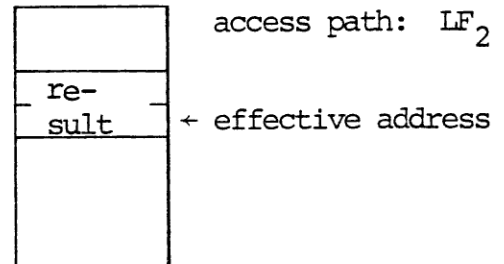
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVLW \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

memword(lf+param) := stack(lp-1);

lp := lp - 2;

ip := ip + 3

END; (\* STVLW \*)

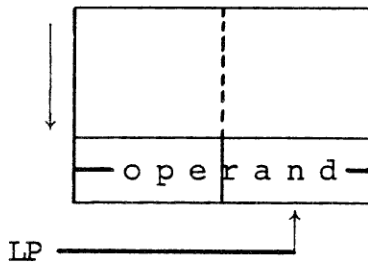
6.2.7 STVLD

6.2.7

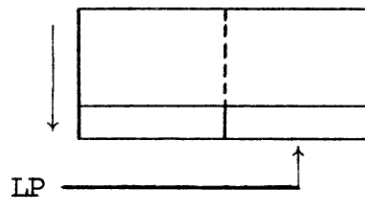
STore Value L(ocal) DoubleValue: F6<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| STVLD | parameter |
|-------|-----------|

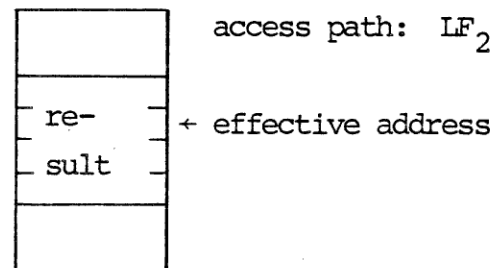
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVLD \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

memword(lf+param) := stack(lp-3);

memword(lf+param+2) := stack(lp-1);

lp := lp - 4;

ip := ip + 3

END;     (\* STVLD \*)

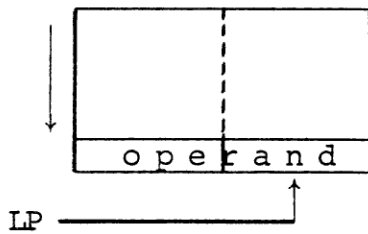
6.2.8 STVLF

6.2.8

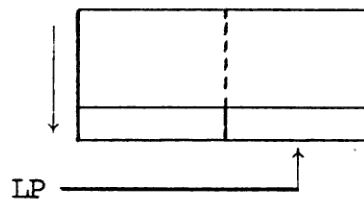
Store Value L(ocal) FieldValue:  $D6_{Hex}$ IP → 

|       |             |         |
|-------|-------------|---------|
| STVLF | parameter 1 | param 2 |
|-------|-------------|---------|

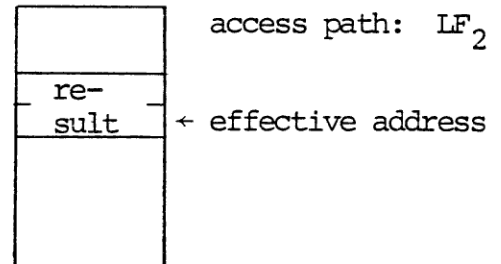
STACK BEFORE:



STACK AFTER:



MEMORY:



```

(* STVLF *)
TYPE
    fieldtype = PACKED RECORD
        firstbit : 0..15;
        lastbit  : 0..15;
    END;
VAR
    param1 : word;
    param2 : fieldtype;
    operand : word;
    result  : word;
    mask    : word;
BEGIN
    param1 := memword(ip+1);
    param2 := membyte(ip+3);
    WITH param2 DO
        BEGIN
            IF firstbit > lastbit THEN
                exception(field error);
            operand := stack(lp-1);
            mask := 1 shift(lastbit+1-firstbit)-1;
            IF mask < operand THEN (* unsigned *)
                exception(field overflow);
            mask := mask shift(15-lastbit);
            operand := operand shift(15-lastbit)
        END;
        result := memword(1f+param1);
        result := result AND (NOT mask);
        memword(1f+param1) := result OR operand;
        lp := lp - 2;
        ip := ip + 4
    END;
    (* STVLF *)

```

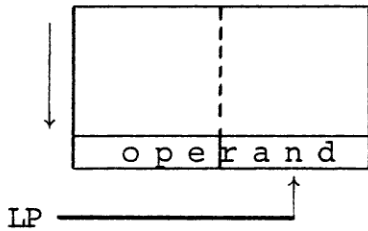
## 6.2.9 STVGB

6.2.9

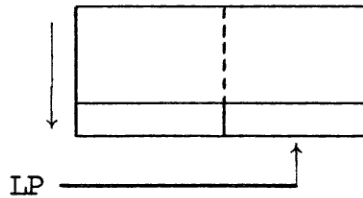
STore Value G(lobal) ByteValue: 92<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| STVGB | parameter |
|-------|-----------|

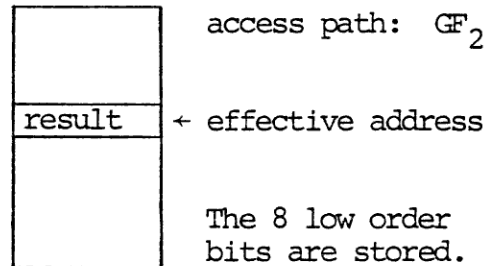
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVGB \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

membyte(gf+param) := stack(lp-1) AND 255;

lp := lp - 2;

ip := ip + 3;

END; (\* STVGB \*)

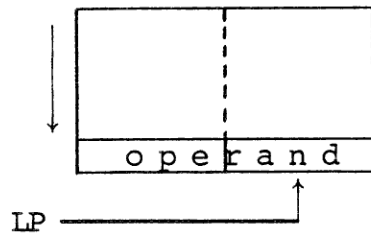
## 6.2.10 STVGW

6.2.10

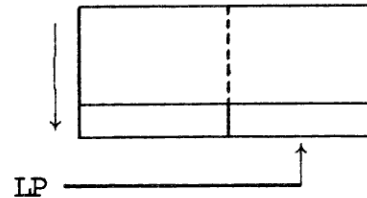
Store Value G(lobal) WordValue:  $B2_{Hex}$ IP → 

|       |           |
|-------|-----------|
| STVGW | parameter |
|-------|-----------|

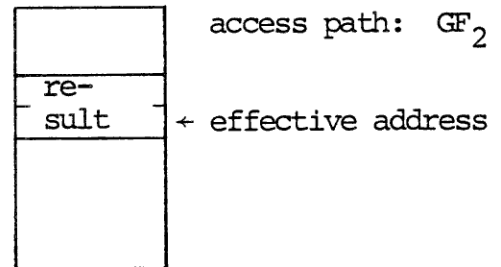
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVGW \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

memword(gf+param) := stack(lp-1);

lp := lp - 2;

ip := ip + 3;

END; (\* STVGW \*)

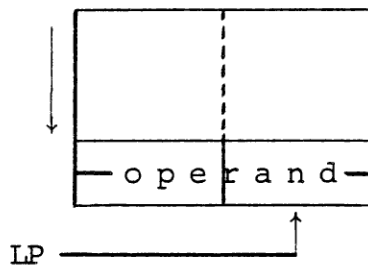
## 6.2.11 STVGD

6.2.11

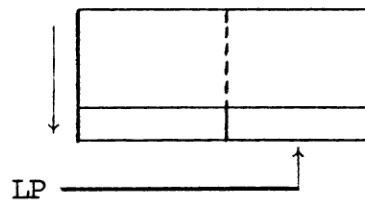
Store Value G(lobal) DoubleValue:  $F2_{Hex}$ IP → 

|       |           |
|-------|-----------|
| STVGD | parameter |
|-------|-----------|

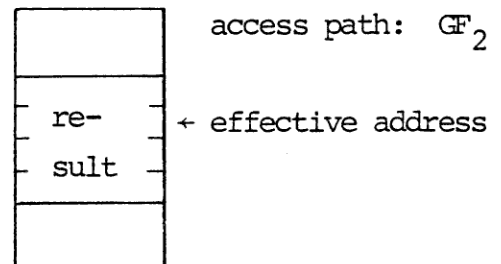
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVGD \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

memword(gf+param) := stack(lp-3);

memword(gf+param+2) := stack(lp-1);

lp := lp - 4;

ip := ip + 3;

END; (\* STVGD \*)

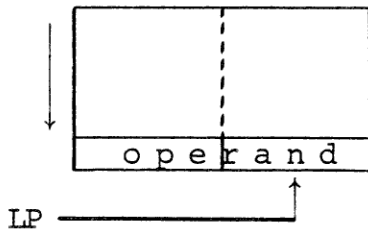
## 6.2.12 STVGF

6.2.12

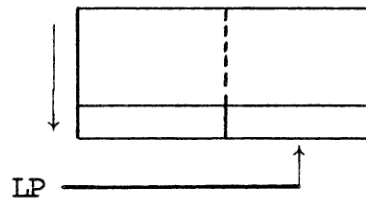
STore Value G(lobal) FieldValue: D2<sub>Hex</sub>IP → 

|       |             |         |
|-------|-------------|---------|
| STVGF | parameter 1 | param 2 |
|-------|-------------|---------|

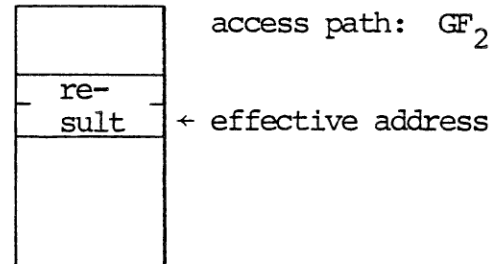
STACK BEFORE:



STACK AFTER:



MEMORY:





```

(* STVGF *)

TYPE
  fieldtype = PACKED RECORD
    firstbit : 0..15;
    lastbit  : 0..15;
  END;

VAR
  param1 : word;
  param2 : fieldtype;
  operand : word;
  result  : word;
  mask    : word;

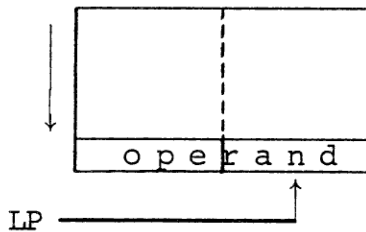
BEGIN
  param1 := memword(ip+1);
  param2 := membyte(ip+3);
  WITH param2 DO
    BEGIN
      IF firstbit > lastbit THEN
        exception(field error);
      operand := stack(lp-1);
      mask := 1 shift(lastbit+1-firstbit)-1;
      IF mask < operand THEN (* unsigned *)
        exception(field overflow);
      mask := mask shift(15-lastbit);
      operand := operand shift(15-lastbit)
    END;
    result := memword(gf+param1);
    result := result AND (NOT mask);
    memword(gf+param1) := result OR operand;
    lp := lp - 2;
    ip := ip + 4
  END; (* STVGF *)

```

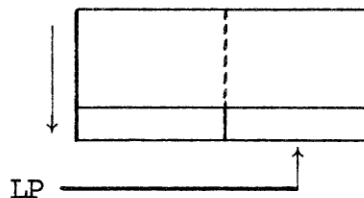
STore Value I(ntermediate) ByteValue: 94<sub>Hex</sub>IP → 

|       |         |             |
|-------|---------|-------------|
| STVIB | param 1 | parameter 2 |
|-------|---------|-------------|

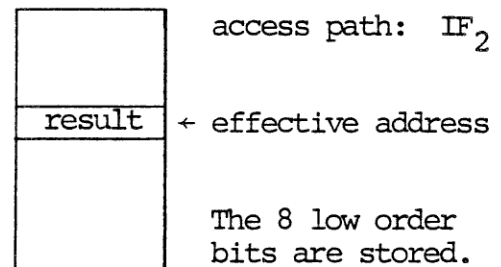
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVIB \*)

VAR

param1 : byte;

param2 : word;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL; (\* to retrieve address of intermediate frame \*)

BEGIN

param1 := membyte(ip+1);

param2 := memword(ip+2);

membyte(get\_if(param1)+param2) := stack(lp-1) AND 255;

lp := lp - 2;

ip := ip + 4;

END; (\* STVIB \*)

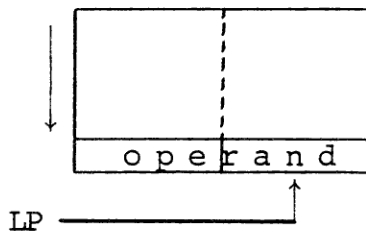
## 6.2.14 STVIW

6.2.14

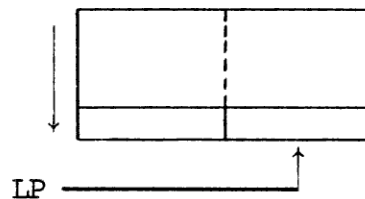
STore Value I(ntermadiate) WordValue: B4<sub>Hex</sub>IP → 

|       |         |             |
|-------|---------|-------------|
| STVIW | param 1 | parameter 2 |
|-------|---------|-------------|

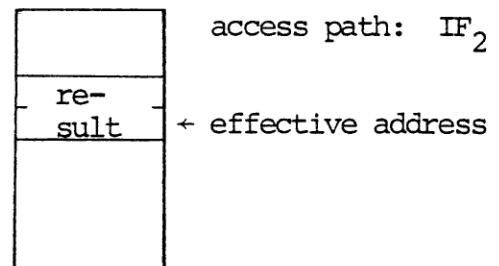
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVIW \*)

VAR

param1 : byte;

param2 : word;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL; (\* to retrieve address of intermediate frame \*)

BEGIN

param1 := membyte(ip+1);

param2 := memword(ip+2);

memword(get\_if(param1)+param2) := stack(lp-1);

lp := lp - 2;

ip := ip + 4;

END; (\* STVIW \*)

## 6.2.15 STVID

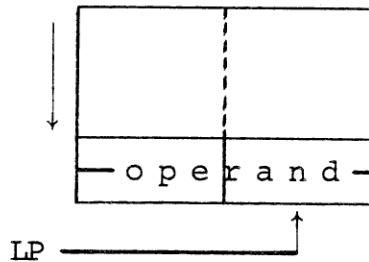
6.2.15

Store Value I(ntermediate) Double      Value: F4<sub>Hex</sub>

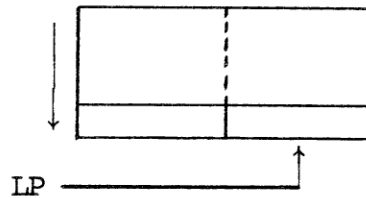
IP → 

|       |         |             |
|-------|---------|-------------|
| STVID | param 1 | parameter 2 |
|-------|---------|-------------|

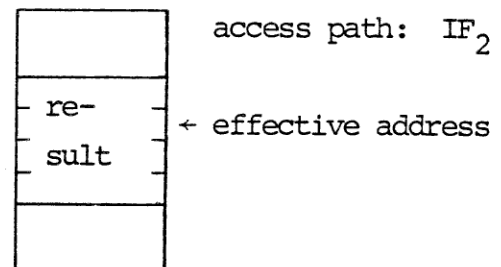
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVID \*)

VAR

    param1 : byte;

    param2 : word;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL;    (\* to retrieve address of intermediate frame \*)

BEGIN

    param1 := membyte(ip+1);

    param2 := memword(ip+2);

    memword(get\_if(param1)+param2) := stack(lp-3);

    memword(get\_if(param1)+param2+2) := stack(lp-1);

    lp := lp - 4;

    ip := ip + 4;

END;       (\* STVID \*)

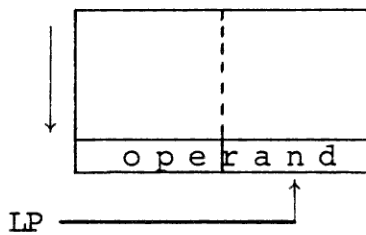
6.2.16 STVIF

6.2.16

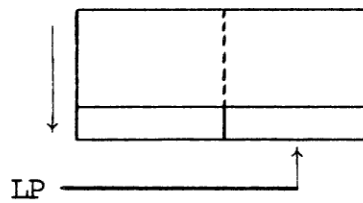
STore Value I(ntermediate) FieldValue:  $D4_{\text{Hex}}$ IP → 

|       |         |             |         |
|-------|---------|-------------|---------|
| STVIF | param 1 | parameter 2 | param 3 |
|-------|---------|-------------|---------|

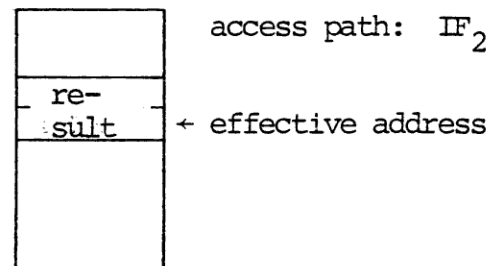
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVIF \*)

TYPE

```
    fieldtype = PACKED RECORD
        firstbit : 0..15;
        lastbit  : 0..15;
    END;
```

VAR

```
    param1 : byte;
    param2 : word;
    param3 : fieldtype;
    operand : word;
    result  : word;
    mask    : word;
```

FUNCTION get\_if(level : byte) : addr;

EXTERNAL; (\* to retrieve address of intermediate frame \*)

BEGIN

```
    param1 := membyte(ip+1);
    param2 := memword(ip+2);
    param3 := membyte(ip+4);
```

WITH param3 DO

BEGIN

```
    IF firstbit > lastbit THEN
        exception(field error);
    operand := stack(lp-1);
    mask := 1 shift(lastbit+1-firstbit)-1;
    IF mask < operand THEN (* unsigned *)
        exception(field overflow);
    mask := mask shift(15-lastbit);
    operand := operand shift(15-lastbit)
```

END;

```
    result := memword(get_if(param1)+param2);
```

```
    result := result AND (NOT mask);
```

```
    memword(get_if(param1)+param2) := result OR operand;
```

```
    lp := lp - 2;
```

```
    ip := ip + 5
```

END; (\* STVIF \*)

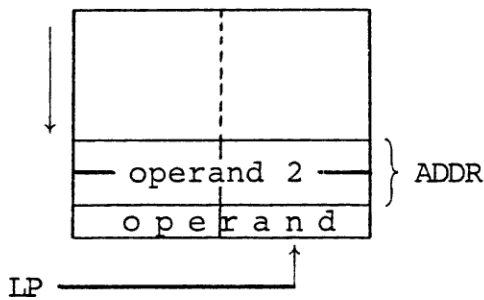
## 6.2.17 STVSB

6.2.17

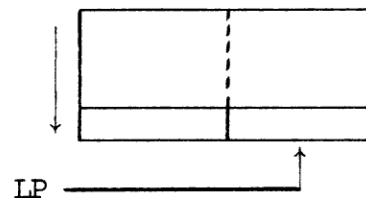
Store Value S(tack) ByteValue: 98<sub>Hex</sub>IP → 

|       |             |
|-------|-------------|
| STVSB | parameter 1 |
|-------|-------------|

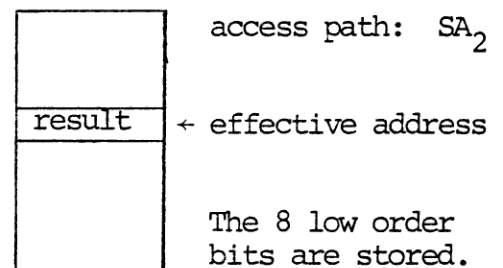
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVSB \*)

VAR

 param1 : word;  
 operand2 : addr;

BEGIN

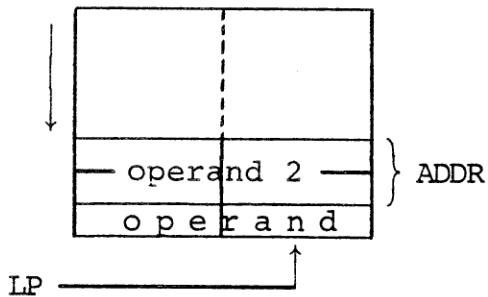
 param1 := memword(ip+1);  
 operand2.base := stack(lp-5);  
 operand2.disp := stack(lp-3);  
 IF operand2.base.nilbit THEN  
 exception(nil exception);  
 membyte(operand2+param1) := stack(lp-1) AND 255;  
 lp := lp - 6;  
 ip := ip + 3

END; (\* STVSB \*)

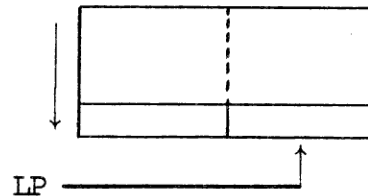
Store Value S(tack) WordValue: B8<sub>Hex</sub>IP → 

|       |             |
|-------|-------------|
| STVSW | parameter 1 |
|-------|-------------|

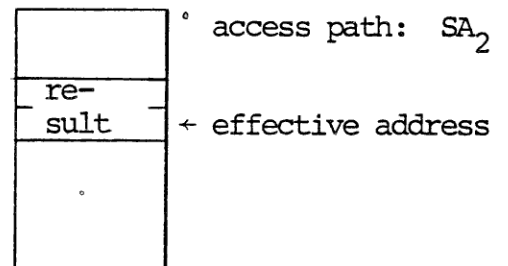
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVSW \*)

VAR

 param1 : word;  
 operand2 : addr;

BEGIN

```

  param1 := memword(ip+1);
  operand2.base := stack(lp-5);
  operand2.disp := stack(lp-3);
  IF operand2.base.nilbit THEN
    exception(nil exception);
  memword(operand2+param1) := stack(lp-1);
  lp := lp - 6;
  ip := ip + 3;
END;    (* STVSW *)

```



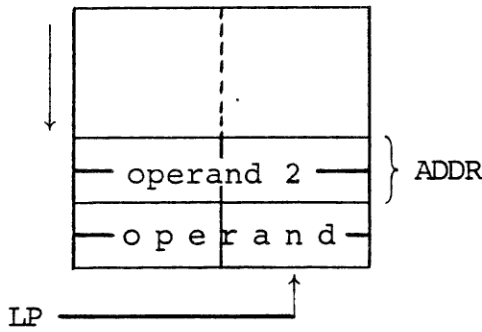
## 6.2.19 STVSD

6.2.19

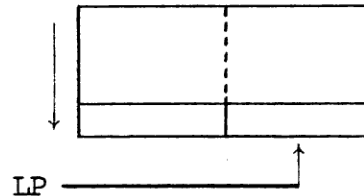
Store Value S(tack) DoubleValue: F8<sub>Hex</sub>IP → 

|       |             |
|-------|-------------|
| STVSD | parameter 1 |
|-------|-------------|

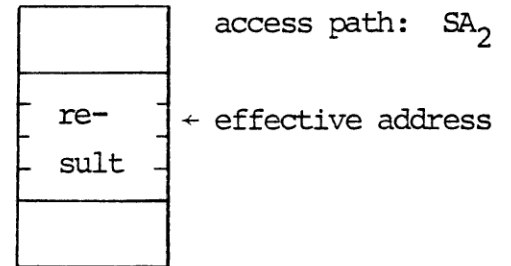
STACK BEFORE:



STACK AFTER:



MEMORY:



(\* STVSD \*)

VAR

param1 : word;

param2 : addr;

BEGIN

param1 := memword(ip+1);

operand2.base := stack(lp-7);

operand2.disp := stack(lp-5);

IF operand2.base.nilbit THEN

exception (nil exception);

memword(operand2+param1) := stack(lp-3);

memword(operand2+param1+2) := stack(lp-1);

lp := lp - 8;

ip := ip + 3

END; (\* STVSD \*)

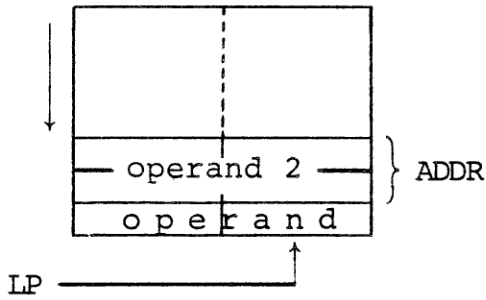
## 6.2.20 STVSF

6.2.20

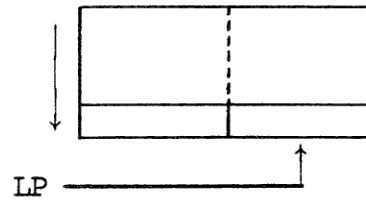
Store Value S(tack) FieldValue:  $D8_{Hex}$ IP → 

|       |             |         |
|-------|-------------|---------|
| STVSF | parameter 1 | param 2 |
|-------|-------------|---------|

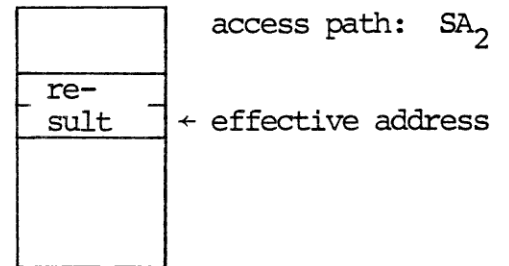
STACK BEFORE:



STACK AFTER:



MEMORY:



```
(* STVSF *)
```

```
TYPE
```

```
    fieldtype = PACKED RECORD
        firstbit : 0..15;
        lastbit  : 0..15;
    END;
```

```
VAR
```

```
    param1 : word;
    param2 : fieldtype;
    operand2 : addr;
    operand1 : word;
    result : word;
    mask : word;
```

```
BEGIN
```

```
    param1 := memword(ip+1)
    param2 := membyte(ip+3)
    operand2.base := stack(lp-5);
    operand2.disp := stack(lp-3);
    IF operand2.base.nilbit THEN
        exception(nil exception);
    WITH param2 DO
        BEGIN
            IF firstbit > lastbit THEN
                exception(field error);
            operand1 := stack(lp-1);
            mask := 1 shift(lastbit+1-firstbit)-1;
            IF mask < operand1 THEN (* unsigned *)
                exception(field overflow);
            mask := mask shift(15-lastbit);
            operand1 := operand1 shift(15-lastbit)
        END;
        result := memword(operand2+param1);
        result := result AND (NOT mask);
        memword(operand2+param1) := result OR operand1;
        lp := lp - 6;
        ip := ip + 4
    END; (* STVSF *)
```

## 6.3 Manipulation of Storage Areas

6.3

### 6.3.1 Move a Storage Area

6.3.1

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.

#### 6.3.1.1 MOVEG

6.3.1.1

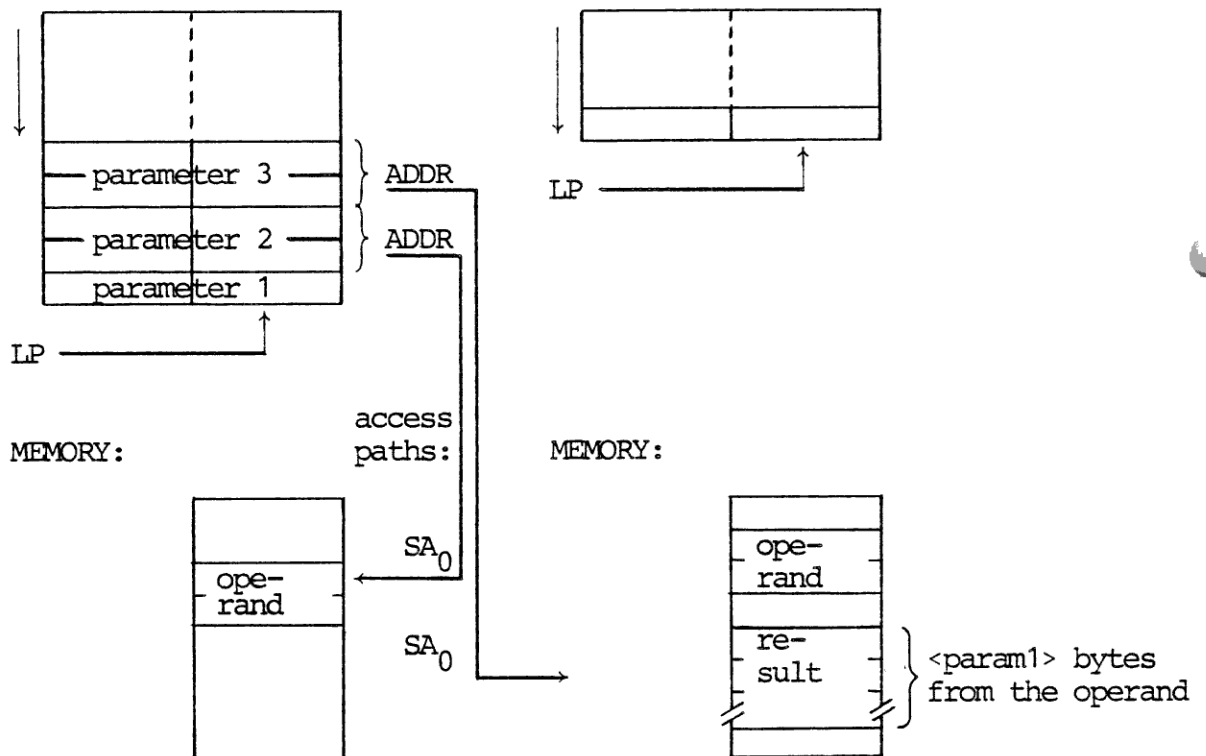
MOVE General

Value:  $AE_{Hex}$

IP → MOVEG

STACK BEFORE:

STACK AFTER:



```

(* MOVEG *)

VAR
  param1 : word;
  param2 : addr; (* source *)
  param3 : addr; (* destination *)
BEGIN
  param1 := stack(lp-1);
  param2.base := stack(lp-5);
  param2.disp := stack(lp-3);
  param3.base := stack(lp-9);
  param3.disp := stack(lp-7);
  IF param2.base.nilbit OR param3.base.nilbit THEN exception(nil exception);
  IF odd(param1) THEN
    BEGIN
      membyte(param3) := membyte(param2);
      param3 := param3+1;
      param2 := param2+1;
      param1 := param1-1
    END;
  WHILE param1 > 0 DO    (* unsigned compare *)
    BEGIN
      memword(param3) := memword(param2);
      param3 := param3 + 2;
      param2 := param2 + 2;
      param1 := param1 - 2;
      IF curlevel < nextlevel THEN
        BEGIN
          stack(lp-7) := param3.disp;
          stack(lp-3) := param2.disp;
          stack(lp-1) := param1;
          GOTO fetch
        END
      END;
      lp := lp - 10;
      ip := ip + 1
    END;    (* MOVEG *)

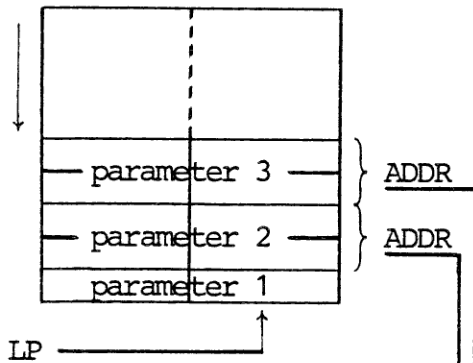
```

## 6.3.1.2 MOVEB

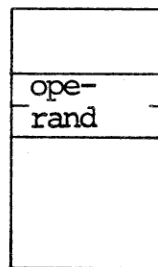
6.3.1.2

MOVE BytesValue:  $AC_{Hex}$ IP → MOVEB

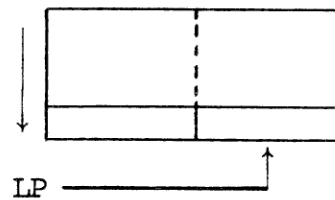
STACK BEFORE:



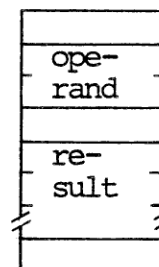
MEMORY:

access  
paths: $SA_0$  $SA_0$ 

STACK AFTER:



MEMORY:

<param1> bytes  
from the operand

```

(* MOVEB *)
VAR
  param1 : word;
  param2 : addr;  (* source *)
  param3 : addr;  (* destination *)
BEGIN
  param1 := stack(lp-1);
  param2.base := stack(lp-5);
  param2.disp := stack(lp-3);
  param3.base := stack(lp-9);
  param3.disp := stack(lp-7);
  IF param2.base.nilbit OR param3.base.nilbit THEN exception(nil exception);
  WHILE param1 > 0 DO    (* unsigned compare *)
    BEGIN
      membyte(param3) := membyte(param2);
      param3 := param3 + 1;
      param2 := param2 + 1;
      param1 := param1 - 1;
      IF curlevel < nextlevel THEN
        BEGIN
          stack(lp-7) := param3.disp;
          stack(lp-3) := param2.disp;
          stack(lp-1) := param1;
          GOTO fetch
        END
      END;
      lp := lp - 10;
      ip := ip + 1
    END;
  (* MOVEB *)

```

### 6.3.2 Exchange Two Storage Addresses

6.3.2

Two addresses are exchanged indivisibly.

#### 6.3.2.1 CEXCH

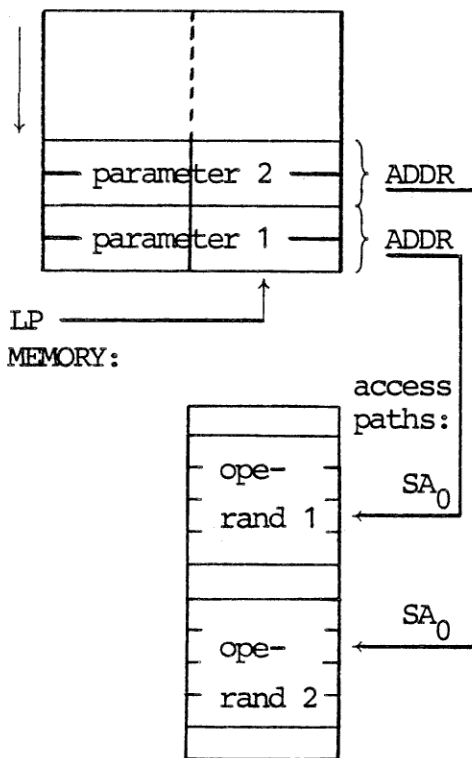
6.3.2.1

Control EXCHange

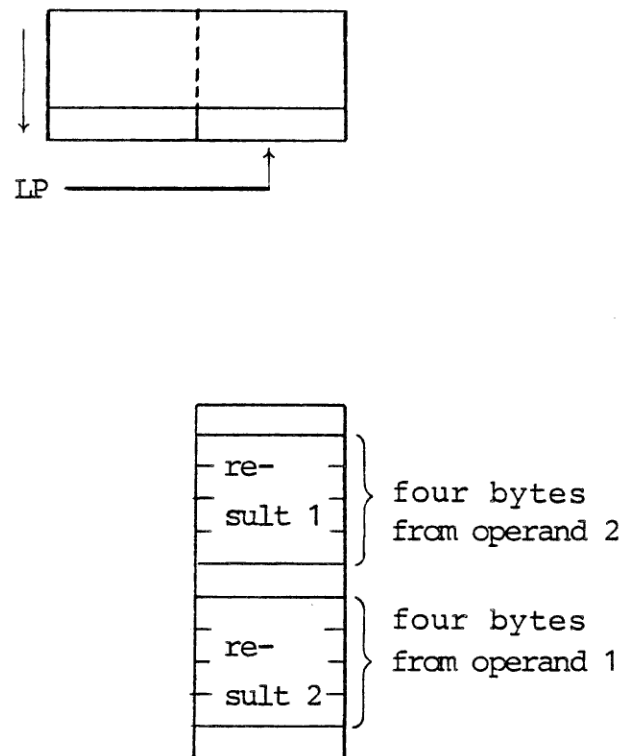
Value: FE<sub>Hex</sub>

IP → CEXCH

STACK BEFORE:



STACK AFTER:



The displacement in SA<sub>0</sub> must be even.



(\* CEXCH \*)

VAR

param1 : addr;  
param2 : addr;  
work1 : addr;  
work2 : addr;

BEGIN

param1.base := stack(lp-3);  
param1.disp := stack(lp-1);  
param2.base := stack(lp-7);  
param2.disp := stack(lp-5);  
IF param1.base.nilbit OR param2.base.nilbit THEN  
  exception(nil exception);  
IF odd(param1.disp) OR odd(param2.disp) THEN  
  exception(odd displacement);  
work1.base := mem(param1);  
work1.disp := mem(param1+2);  
work2.base := mem(param2);  
work2.disp := mem(param2+2);  
IF work1.base.lockbit OR work2.base.lockbit THEN  
  exception(lock exception);  
mem(param1) := work2.base;  
mem(param1+2) := work2.disp;  
mem(param2) := work1.base;  
mem(param2+2) := work1.disp;

lp := lp - 8;  
ip := ip + 1

END; (\* CEXCH \*)

### 6.3.3 Compare Two Storage Areas

6.3.3

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 true (= 1) if all the bytes are pairwise equal; otherwise the result is false (= 0).

#### 6.3.3.1 STCEA

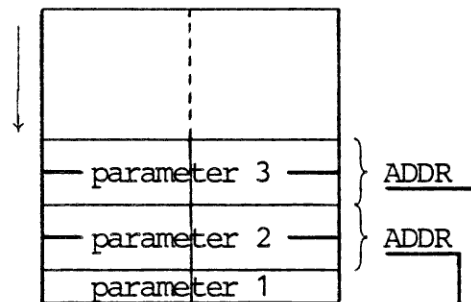
6.3.3.1

Storage Compare Equal Area

Value: EE<sub>Hex</sub>

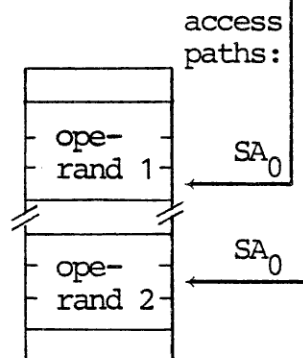
IP → STCEA

STACK BEFORE:

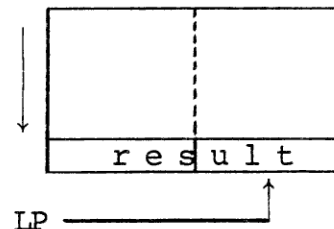


LP →

MEMORY:



STACK AFTER:



#### Relation

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

```

(* STCEA *)
VAR
  param1 : word;
  param2 : addr;
  param3 : addr;
  equal : boolean := true;
BEGIN
  param1 := stack(lp-1);
  param2.base := stack(lp-5);
  param2.disp := stack(lp-3);
  param3.base := stack(lp-9);
  param3.disp := stack(lp-7);
  IF param2.base.nilbit OR param3.base.nilbit THEN exception(nil exception);
  WHILE param1 > 0 DO    (* unsigned compare *)
    BEGIN
      IF membyte(param2) <> membyte(param3) THEN
        BEGIN
          equal := false;
          GOTO finis
        END;
      param1 := param1 - 1;
      param2 := param2 + 1;
      param3 := param3 + 1;
      IF curlevel < nextlevel THEN
        BEGIN
          stack(lp-1) := param1;
          stack(lp-3) := param2.disp;
          stack(lp-7) := param3.disp;
          GOTO fetch
        END
      END;
    finis: stack(lp-9) := word(equal);
      lp := lp - 8;
      ip := ip + 1
    END;    (* STCEA *)

```

### 6.3.4 Push a Storage Area

6.3.4

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 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.

#### 6.3.4.1 REVSM

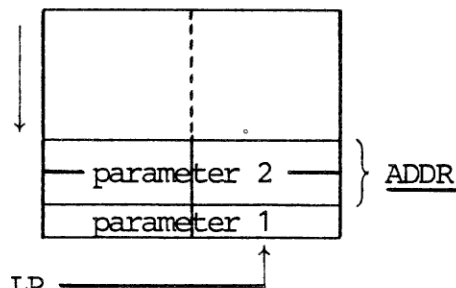
6.3.4.1

REtrieve VAlue STack Multiple

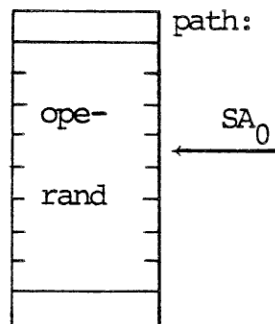
Value: CC<sub>Hex</sub>

IP → REVSM

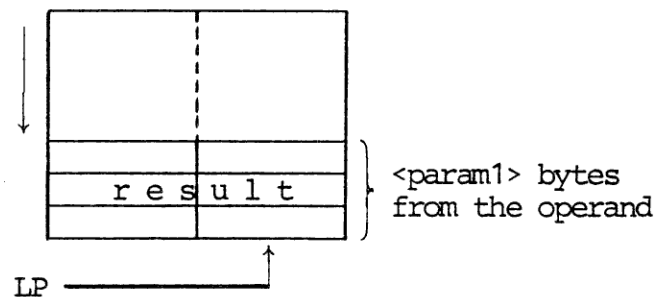
STACK BEFORE:



MEMORY:



STACK AFTER:



```

(* REVSM *)
VAR
  param1 : word;
  param2 : addr;
BEGIN
  param1 := stack(lp-1);
  IF lp > ms - param1 - 4 (* room for parameters in case of interrupt *)
  THEN exception(stack overflow);
  IF odd(param1) THEN
    exception(odd operand);
  param2.base := stack(lp-5);
  param2.disp := stack(lp-3);
  IF param2.base.nilbit THEN
    exception(nil exception);
  lp := lp - 6;
  WHILE param1 > 0 DO    (* unsigned compare *)
    BEGIN
      lp := lp + 2;
      stack(lp-1) := memword(param2);
      param2 := param2 + 2;
      param1 := param1 - 2;
      IF curlevel < nextlevel THEN
        BEGIN
          lp := lp + 6;
          stack(lp-1) := param1;
          stack(lp-3) := param2.disp;
          stack(lp-5) := param2.base;
          GOTO fetch
        END
      END;
      ip := ip + 1
    END; (* REVSM *)

```

### 6.3.5 Pop a Storage Area

6.3.5

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.

#### 6.3.5.1 SETST

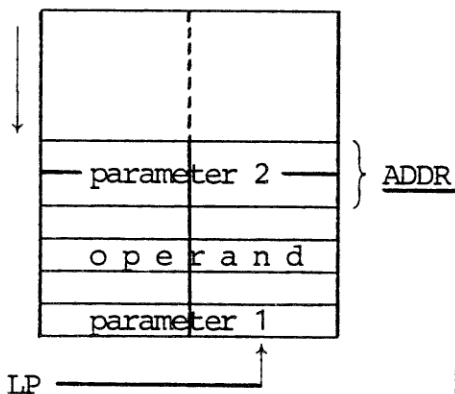
6.3.5.1

SET Store

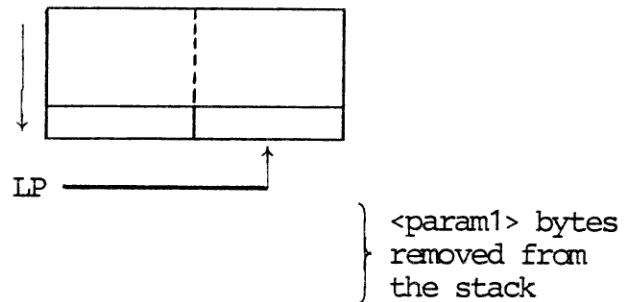
Value:  $EC_{Hex}$

IP → SETST

STACK BEFORE:



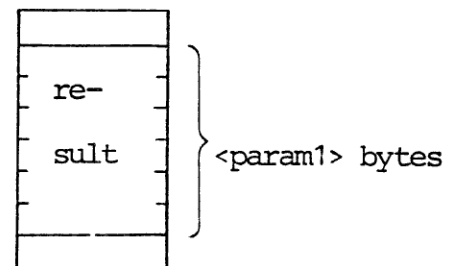
STACK AFTER:



access  
path:

MEMORY:

$SA_0$



```
(* SETST *)
```

```
VAR
```

```
  param1 : word;  
  start1,param2 : addr;  
  i : integer;
```

```
BEGIN
```

```
  resume := true;  
  param1 := stack(lp-1);  
  start1 := lp-1 - param1;  
  param2.base := stack(lp-1-param1-4);  
  param2.disp := stack(lp-1-param1-2);  
  IF odd(param1) THEN exception(odd operand);  
  IF param2.base.nilbit THEN exception(nil exception);  
  FOR i := 0 TO param1 DIV2-1 DO
```

```
    BEGIN
```

```
      membyte(param2-i*2) := stack(start1+i*2);  
      IF interrupt condition THEN  
        instruction_fetch
```

```
    END;
```

```
  resume := false;
```

```
  lp := start1 - 5;
```

```
  ip := ip + 1
```

```
END;      (* SETST *)
```

## 7. PROCEDURE CALL AND EXIT

7.

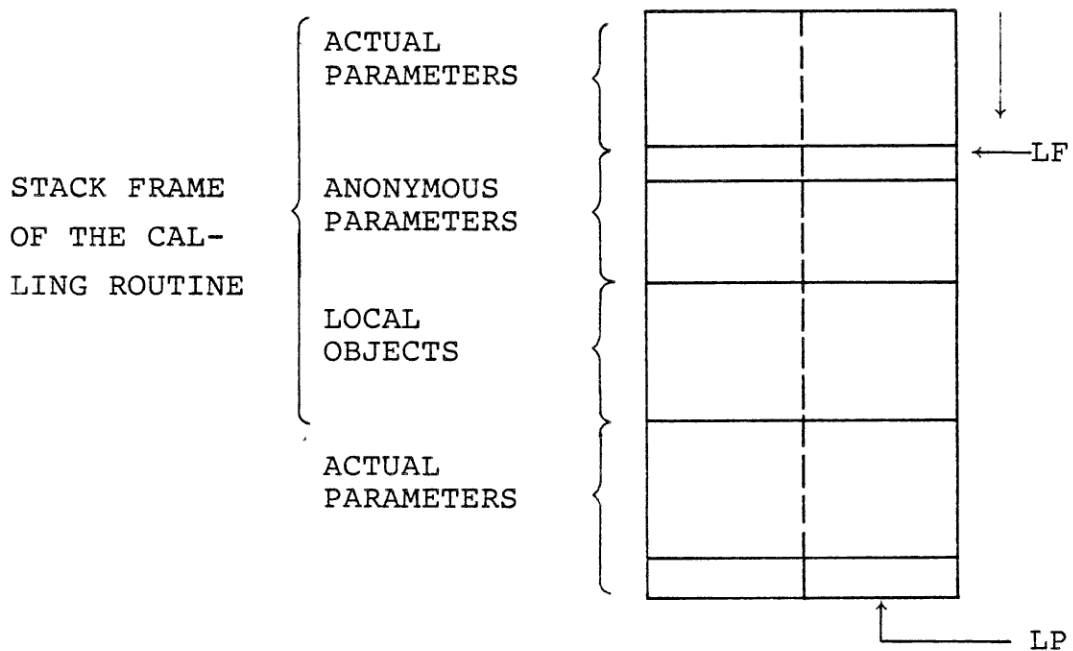
### 7.1 Enter a Routine

7.1

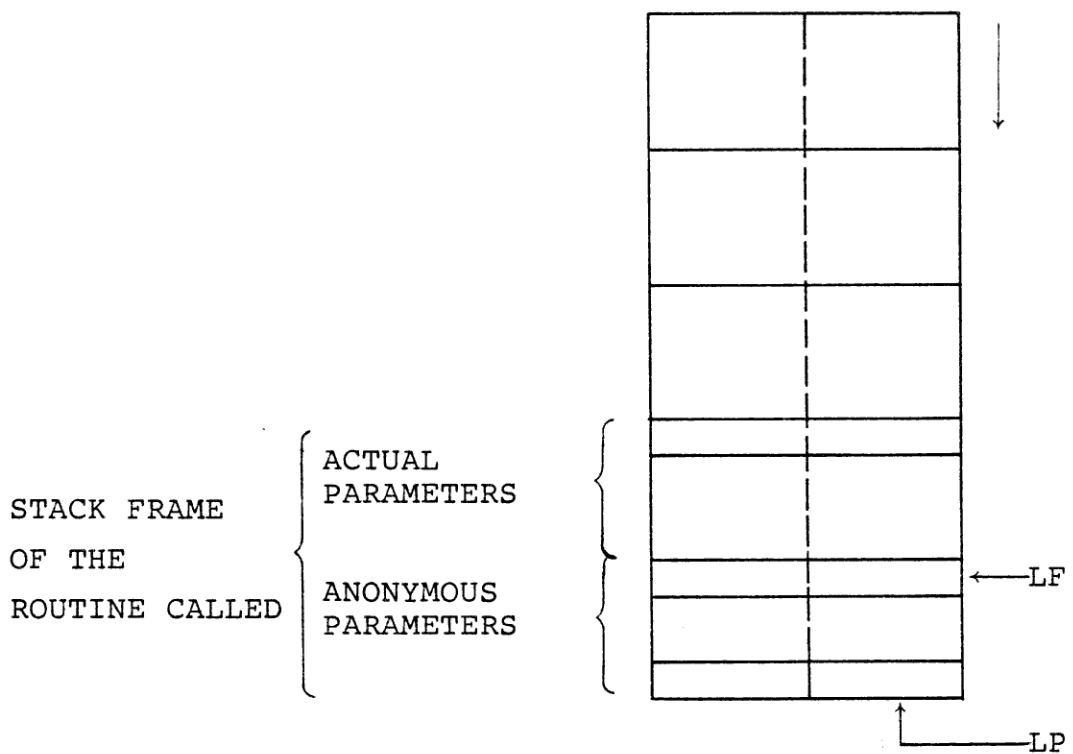
The execution of this instruction initiates a routine call. It assumes that the actual parameters have been calculated on the top of the stack.

(tegning 7.1)

#### STACK BEFORE EXECUTION OF THE INSTRUCTION



#### STACK AFTER EXECUTION OF THE INSTRUCTION





The instruction has three parameters:

- Parameter 1  
the value of which specifies the difference between the static level of the calling routine and the static level of the routine to be called.
- Parameter 2  
the value of which specifies the entry point in the routine to be called.

The execution of the instruction has the following effect:

- An area for the anonymous parameters is reserved and the contents are defined.
- The local frame pointer, LF, is set to address the second byte of the first anonymous parameter.
- The value of parameter 2 (an ADDR) is assigned to the instruction pointer, IP.

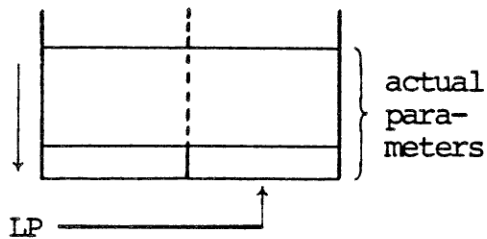
The anonymous parameters are:

Static link pointer  
Dynamic link pointer, i.e. old LF  
Return point

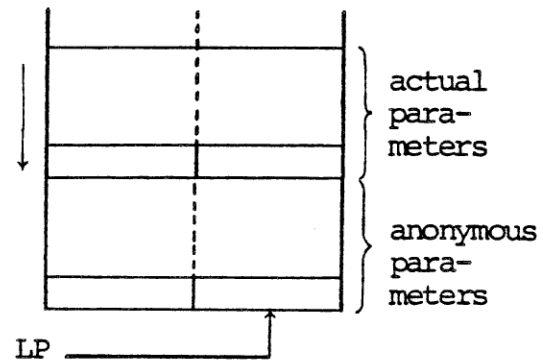
Procedure CALL StaticValue: 7A<sub>Hex</sub>IP → 

|       |         |   |   |   |   |   |   |   |   |   |   |
|-------|---------|---|---|---|---|---|---|---|---|---|---|
| PCALS | param 1 | p | a | r | a | m | e | t | e | r | 2 |
|-------|---------|---|---|---|---|---|---|---|---|---|---|

STACK BEFORE:



STACK AFTER:



(\* PCALS \*)

VAR

param1 : byte;

param2 : addr;

FUNCTION get\_if(level : byte) : addr;

EXTERNAL; (\* to retrieve address of intermediate frame \*)

BEGIN

IF lp &gt; ms - 12 THEN exception(stack overflow)

param1 := membyte(ip+1);

param2.base := memword(ip+2);

param2.disp := memword(ip+4);

stack(lp+1) := get\_if(param1).base; (\* static link \*)

stack(lp+3) := get\_if(param1).disp;

stack(lp+5) := lf.base; (\* dynamic link \*)

stack(lp+7) := lf.disp;

stack(lp+9) := ip.base; (\* return point \*)

stack(lp+11) := ip.disp + 6;

lf := lp + 2;

ip := param2;

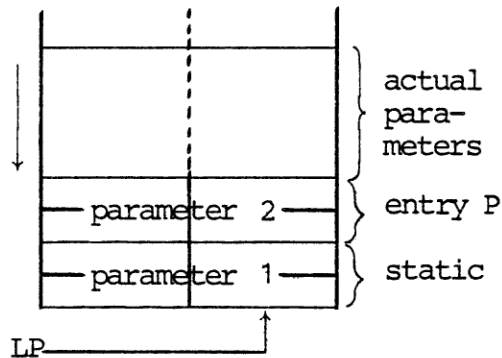
lp := lp + 12

END; (\* PCALS \*)

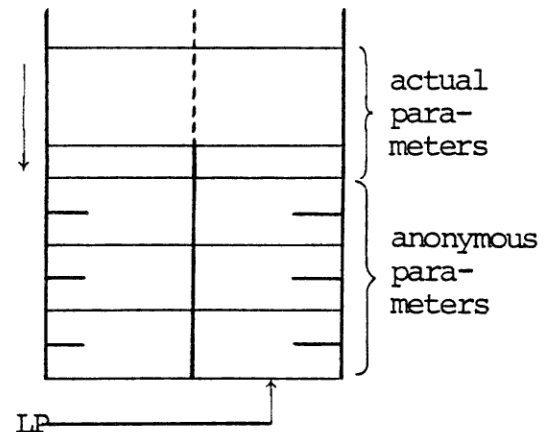
Procedure CALl DynamicValue: 79<sub>Hex</sub>

IP → PCALD

STACK BEFORE:



STACK AFTER:



Parameter 1 specifies explicitly the ADDR of the stack frame to which the static link is to be connected.

(\* PCALD \*)

VAR

entry\_point : addr;

BEGIN

IF lp &gt; ms - 4 THEN exception(stack overflow)

entry\_point.base := stack(lp-7);

entry\_point.disp := stack(lp-5);

stack(lp-7) := stack(lp-3); (\*static link \*)

stack(lp-5) := stack(lp-1);

stack(lp-3) := lf.base; (\* dynamic link \*)

stack(lp-1) := lf.disp;

lf := lp - 6;

stack(lp+1) := ip.base; (\* return point \*)

stack(lp+3) := ip.disp + 1;

ip := entry\_point;

lp: lp + 4

END; (\* PCALD \*)

## 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

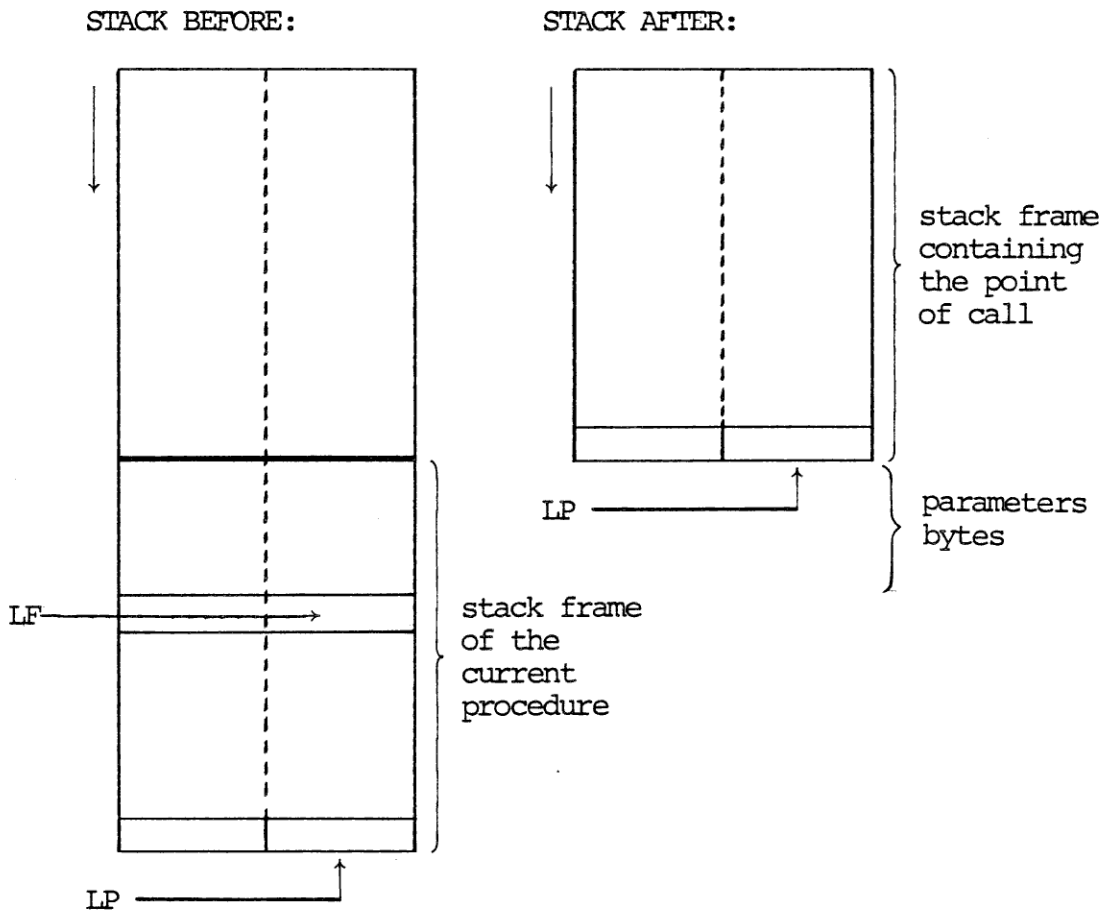
7.2.1

Procedure EXIT

Value: 7B<sub>Hex</sub>

IP → 

|       |           |
|-------|-----------|
| PEXIT | parameter |
|-------|-----------|



(\* PEXIT \*)

BEGIN

lp := lf - memword(ip+1);

ip.base := stack(lf+7);

ip.disp := stack(lf+9);

lf.disp := stack(lf+5)

END; (\* PEXIT \*)

8. JUMPS

8.

8.1 Unconditional Jumps

8.1

The operand is retrieved, and a result, which is interpreted as an ADDR, is calculated and assigned to the instruction pointer, IP.

8.1.1 JMPHC

8.1.1

JuMP H (path) local Context

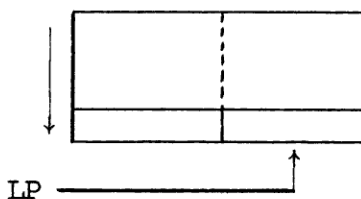
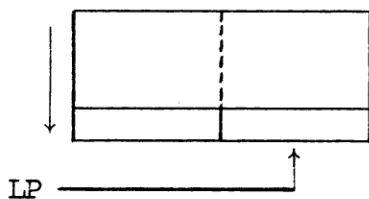
Value: 69<sub>Hex</sub>

IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| JMPHC | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:

STACK AFTER:



ip := result;

(\* JMPHC \*)

VAR

param : addr;

BEGIN

param.base := memword(ip+1);

param.disp := memword(ip+3);

ip.base := param.base;

ip.disp := param.disp

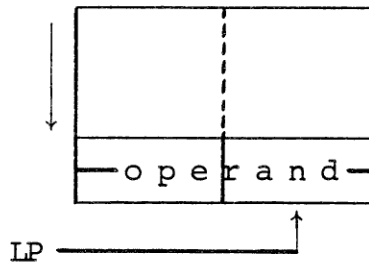
END; (\* JMPHC \*)

8.1.2 JMPPD

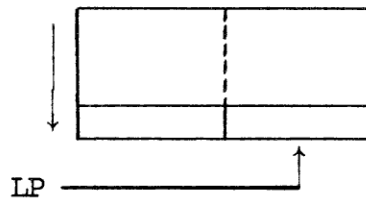
8.1.2

JuMP P (path) DirectValue: 6A<sub>Hex</sub>IP → JMPPD

STACK BEFORE:



STACK AFTER:



ip := result;

(\* JMPPD \*)

BEGIN

ip.base := stack(lp-3);

ip.disp := stack(lp-1);

lp := lp - 4

END; (\* JMPPD \*)

8.1.3 JMPRW

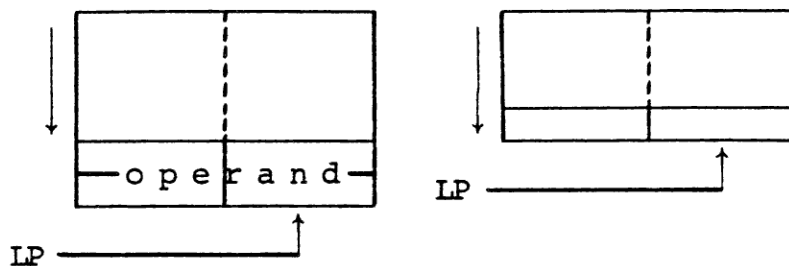
8.1.3

JuMP Relative WordValue: 67<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| JMPRW | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:



ip := result;

The parameter is interpreted as a signed integer (relative address) and added to the displacement field of the instruction pointer (ip.disp). A carry is not transferred to the base field (ip.base), but a 16-bit wrap-around is used.

(\* JMPRW \*)

VAR

param : word;

BEGIN

param := memword(ip+1);

ip := ip + param

END; (\* JMPRW \*)

8.2 Case Jump

8.2

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

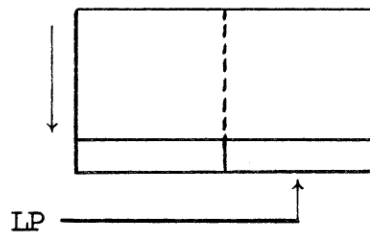
8.2.1 JMCHT

8.2.1

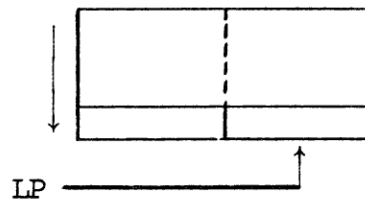
JuMp Case H (path) TableValue: 6B<sub>Hex</sub>IP → 

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| JMCHT | p | a | r | a | m | e | t | e | r |
|-------|---|---|---|---|---|---|---|---|---|

STACK BEFORE:



STACK AFTER:



PARAMETER →

CASE JUMPTABLE:

4 x (max. value  
min. value + 1)  
bytes

|                         |
|-------------------------|
| min. value              |
| max. value              |
| otherwise -<br>ADDR     |
| case min. value<br>ADDR |
|                         |
| case max. value<br>ADDR |



```

(* JMCHT *)
VAR
  param : addr;
  operand : word;
BEGIN
  param.base := memword(ip+1);
  param.disp := memword(ip+3);
  operand := stack(lp-1);
  IF (operand < memword(param))
    OR (operand > memword(param+2)) THEN
  BEGIN
    ip.base := memword(param+4);
    ip.disp := memword(param+6)
  END
  ELSE
  BEGIN
    operand := operand - memword(param);
    ip.base := memword(param+8+operand*4);
    ip.disp := memword(param+10+operand*4)
  END;
  lp := lp - 2
END;      (* JMCHT *)

```

## 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, IP. The result, which is interpreted as an ADDR, is calculated as

$$\text{result} := \text{IP} + \text{parameter}$$

## 8.3.1 JMZEQ

8.3.1

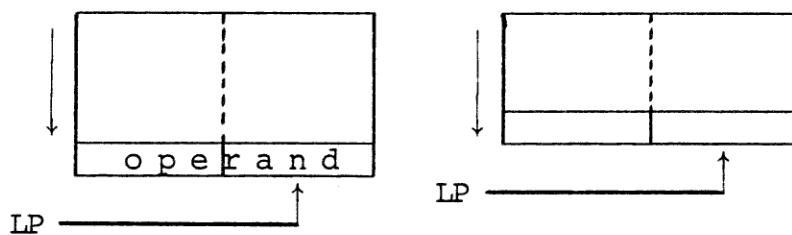
JuMp Zero EqualValue: 61<sub>Hex</sub>

IP → 

|       |           |
|-------|-----------|
| JMZEQ | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:



relation: operand = 0

(\* JMZEQ \*)

VAR

```

  param : word;
  operand : integer;

```

BEGIN

```

  param := memword(ip+1);
  operand := stack(lp-1);

```

```

  IF operand = 0 THEN

```

```

    ip := ip + param

```

```

  ELSE

```

```

    ip := ip + 3;

```

```

    lp := lp - 2

```

```

  END;      (* JMZEQ *)

```

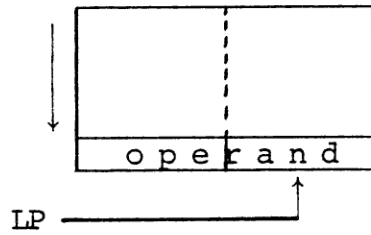
8.3.2 JMZNE

8.3.2

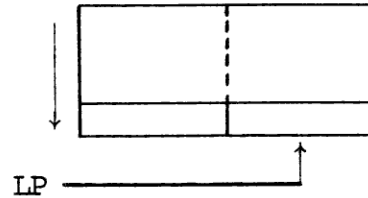
JuMp Zero Not EqualValue: 62<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| JMZNE | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:

relation: operand  $\neq$  0

(\* JMZNE \*)

VAR

 param : word;  
 operand : integer;

BEGIN

param := memword(ip+1);

operand := stack(lp-1);

IF operand  $\neq$  0 THEN

ip := ip + param;

ELSE

ip := ip + 3;

lp := lp - 2

END; (\* JMZNE \*)

8.3.3 JMZLT

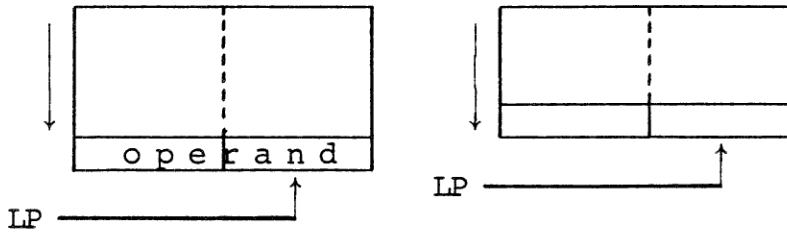
8.3.3

Jmp Zero Less ThanValue: 63<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| JMZLT | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:



relation: operand &lt; 0

The operand is interpreted as a signed integer.

(\* JMZLT \*)

VAR

param : word;

operand : integer;

BEGIN

param := memword(ip+1);

operand := stack(lp-1);

IF operand &lt; 0 THEN

ip := ip + param

ELSE

ip := ip + 3;

lp := lp - 2

END; (\* JMZLT \*)

## 8.3.4 JMZGT

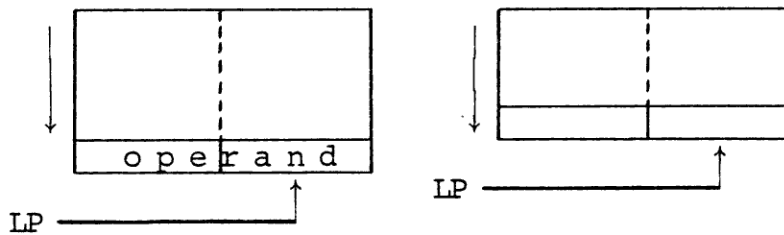
8.3.4

Jump Zero Greater ThanValue: 64<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| JMZGT | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:



relation: operand &gt; 0

The operand is interpreted as a signed integer.

(\* JMZLT \*)

VAR

param : word;

operand : integer;

BEGIN

param := memword(ip+1);

operand := stack(lp-1);

IF operand &gt; 0 THEN

ip := ip + param

ELSE

ip := ip + 3;

lp := lp - 2

END; (\* JMZGT \*)

8.3.5 JMZLE

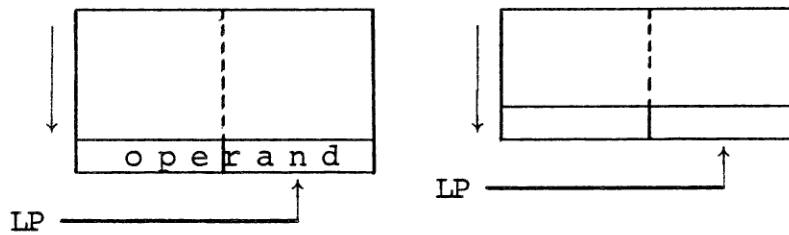
8.3.5

Jump Zero Less Than or EqualValue: 65<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| JMZLE | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:

relation: operand  $\leq$  0

The operand is interpreted as a signed integer.

(\* JMZLE \*)

VAR

param : word;

operand : integer;

BEGIN

param := memword(ip+1);

operand := stack(lp-1);

IF operand  $\leq$  0 THEN

ip := ip + param

ELSE

ip := ip + 3;

lp := lp - 2

END; (\* JMZLE \*)

8.3.6 JMZGE

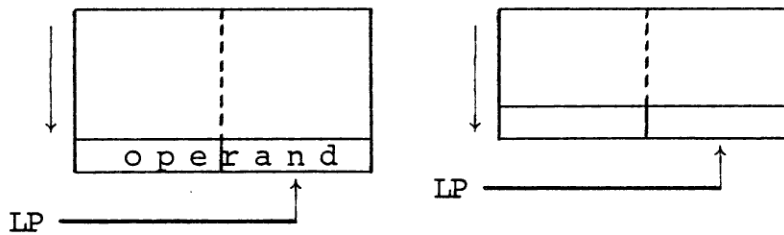
8.3.6

Jump Zero Greater Than or EqualValue: 66<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| JMZGE | parameter |
|-------|-----------|

STACK BEFORE:

STACK AFTER:

relation: operand  $\geq$  0

The operand is interpreted as a signed integer.

(\* JMZGE \*)

VAR

param : word;

operand : integer;

BEGIN

param := memword(ip+1);

operand := stack(lp-1);

IF operand  $\geq$  0 THEN

ip := ip + param;

ELSE

ip := ip + 3;

lp := lp - 2

END; (\* JMZGE \*)

## 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.

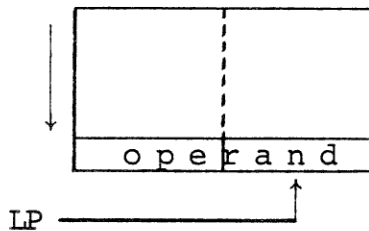
#### 9.1.1 NEG

9.1.1

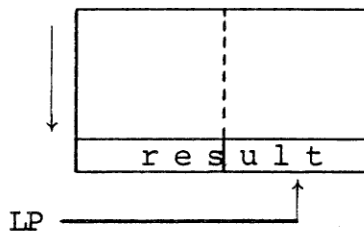
NEGate

Value: 50<sub>Hex</sub>

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.

(\* NEG \*)

```
VAR
  result : integer;
BEGIN
  result := -stack(lp-1);
  IF (overflow_bit=1) AND (ps.suppr_overflow=0) THEN
    exception(overflow);
  stack(lp-1) := result;
  ip := ip + 1
END;    (* NEG *)
```

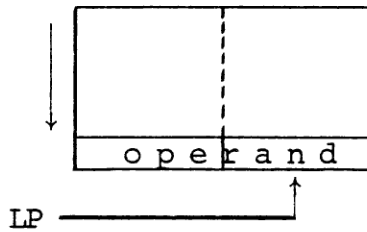


9.1.2 NOT

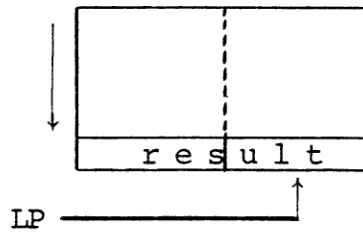
9.1.2

NOTValue: 3D<sub>Hex</sub>IP → NOT

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).

(\* NOT \*)

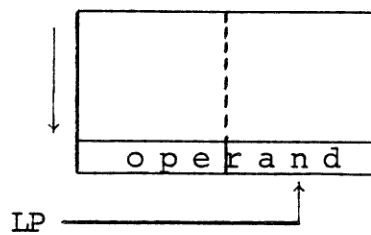
```

BEGIN
  IF stack(lp-1) = 1 THEN
    stack(lp-1) := false
  ELSE
    stack(lp-1) := true;
  ip := ip + 1
END;    (* NOT *)

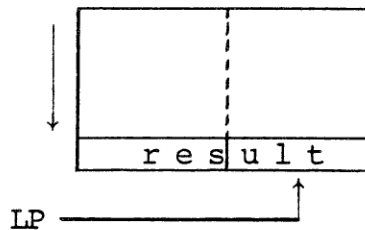
```

ABSsolute ValueValue: 51<sub>Hex</sub>IP → ABS

STACK BEFORE:



STACK AFTER:



operator: absolute value

The operand is interpreted as a signed integer.

The result is the absolute value of the operand.

(\* ABS \*)

VAR

operand : integer;

BEGIN

operand := stack(lp-1);

IF operand &lt; 0 THEN

BEGIN

operand := -operand;

        IF (overflow\_bit=1) AND (ps.suppr\_overflow=0) THEN  
            exception(overflow);

stack(lp-1) := operand

END;

ip := ip + 1

END;     (\* ABS \*)

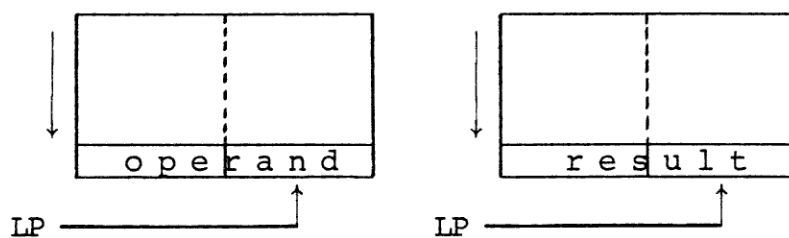
9.1.4 COMPL

9.1.4

COMPLementValue: 52<sub>Hex</sub>IP → 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.

(\* COMPL \*)

TYPE

bitword = PACKED ARRAY(0..15) OF bit;

VAR

i : integer;

operand : bitword;

BEGIN

operand := stack(lp-1);

FOR i := 0 TO 15 DO

operand(i) := 1 - operand(i);

stack(lp-1) := operand;

ip := ip + 1

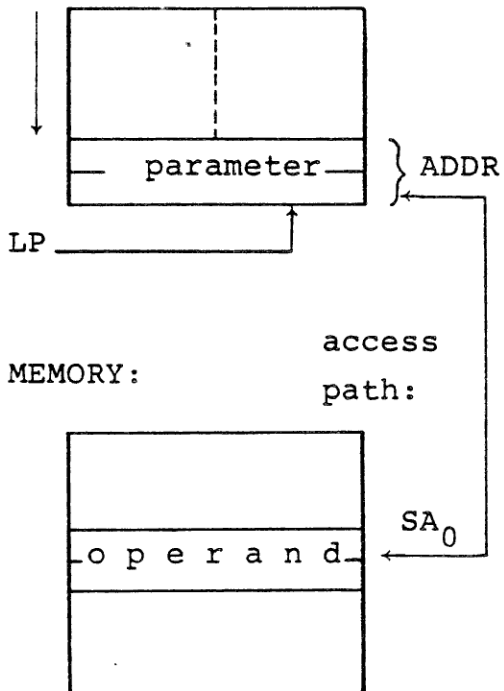
END; (\* COMPL \*)

9.1.5 TNILL

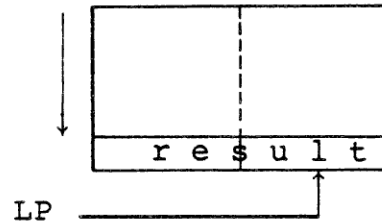
9.1.5

Test NILL addressValue: 39<sub>Hex</sub>IP → TNILL

STACK BEFORE:



STACK AFTER:



The operand must start on a word boundary.

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

Warning: The parameter must be a valid address, not nil; otherwise the result of executing TNILL is unpredictable.

(\* TNILL \*)

VAR

parameter : addr;  
operand : addr;  
result : word;

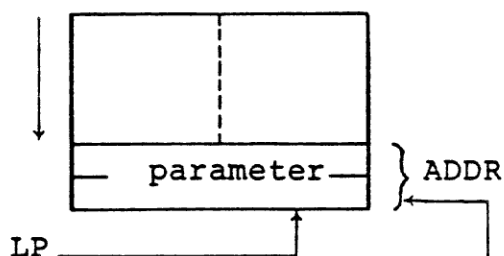
BEGIN

parameter.base := stack(lp-3);  
parameter.disp := stack(lp-1);  
operand := memaddr(parameter);  
result := operand.base.nilbit;  
stack(lp-3) := result;  
lp := lp - 2;  
ip := ip + 1

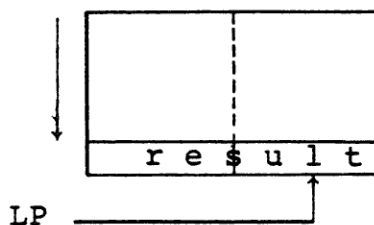
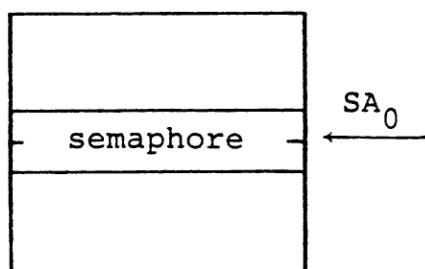
END; (\* TNILL \*)

Test OPEN SemaphoreValue: 3A<sub>Hex</sub>IP → TOPEN

STACK BEFORE:



STACK AFTER:

MEMORY: access  
path:

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).

Warning: The parameter must be a valid address, not nil; otherwise the result of executing TOPEN is unpredictable.

(\* TOPEN \*)

VAR

```
parameter : addr;
semaphore : addr;
result    : word;
```

BEGIN

```
parameter.base := stack(lp-3);
parameter.disp := stack(lp-1);
semaphore := memaddr(parameter);
IF semaphore.base.nilbit = 1 THEN
  result := 0
```

ELSE

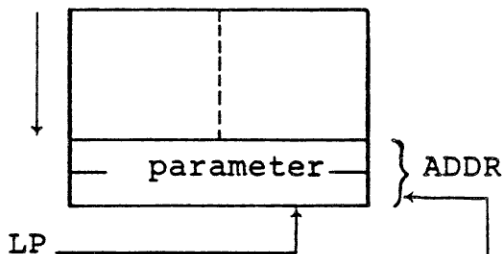
```
  result := 1 - semaphore.disp AND 1;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
```

END; (\* TOPEN \*)

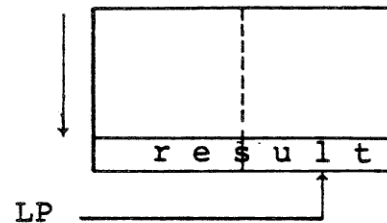
Test LOCKed SemaphoreValue: 3B<sub>Hex</sub>

IP → TLOCK

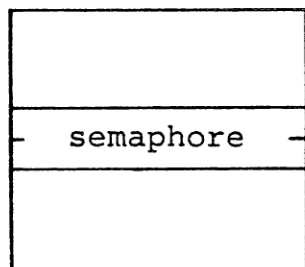
STACK BEFORE:



STACK AFTER:



MEMORY: access path:



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).

Warning: The parameter must be a valid address, not nil; otherwise the result of executing TLOCK is unpredictable.

(\* TLOCK \*)

VAR

```
parameter : addr;
semaphore : addr;
result    : word;
```

BEGIN

```
parameter.base := stack(lp-3);
parameter.disp := stack(lp-1);
semaphore := memaddr(parameter);
IF semaphore.base.nilbit = 1 THEN
  result := 0
ELSE
  result := semaphore.disp AND 1;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
```

END; (\* TLOCK \*)

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.

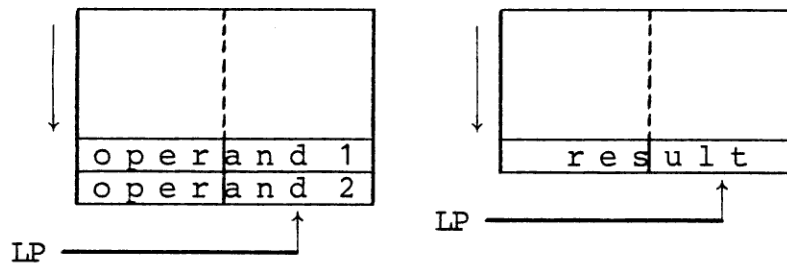
9.2.1 ADD

9.2.1

ADDValue: 44<sub>Hex</sub>IP → ADD

STACK BEFORE:

STACK AFTER:



operator: add

Operands 1 and 2 are interpreted as signed integers.

(\* ADD \*)

VAR

operand1, operand2, result : integer;

BEGIN

operand1 := stack(lp-3);

operand2 := stack(lp-1);

result := operand1 + operand2;

IF (overflow bit=1) AND (ps.suppr\_overflow=0) THEN  
exception(overflow);

stack(lp-3) := result;

lp := lp - 2;

ip := ip + 1

END; (\* ADD \*)

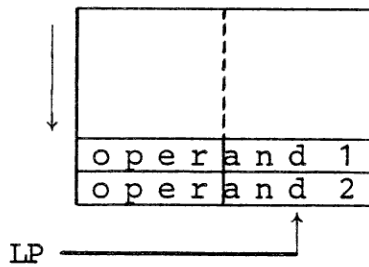
## 9.2.2 SUB

9.2.2

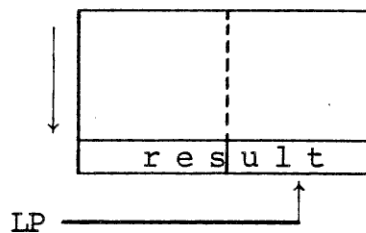
SUBtractValue: 45<sub>Hex</sub>IP → 

|     |
|-----|
| SUB |
|-----|

STACK BEFORE:



STACK AFTER:



operator: subtract

Operands 1 and 2 are interpreted as signed integers.

Operand 2 is subtracted from operand 1.

(\* SUB \*)

VAR

```

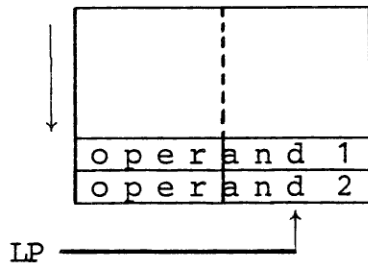
  operand1, operand2, result : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  result := operand1 - operand2;
  IF (overflow_bit=1) AND (ps.suppr_overflow=0) THEN
    exception(overflow);
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;    (* SUB *)

```

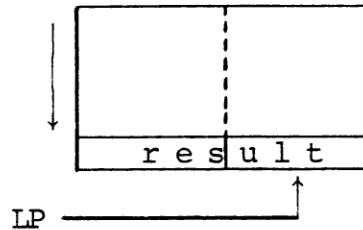


MultiplyValue: 49<sub>Hex</sub>IP → MUL

STACK BEFORE:



STACK AFTER:



operator: multiply

Operands 1 and 2 are interpreted as signed integers.

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

(\* MUL \*)

VAR

```

    operand1, operand2, result : integer;
BEGIN
    operand1 := stack(lp-3);
    operand2 := stack(lp-1);
    result := operand1 * operand2;
    IF (overflow_bit=1) AND (ps.suppr_overflow=0) THEN
        exception(overflow);
    stack(lp-3) := result;
    lp := lp - 2;
    ip := ip + 1
END;    (* MUL *)

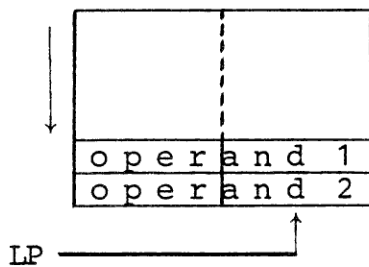
```

9.2.4 DIV

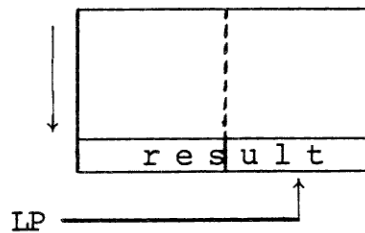
9.2.4

DivideValue: 4A<sub>Hex</sub>IP → 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.

(\* DIV \*)

VAR

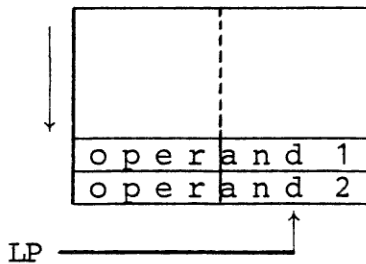
```

  operand1, operand2, result : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF (operand2=0) AND (ps.suppr_overflow=0) THEN
    exception(overflow);
  result := operand1 DIV operand2; (* integer division *)
  IF (overflow_bit=1) AND (ps.suppr_overflow=0) THEN
    exception(overflow);
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;    (* DIV *)

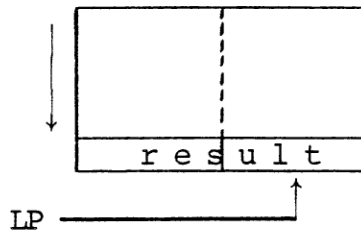
```

MODulusValue: 4B<sub>Hex</sub>IP → MOD

STACK BEFORE:



STACK AFTER:



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.

(\* MOD \*)

VAR

operand1, operand2, result : integer;

BEGIN

operand1 := stack(lp-3);

operand2 := stack(lp-1);

IF (operand2=0) AND (ps.suppr\_overflow=0) THEN  
exception(overflow);

result := operand1 - ((operand1 DIV operand2) \* operand2);

IF (overflow\_bit=1) AND (ps.suppr\_overflow=0) THEN  
exception(overflow);

stack(lp-3) := result;

lp := lp - 2;

ip := ip + 1

END; (\* MOD \*)

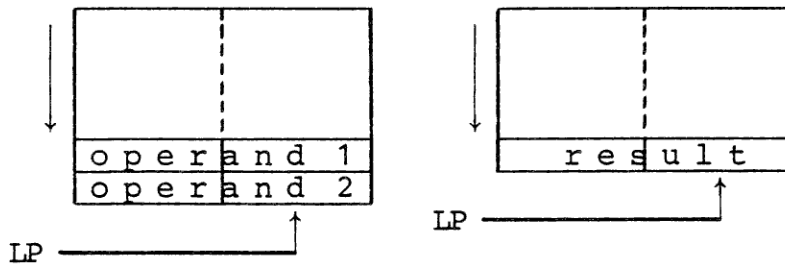
9.2.6 AND

9.2.6

ANDValue: 4C<sub>Hex</sub>IP → 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.

(\* AND \*)

```

TYPE
  bitword = PACKED ARRAY(0..15) OF bit;
VAR
  operand1, operand2, result : bitword;
  i : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  FOR i := 0 TO 15 DO
    CASE operand1(i) + operand2(i) OF
      0 : result(i) := 0;
      1 : result(i) := 0;
      2 : result(i) := 1
    END;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;      (* AND *)

```

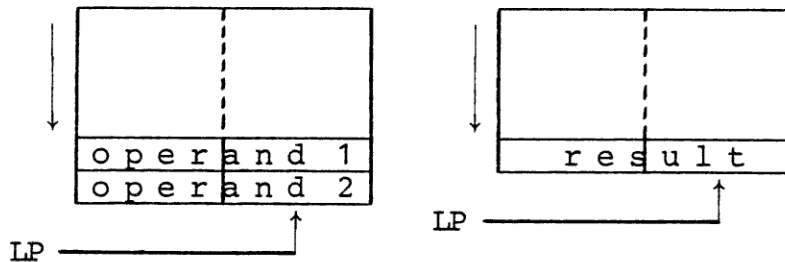
## 9.2.7 OR

9.2.7

ORValue: 4D<sub>Hex</sub>IP → 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.

(\* OR \*)

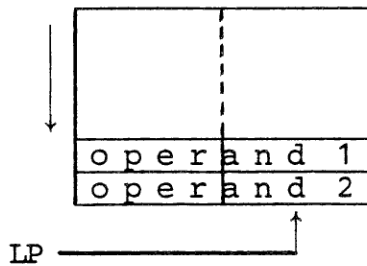
```

TYPE
  bitword = PACKED ARRAY(0..15) OF bit;
VAR
  operand1, operand2, result : bitword;
  i : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  FOR i := 0 TO 15 DO
    CASE operand1(i) + operand2(i) OF
      0 : result(i) := 0;
      1 : result(i) := 1;
      2 : result(i) := 1
    END;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;    (* OR *)

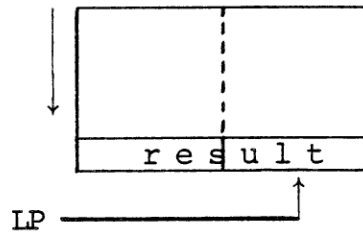
```

Shift CyclicValue: 53<sub>Hex</sub>IP → 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.

(\* SHC \*)

```

TYPE
  bitword = PACKED ARRAY(0..15) OF bit;
VAR
  operand1, result : bitword;
  operand2 : integer;
  i : integer;
  j : integer;
  b : bit;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  operand2 := operand2 MOD 16;
  IF operand2 < 0 THEN
    FOR i := -1 DOWNTO operand2 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
          b := operand1(0);
          FOR j := 1 TO 15 DO
            operand1(j-1) := operand1(j);
          operand1(15) := b;
        END;
      stack(lp-3) := operand1;
      lp := lp - 2;
      ip := ip + 1
    END;
  (* SHC *)

```

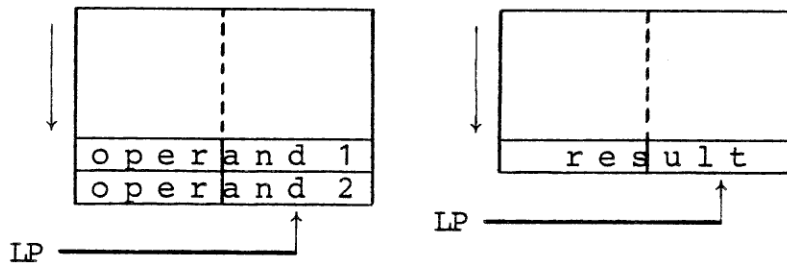
9.2.9 ULT

9.2.9

Unsigned Less ThanValue: 31<sub>Hex</sub>IP → ULT

STACK BEFORE:

STACK AFTER:



operator: operand 1 &lt; 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).

(\* ULT \*)

```

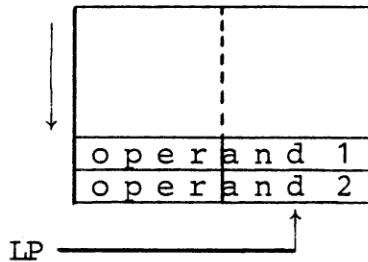
VAR
  operand1, operand2: word;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1 < operand2 THEN
    stack(lp-3) := true
  ELSE
    stack(lp-3) := false;
  lp := lp - 2;
  ip := ip + 1
END;    (* ULT *)

```

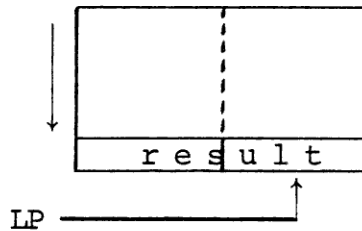


EqualValue: 32<sub>Hex</sub>IP → EQ

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).

(\* EQ \*)

```

VAR
  operand1, operand2: integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1 = operand2 THEN
    stack(lp-3) := true
  ELSE
    stack(lp-3) := false;
  lp := lp - 2;
  ip := ip + 1
END;    (* EQ *)

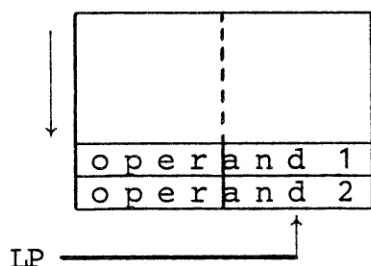
```

9.2.11 NE

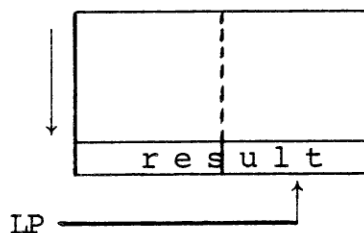
9.2.11

Not EqualValue: 33<sub>Hex</sub>IP → NE

STACK BEFORE:



STACK AFTER:

operator: operand 1  $\diamond$  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).

(\* NE \*)

```

VAR
  operand1, operand2: integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1 = operand2 THEN
    stack(lp-3) := false
  ELSE
    stack(lp-3) := true;
  lp := lp - 2;
  ip := ip + 1
END;    (* NE *)

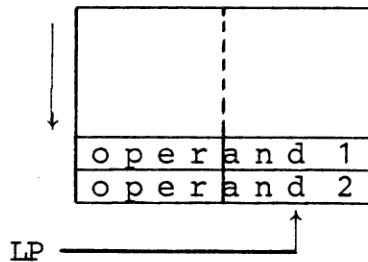
```

## 9.2.12 LT

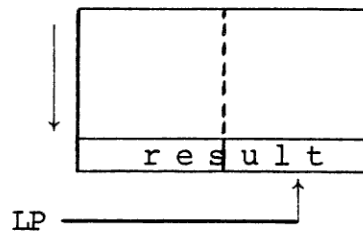
9.2.12

Less ThanValue: 34<sub>Hex</sub>IP → LT

STACK BEFORE:



STACK AFTER:



operator: operand 1 &lt; 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.

(\* LT \*)

```

VAR
  operand1, operand2 : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1 < operand2 THEN
    stack(lp-3) := true
  ELSE
    stack(lp-3) := false;
  lp := lp - 2;
  ip := ip + 1
END;    (* LT *)

```

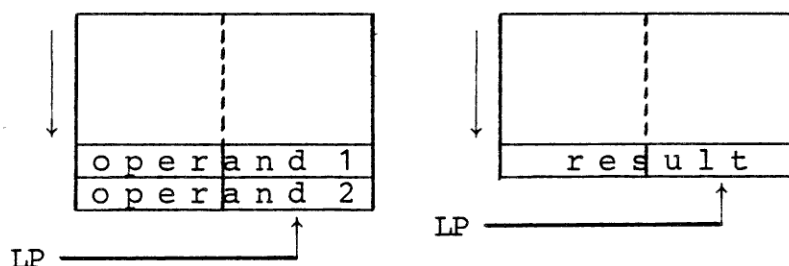
## 9.2.13 GT

9.2.13

Greater ThanValue: 35<sub>Hex</sub>IP → GT

STACK BEFORE:

STACK AFTER:



operator: operand 1 &gt; 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.

(\* GT \*)

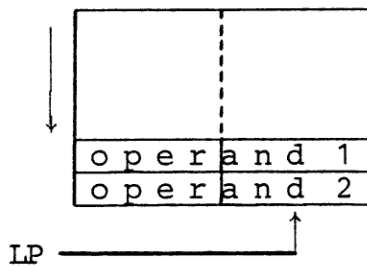
```

VAR
  operand1, operand2 : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1 > operand2 THEN
    stack(lp-3) := true
  ELSE
    stack(lp-3) := false;
  lp := lp - 2;
  ip := ip + 1
END;    (* GT *)

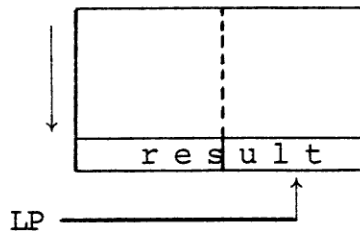
```

Less Than or EqualValue: 36<sub>Hex</sub>IP → LE

STACK BEFORE:



STACK AFTER:

operator: operand 1  $\Leftarrow$  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.

(\* LE \*)

VAR

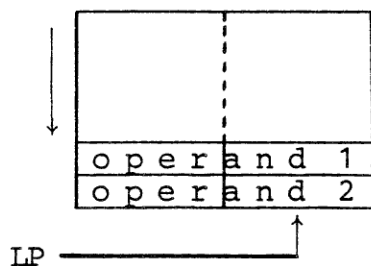
```

operand1, operand2 : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1  $\Leftarrow$  operand2 THEN
    stack(lp-3) := true
  ELSE
    stack(lp-3) := false;
  lp := lp - 2;
  ip := ip + 1
END;    (* LE *)

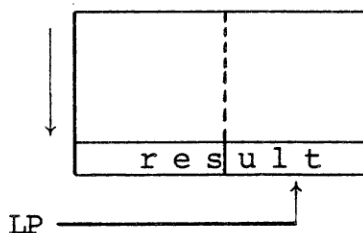
```

Greater Than or EqualValue: 37<sub>Hex</sub>IP → GE

STACK BEFORE:



STACK AFTER:

operator: operand 1  $\geq$  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.

(\* GE \*)

```

VAR
  operand1, operand2 : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF operand1  $\geq$  operand2 THEN
    stack(lp-3) := true;
  ELSE
    stack(lp-3) := false;
  lp := lp - 2;
  ip := ip + 1
END;    (* GE *)

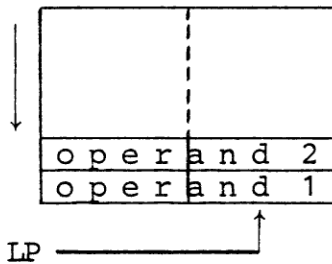
```

## 9.2.16 XOR

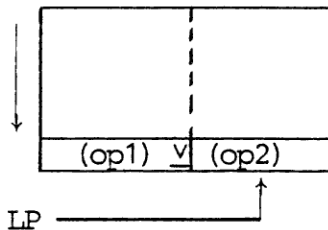
9.2.16

EXclusive ORValue: 4E<sub>Hex</sub>IP → XOR

STACK BEFORE:



STACK AFTER:



result := (operand1    operand2)    (operand1    operand2)

(\* XOR \*)

TYPE

bitword = PACKED ARRAY(0..15) OF bit;

VAR

operand1, operand2, result : bitword;

i : integer;

BEGIN

operand1 := stack(lp-3);

operand2 := stack(lp-1);

FOR i := 0 TO 15 DO

CASE operand1(i) + operand2(i) OF

0: result(i) := 0;

1: result(i) := 1;

2: result(i) := 0

END;

stack(lp-3) := result;

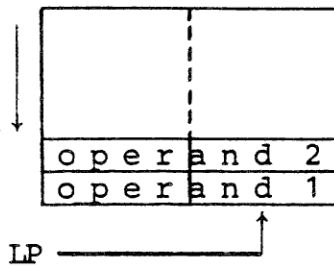
lp := lp - 2;

ip := ip + 1

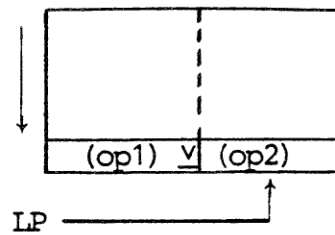
END;    (\* XOR \*)

Unsigned ADDValue: 42<sub>Hex</sub>IP → 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).

(\* UADD \*)

VAR

```

    operand1, operand2, result : word;
BEGIN
    operand1 := stack(lp-3);
    operand2 := stack(lp-1);
    result := operand1 + operand2;
    IF (carry_bit=1) AND (ps.suppr_overflow=0) THEN
        exception(overflow);
    stack(lp-3) := result;
    lp := lp - 2;
    ip := ip + 1;
END;    (* UADD *)

```

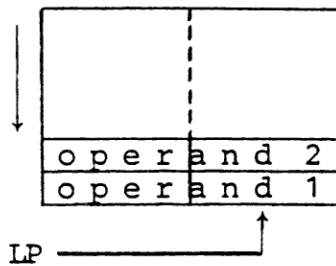


## 9.2.18 USUB

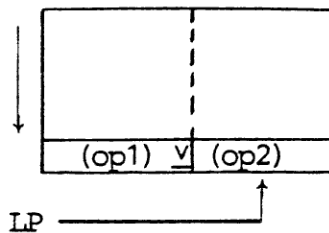
9.2.18

Unsigned SUBtractValue: 43<sub>Hex</sub>IP → 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).

(\* UADD \*)

VAR

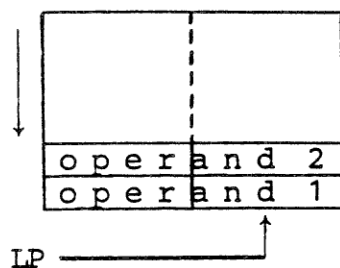
```

operand1, operand2, result : word;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  result := operand1 - operand2;
  IF (carry_bit=0) AND (ps.suppr_overflow=0) THEN
    exception(overflow);
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;    (* UADD *)

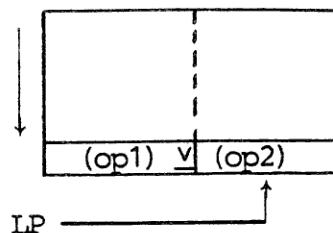
```

Unsigned MultiplyValue: 46<sub>Hex</sub>IP → 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).

(\* UMUL \*)

VAR

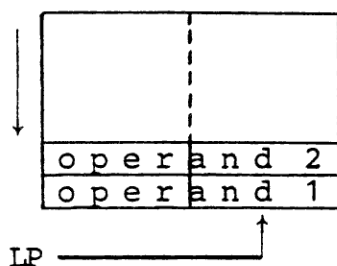
```

    operand1, operand2, result : word;
BEGIN
    operand1 := stack(lp-3);
    operand2 := stack(lp-1);
    result := operand1 * operand2;
    IF (carry_bit=1) AND (ps.suppr_overflow=0) THEN
        exception(overflow);
    stack(lp-3) := result;
    lp := lp - 2;
    ip := ip + 1
END;    (* UMUL *)

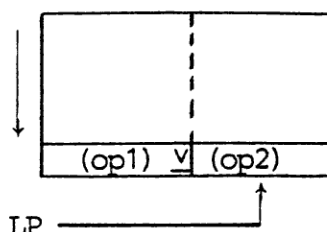
```

Unsigned DIVideValue: 47<sub>Hex</sub>IP → 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).

(\* UDIV \*)

VAR

operand1, operand2, result : word;

BEGIN

operand1 := stack(lp-3);

operand2 := stack(lp-1);

IF (operand2=0) AND (ps.suppr\_overflow=0) THEN

exception(overflow);

result := operand1 DIV operand2; (\* integer division \*)

stack(lp-3) := result;

lp := lp - 2;

ip := ip + 1

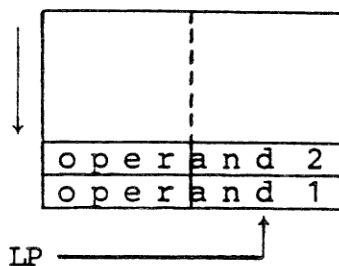
END; (\* UDIV \*)

9.2.21 UMOD

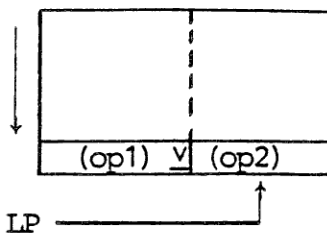
9.2.21

Unsigned MODulusValue: 48<sub>Hex</sub>IP → 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).

(\* UMOD \*)

VAR

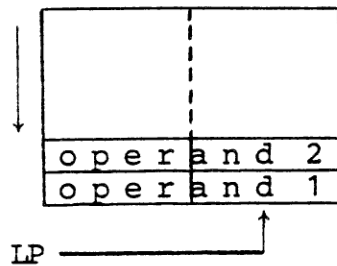
```

operand1, operand2, result : word;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  IF (operand2=0) AND (ps.suppr_overflow=0) THEN
    exception(overflow);
  result := operand1 - (operand1 DIV operand2) * operand2;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;    (* UMOD *)

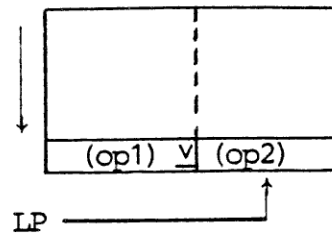
```

Modulo ADDValue: 40<sub>Hex</sub>IP → 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.

(\* MADD \*)

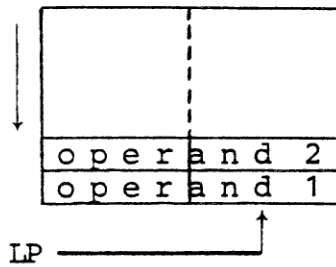
```

VAR
  operand1, operand2, result : word;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  result := operand1 + operand2;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;    (* MADD *)

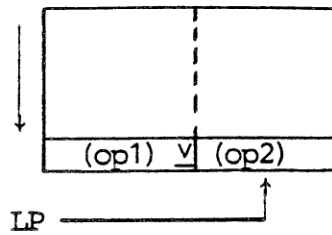
```

Modulo SUBtractValue: 41<sub>Hex</sub>IP → 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.

(\* MSUB \*)

```

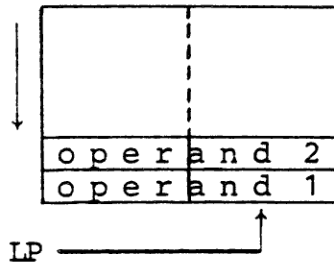
VAR
  operand1, operand2, result : word;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  result := operand1 - operand2;
  stack(lp-3) := result;
  lp := lp - 2;
  ip := ip + 1
END;      (* MSUB *)

```

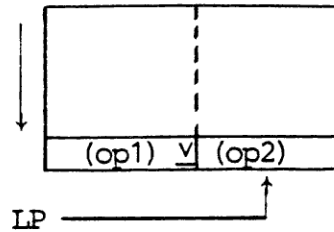
Cyclic Redundancy Check

Value: 4F<sub>Hex</sub>IP → CRC16

STACK BEFORE:



STACK AFTER:



operator: crc16

- Operand1 represents the polynomial

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

where  $a_j = \text{operand1.bit}_j$ .Note that  $\text{bit}_0$  is the most significant bit.

- Operand2 represents the polynomial

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

where  $b_j = \text{operand2.bit}_j$ .Note that  $x^{16}$  by convention is implicitly given.

The instruction delivers the remainder by the division

$$(f(x) * x^8) / g(x)$$

(\* CRC16 \*)

```

VAR
  operand1, operand2 : integer;
  i : integer;
BEGIN
  operand1 := stack(lp-3);
  operand2 := stack(lp-1);
  FOR i := 1 TO 8 DO
    IF (operand1 AND 1) = 1 THEN
      operand1 := (operand1 SHIFT (-1)) XOR operand2
    ELSE
      operand1 := operand1 SHIFT (-1);
  stack(lp-3) := operand1;
  lp := lp - 2;
  ip := ip + 1
END;      (* CRC16 *)

```



The representation of a set on the stack 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 on the stack, where "a" and "b" are greater than zero.

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

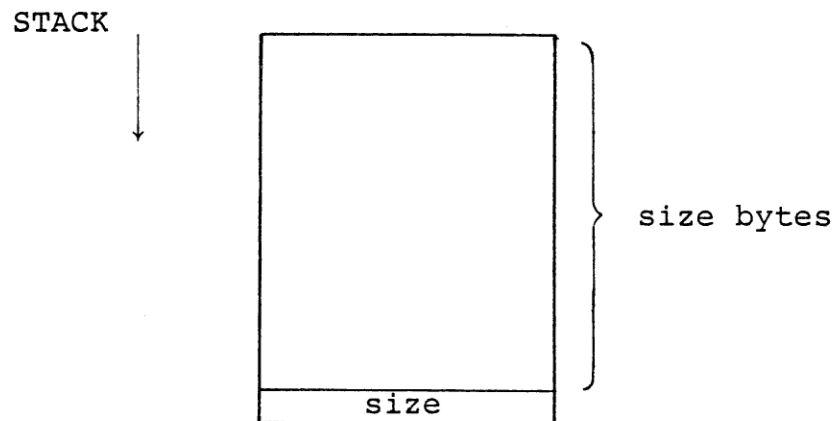


Figure 15: Stack Representation of the Set (.a..b.),  $0 \leq a \leq b$ ,  
 $\text{size} = (1 + b \text{ DIV } 16) * 2$ .

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 resumption mode. The mode is detected by means of the resume bit in the ps register of the register set. Execution in the normal mode starts with the setting of the resume bit and testing against stack overflow in the event of an interrupt.

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 (7 words) 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 a procedure call (`instruction_fetch`).

#### 9.3.1 Construct a Set from a Subrange

9.3.1

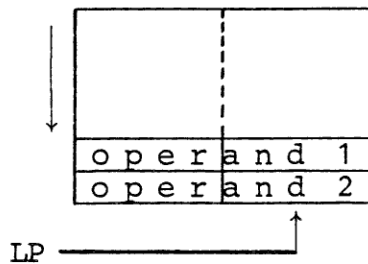
Operands 1 and 2 are retrieved, and a set is constructed and pushed on the stack as the result. The set is initialized to contain the integers in the subrange (operand 1 .. operand 2). Operand 1 must be greater than or equal to zero. If operand 2 is less than operand 1, the set will be empty.

## 9.3.1.1 SETCR

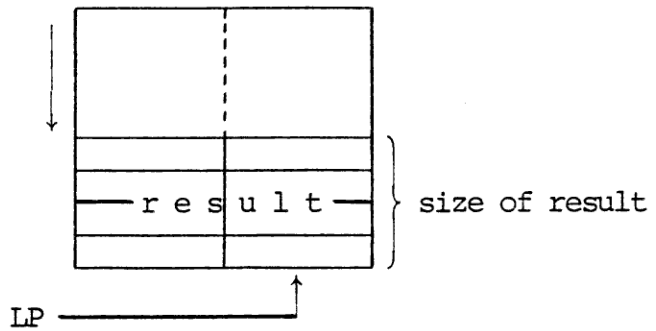
9.3.1.1

SET CRateValue: 56<sub>Hex</sub>IP → SETCR

STACK BEFORE:



STACK AFTER:



size of result: if operand2 &gt;= operand1 then:

$$(1 + \text{operand2 DIV } 16) * 2 + 2,$$

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.

(\* SETCR \*)

TYPE

```

    bit = 0..1;
    setareatype = PACKED ARRAY(0..n) OF bit;
    settype = PACKED RECORD
        setarea : setareatype;
        size : integer
    END;
    wordarray = ARRAY(0..(n DIV 16)) OF word;

```

VAR

```

    i : integer;
    result : settype;
    resultwords : wordarray;
    operand1, operand2 : word;

```

PROCEDURE change(VAR s : settype; VAR w : wordarray); EXTERNAL;

BEGIN

```

    operand1 := stack(lp-3);
    operand2 := stack(lp-1);
    IF operand1 > operand2 THEN
        result.size := 0
    ELSE
        result.size := (1 + operand2 DIV 16) * 2;
    IF operand1 < 0 THEN operand1 := 0;
    IF lp > ms - (result.size + 2) - 14 THEN exception(stack overflow)
    ELSE

```

BEGIN

```

        resume := true;
        FOR i := 0 TO operand1 - 1 DO
            result.setarea(i) := 0;
        FOR i := operand1 TO operand2 DO
            result.setarea(i) := 1;
        FOR i := operand2 + 1 TO 8 * result.size DO
            result.setarea(i) := 0;
        change(result, resultwords);
        FOR i := 0 TO result.size DIV 2 DO
            BEGIN
                stack(lp-3+2*i) := resultwords(i);
                IF interrupt_condition THEN instruction_fetch
            END;
        resume := false;
        lp := lp - 4 + result.size + 2;
        ip := ip + 1
    END

```

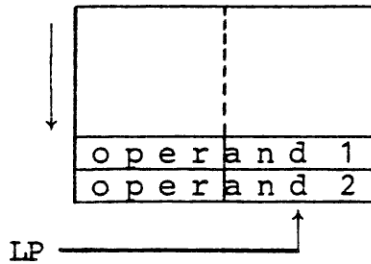
END; (\* SETCR \*)

## 9.3.1.2 SETAD

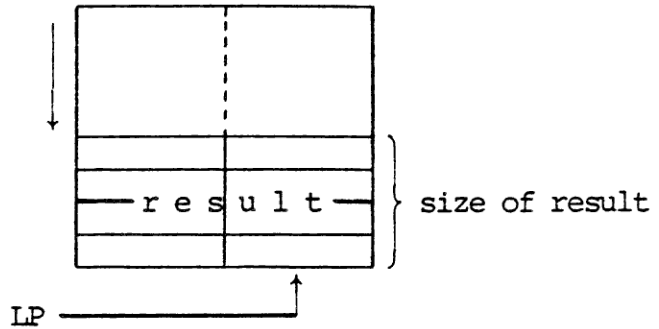
9.3.1.2

SET ADjustValue: 5F<sub>Hex</sub>IP → 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.

(\* SETAD \*)

```

TYPE
  size1, size2 : word;
  set_word      : word;
  i             : integer;
BEGIN
  resume := true;
  size2 := stack(lp-1);
  IF odd(size2) THEN
    exception(odd operand);
  size1 := stack(lp-3);
  IF size1 < size2 THEN (* unsigned compare *)
    BEGIN (* enlarge *)
      IF lp > ms - (size2-size1) - 14 THEN
        exception(stack overflow);
      FOR i := 1 TO (size2-size1) DIV 2 DO
        BEGIN
          stack(lp-3+(i-1)*2) := 0;
          IF interrupt_condition THEN
            instruction_fetch
          END;
        END
      ELSE
        BEGIN (* size1 >= size2 *)
          IF lp > ms - 14 THEN
            exception(stack overflow);
          FOR i := 1 TO (size1-size2) DIV 2 DO
            BEGIN
              set_word := stack(lp-3 - 2*i);
              IF set_word <> 0 THEN
                exception(setadjust truncation error);
              IF interrupt_condition THEN
                instruction_fetch
              END
            END
          END;
          resume := false;
          lp := lp - 2 + (size2-size1);
          stack(lp-1) := size2;
          ip := ip + 1
        END; (* SETAD *)
      END;
    END;
  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

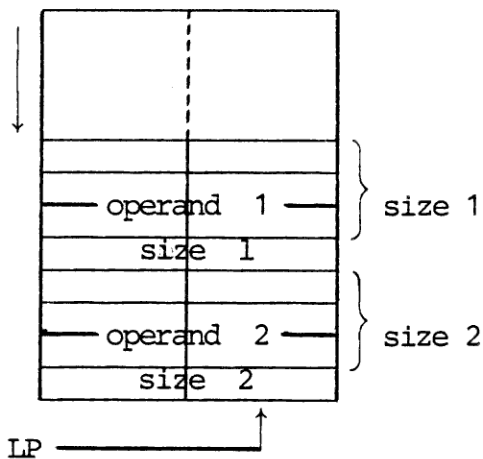
9.3.2.1

SET Union

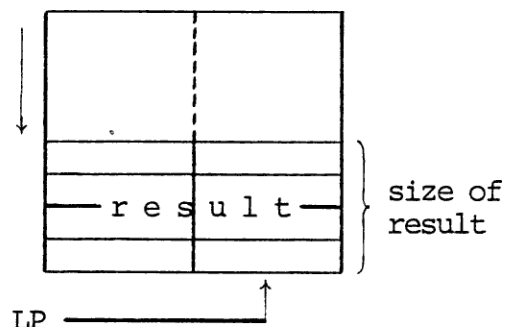
Value: 5C<sub>Hex</sub>

IP → SETUN

STACK BEFORE:



STACK AFTER:



operator: union

size of result:  $\max(\text{size1}, \text{size2}) + 2$

(\* SETUN \*)

```

VAR
  start1, start2 : addr;
  size1, size2 : word;
  i : integer;
FUNCTION min(a, b : word) : word; EXTERNAL;
FUNCTION max(a, b : word) : word; EXTERNAL;
BEGIN
  resume := true;
  IF lp > ms - 14 THEN exception(stack overflow);
  size2 := stack(lp-1);
  size1 := stack(lp-size2-3);
  start2 := lp-size2-1;
  start1 := start2-size1-2;
  FOR i := 0 TO min(size1, size2) DIV 2 - 1 DO
    BEGIN
      stack(start1+2*i) := stack(start1+2*i) OR stack(start2+2*i);
      IF interrupt_condition THEN instruction_fetch
    END;
  IF size2 = max(size1, size2) THEN
    BEGIN
      FOR i := 0 TO (size2-size1) DIV 2-1 DO
        BEGIN
          stack(start1+size1+2*i) := stack(start2+size1+2*i);
          IF interrupt_condition THEN instruction_fetch
        END;
      stack(start1+size2) := size2
    END;
  lp := lp - min(size1, size2) - 2;
  resume := false;
  ip := ip + 1
END;    (* SETUN *)

```

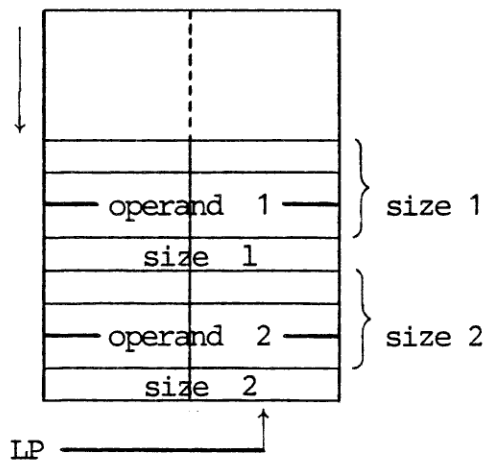


## 9.3.2.2 SETIN

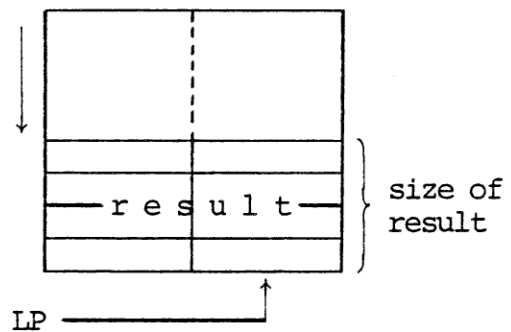
9.3.2.2

SET IntersectionValue: 5D<sub>Hex</sub>IP → SETIN

STACK BEFORE:



STACK AFTER:



operator: intersection

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

(\* SETIN \*)

VAR

start1, start2 : addr;  
size1, size2, minlength, resultlength : word;  
i : integer;

BEGIN

resume := true;

IF lp > ms - 14 THEN exception(stack overflow);

size2 := stack(lp-1);

start2 := lp - size2-1;

size1 := stack(lp-1 - size2-2);

start1 := start2 - size1 - 2;

IF size1 >= size2 THEN

BEGIN

minlength := size2;

resultlength := size1

END

ELSE

BEGIN

minlength := size1;

resultlength := size2

END;

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\_condition THEN instruction\_fetch

END;

FOR i := 0 TO (resultlength - minlength) DIV 2 - 1 DO

BEGIN

stack(start1 + minlength + 2\*i) := 0;

IF interrupt\_condition THEN instruction\_fetch

END;

stack(start1 + resultlength) := resultlength;

resume := false;

lp := lp - minlength - 2;

ip := ip + 1

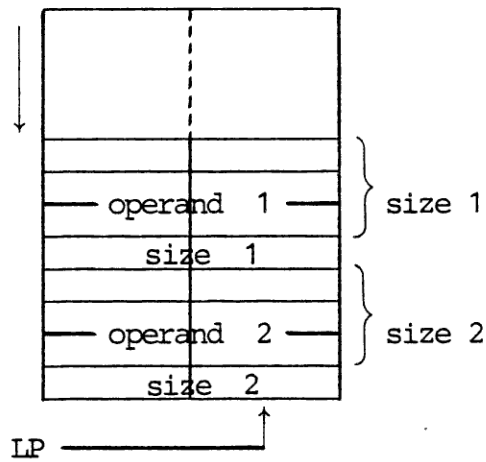
END; (\* SETIN \*)

## 9.3.2.3 SETDI

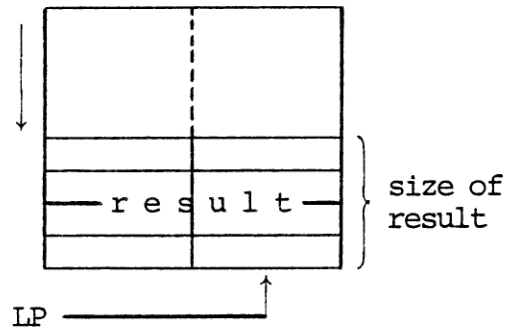
9.3.2.3

SET DifferenceValue: 5E<sub>Hex</sub>IP → SETDI

STACK BEFORE:



STACK AFTER:



operator: difference

size of result:  $\max(\text{size1}, \text{size2}) + 2$

(\* SETDI \*)

VAR

start1, start2 : addr;  
size1, size2, minlength, resultlength : word;  
i : integer;

BEGIN

resume := true;  
IF lp > ms - 14 THEN exception(stack overflow);  
size2 := stack(lp-1);  
start2 := lp - size2-1;  
size1 := stack(lp-1 - size2-2);  
start1 := start2 - size1-2;  
IF size1 >= size2 THEN

BEGIN

minlength := size2;  
resultlength := size1

END

ELSE

BEGIN

minlength := size1;  
resultlength := size2

END;

FOR i := 0 TO minlength DIV 2 - 1 DO

BEGIN

stack(start1 + 2\*i) := stack(start1 + 2\*i) AND  
COMPL (stack(start2 + 2\*i));

IF interrupt\_condition THEN instruction\_fetch

END;

IF size1 = minlength THEN

FOR i := 0 TO (resultlength - minlength) DIV 2 - 1 DO

BEGIN

stack(start1 + minlength + 2\*i) := 0;

IF interrupt\_condition THEN instruction\_fetch

END;

stack(start1 + resultlength) := resultlength;

resume := false;

lp := lp - minlength - 2;

ip := ip + 1

END; (\* SETDI \*)

### 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

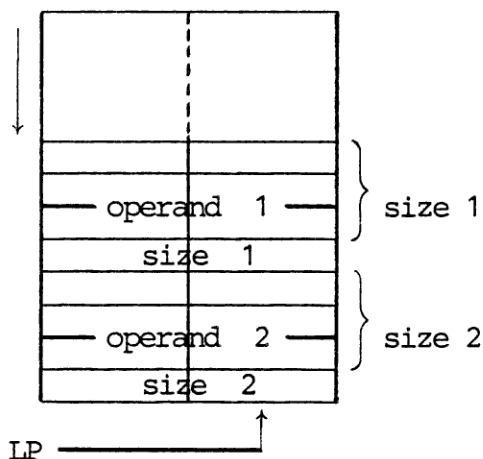
9.3.3.1

SET Equal

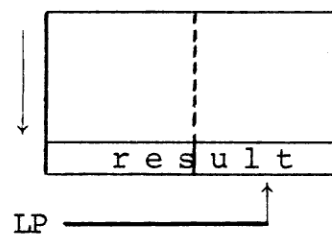
Value: 59<sub>Hex</sub>

IP → SETEQ

STACK BEFORE:



STACK AFTER:



relation : equal

(\* SETEQ \*)

```

LABEL exittest;
VAR
  i : integer;
  start1, start2 : addr;
  result : boolean := true;
  size1, size2 : word;
FUNCTION min(a, b : word) : word; EXTERNAL;
BEGIN
  resume := true;
  IF lp > ms - 14 THEN exception(stack overflow);
  size2 := stack(lp-1);
  start2 := lp - size2-1;
  size1 := stack(lp-size2-3);
  start1 := start2-size1-2;
  FOR i := 0 TO min(size1, size2) DIV 2 - 1 DO
    BEGIN
      IF stack(start1+i*2) <> stack(start2+i*2) THEN
        BEGIN
          result := false;
          GOTO exittest
        END;
      IF interrupt_condition THEN instruction_fetch
    END;
  IF size1 < size2 THEN
    BEGIN
      FOR i := 0 TO (size1-size2) DIV 2-1 DO
        BEGIN
          IF stack(start2+size1+i*2) <> 0 THEN
            BEGIN
              result := false;
              GOTO exittest
            END;
          IF interrupt_condition THEN instruction_fetch
        END
      END
    ELSE
      FOR i := 0 TO (size2-size1) DIV 2-1 DO
        BEGIN
          IF stack(start+size2+2*i) <> 0 DO
            BEGIN
              result := false;
              GOTO exittest
            END;
          IF interrupt_condition THEN instruction_fetch
        END;
      exittest: stack(start1) := result;
      lp := start 1 + 1;
      resume := false;
      ip := ip + 1
    END; (* SETEQ *)

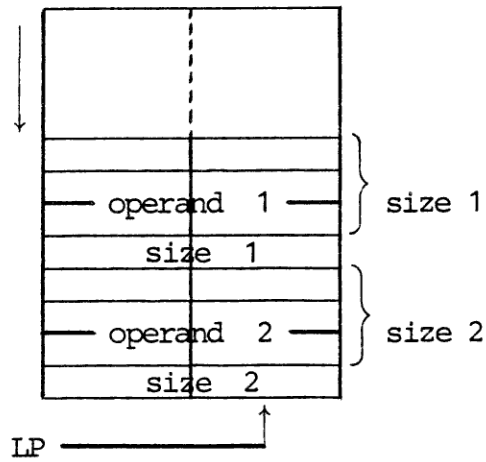
```

## 9.3.3.2 SETSB

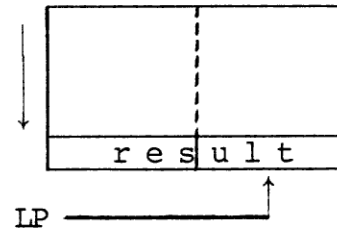
9.3.3.2

SET SuBsetValue: 5A<sub>Hex</sub>IP → SETSB

STACK BEFORE:



STACK AFTER:



relation : subset

(\* SETSB \*)

LABEL exittest;

VAR

  i : integer;

  start1, start2 : addr;

  result : boolean := true;

  size1, size2 : word;

FUNCTION min(a, b : word) : word; EXTERNAL;

BEGIN

  resume := true;

  IF lp > ms - 14 THEN exception(stack overflow);

  size2 := stack(lp-1);

  start2 := lp - size2-1;

  size1 := stack(lp-1 - size2-2);

  start1 := start2 - size1-2;

  FOR i := 0 TO min(size1, size2) DIV 2 - 1 DO

    BEGIN

      IF stack(start2+2\*i) AND stack(start1+2\*i)

      ◊ stack(start1+i\*2) THEN

        BEGIN

          result := false;

          GOTO exittest

        END;

      IF interrupt\_condition THEN instruction\_fetch

    END;

  IF size1 > size2 THEN

    FOR i := 0 TO (size1-size2) DIV 2 - 1 DO

      BEGIN

        IF stack(start1+size2+i\*2) ◊ 0 THEN

          BEGIN

            result := false;

            GOTO exittest

          END;

        IF interrupt\_condition THEN instruction\_fetch

      END;

  exittest: stack(start1) := result;

  lp := start1+1;

  resume := false;

  ip := ip + 1

END; (\* SETSB \*)

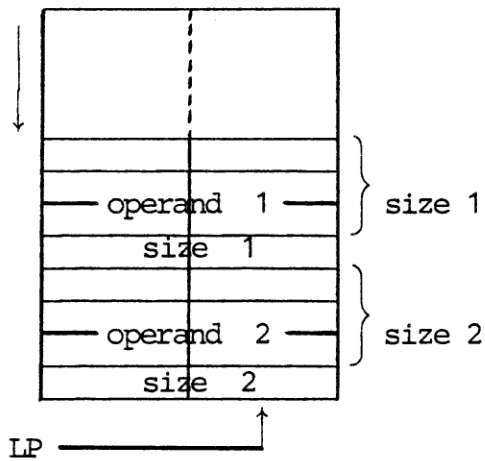


## 9.3.3.3 SETSP

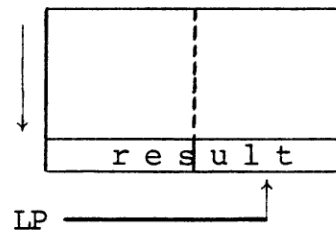
9.3.3.3

SET SuPersetValue: 5B<sub>Hex</sub>IP → SETSP

STACK BEFORE:



STACK AFTER:



relation : superset

(\* SETSP \*)

```

LABEL exittest;
VAR
  i : integer;
  start1, start2 : addr;
  result : boolean := true;
  size1, size2 : word;
FUNCTION min(a, b : word) : word; EXTERNAL;
BEGIN
  resume := true;
  IF lp > ms - 14 THEN exception(stack overflow);
  size2 := stack(lp-1);
  start2 := lp - size2-1;
  size1 := stack(lp-1 - size2-2);
  start1 := start2 - size1-2;
  FOR i := 0 TO min(size1, size2) DIV 2 - 1 DO
    BEGIN
      IF stack(start1+i*2) AND stack(start2+i*2)
        <> stack(start2+i*2) THEN
        BEGIN
          result := false;
          GOTO exittest
        END;
      IF interrupt_condition THEN instruction_fetch
    END;
  IF size2 > size1 THEN
    FOR i := 0 TO (size2-size1) DIV 2 - 1 DO
      BEGIN
        IF stack(start2 + size1 + 2*i) <> 0 THEN
          BEGIN
            result := false;
            GOTO exittest
          END;
        IF interrupt_condition THEN instruction_fetch
      END;
    exittest: stack(start1) := result;
    lp := start1 + 1;
    resume := false;
    ip := ip + 1
  END;  (* SETSP *)

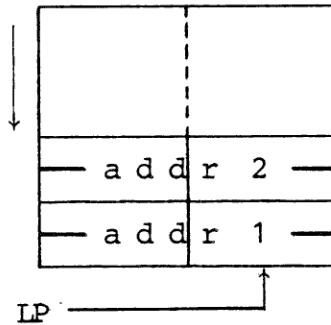
```

9.3.3.4 TEQAD

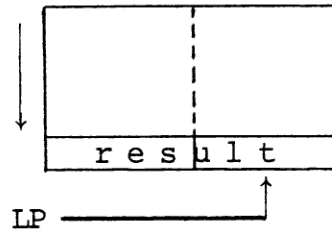
9.3.3.4

Test Equal AddressesValue: 3C<sub>Hex</sub>IP → TEQAD

STACK BEFORE:



STACK AFTER:



(\* TEQAD \*)

VAR

addr1, addr2 : addr;

result : boolean;

BEGIN

addr1.base := stack(lp-3);

addr1.disp := stack(lp-1);

addr2.base := stack(lp-7);

addr2.disp := stack(lp-5);

result := (nil(addr1) AND nil(addr2)) OR (addr1 = addr2);

stack(lp-7) := result;

lp := lp - 6;

ip := ip + 1

END; (\*TEQAD \*)

### 9.3.4 Test for Membership of a Set

9.3.4

Operand 1 is interpreted as a set. Operand 2 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 SETM

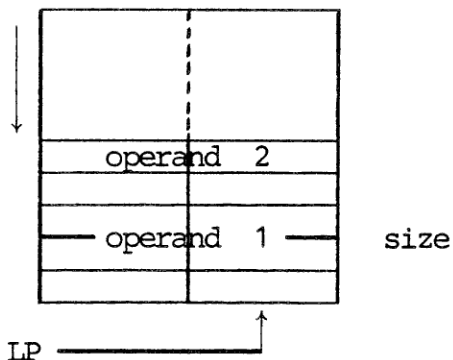
9.3.4.1

SET Test Membership

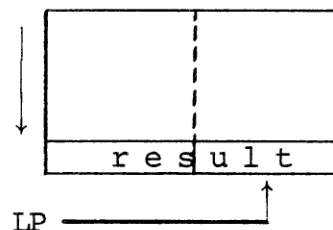
Value: 57<sub>Hex</sub>

IP → SETM

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).

(\* SETTM \*)

9.3.3.4

```
VAR
  size : word;
  operand : word;
  work : word;
  boo : boolean;
BEGIN
  boo := false;
  size := stack(lp-1);
  operand := stack(lp-1-size-2);
  IF operand < size*8 THEN
    BEGIN
      work := stack(lp-1-size+(operand DIV 16) * 2);
      work := work shift(operand AND 15);
      boo := work < 0;
    END;
  stack(lp-1-size-2) := boo;
  lp := lp - size - 2;
  ip := ip + 1
END;  (*SETTM *)
```

10. INPUT/OUTPUT

10.

The execution of an I/O instruction, like that of any RC3502 machine instruction, presupposes a process incarnation environment, with a register set that points out an incarnation stack (see subsection 3.6.1 and section 4.1).

I/O instructions can operate on devices and process incarnation environments (either the incarnation that executes the I/O instruction or other process incarnations). I/O instructions can also perform functions on devices.

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 be a subset of the above register set. The structure of the individual registers is device dependent.

### Algorithmic Descriptions

It is assumed that `field_type` is either a `byte_bit` type or a `word_bit` type, where

```

bit = 0..1;
bit_byte = PACKED ARRAY(0..7) OF bit;
bit_word = PACKED ARRAY(0..15) OF bit;
```

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;

device_type = RECORD
    function : function_type;
    control  : field_type;
    status_in : field_type;
    status_out : field_type;
    data_in : field_type;
    data_out : field_type;
    eoi : bit; (* end of information *)
    interrupt : bit
END;
```

The run-time environment which the I/O instructions presuppose and on which they operate is expressed by the following variables:

```

device : ARRAY(0..123) OF device_type;
reg_set : ARRAY(0..123) OF register_set_type;
incarnation_descriptor : incarnation_descriptor_type;
```

The type message is defined in subsection 3.7.2.1.

The register\_set\_type is defined in subsection 3.6.1.1, and the incarnation\_descriptor\_type in subsection 3.7.3.1.

The block instructions interpret the memory as being of type

```
buffer = ARRAY(0..32767) OF byte;
```

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

```
first ≤ last < top (* signed comparison *)
```

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.

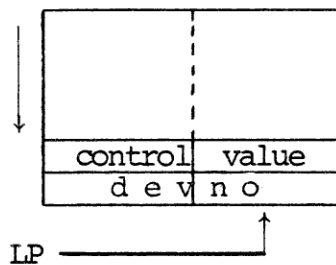
10.1.1 IOWC

10.1.1

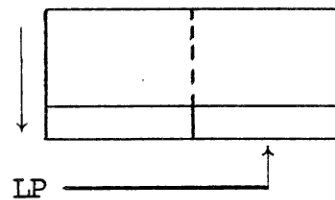
Input/Output Write ControlValue: 21<sub>Hex</sub>IP → 

|      |
|------|
| IOWC |
|------|

STACK BEFORE:



STACK AFTER:



devices (devno) :

| control | value |
|---------|-------|
|         |       |
|         |       |
|         |       |
|         |       |
|         |       |
|         |       |
|         |       |

(\* IOWC \*)

```

VAR
  devno : integer;
BEGIN
  devno := stack (lp-1);
  device(devno).function := write control;
  device(devno).control := stack(lp-3);
  lp := lp - 4;
  ip := ip + 1
END;  (* IOWC *)

```



10.2 Write Word

10.2

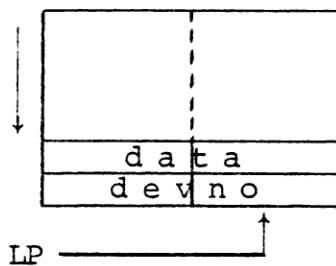
The parameter is interpreted as a device number. The value of the operand is transferred to the data out register of the device. On end of information, the value of the eoi register is transferred to the eoi field of the ps register.

10.2.1 IOWW

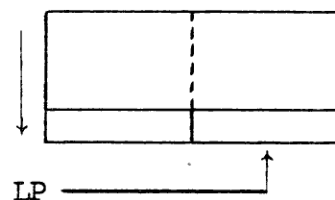
10.2.1

Input/Output Write WordValue: 24<sub>Hex</sub>IP → IOWW

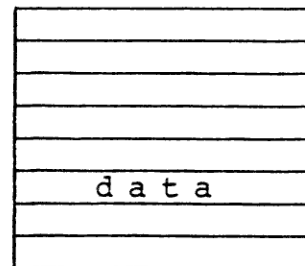
STACK BEFORE:



STACK AFTER:



devices(devno):



(\* IOWW \*)

VAR

devno : integer;

BEGIN

devno := stack (lp-1);

device(devno).function := write\_data;

IF device(devno).data\_out := stack(lp-3);

IF device(devno).eoi = 1 THEN ps.eoi := 1;

lp := lp - 4;

ip := ip + 1

END; (\* IOWW \*)

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 data\_out, status\_out, 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

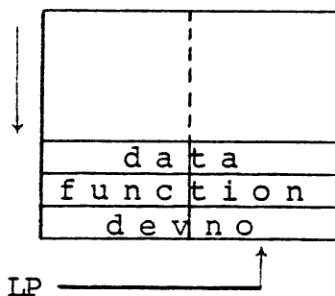
10.3.1

Input/Output General Output

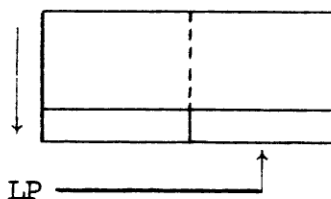
Value: 25<sub>Hex</sub>

IP → Iogo

STACK BEFORE:



STACK AFTER:



devices (devno) :

|               |   |                                                                |
|---------------|---|----------------------------------------------------------------|
| write_control | → | <div style="border: 1px solid black; padding: 2px;">data</div> |
| read_status   | → | <div style="border: 1px solid black; padding: 2px;">data</div> |
| write_data    | → | <div style="border: 1px solid black; padding: 2px;">data</div> |
|               |   | <div style="border: 1px solid black; padding: 2px;"> </div>    |
|               |   | <div style="border: 1px solid black; padding: 2px;"> </div>    |

(\* IOGO \*)

VAR

devno : integer;

function : function\_type;

data : word;

BEGIN

devno := stack(lp-1);

function := stack(lp-3);

data := stack(lp-5);

device(devno).function := function

device(devno).status\_out := data

IF device(devno).eoi = 1 THEN ps.eoi := 1;

lp := lp - 6;

ip := ip + 1

END; (\* IOGO \*)

10.4      Read Status

10.4

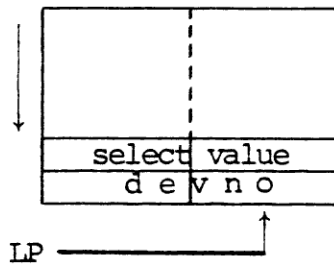
The first operand is interpreted as a device number. The result is the status information selected by the second operand from the device. First the second operand is transferred to the status\_out register of the device, and then the contents of the status\_in register are transferred as the result.

10.4.1      IORS

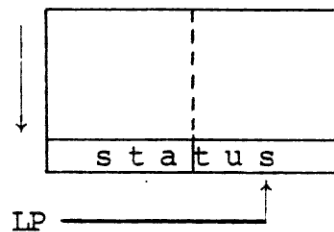
10.4.1

Input/Output Read StatusValue: 22<sub>Hex</sub>IP → IORS

STACK BEFORE:



STACK AFTER:



devices(devno):

|              |
|--------------|
|              |
|              |
| ( result )   |
| select value |
|              |
|              |
|              |
|              |

(\* IORS \*)

VAR

    devno : integer;

BEGIN

    devno := stack (lp-1);

    device(devno).function := read\_status;

    device(devno).status\_out := stack(lp-3);

    stack(lp-3) := device(devno).status\_in;

    lp := lp - 2;

    ip := ip + 1

END; (\* IORS \*)

10.5      Read Word

10.5

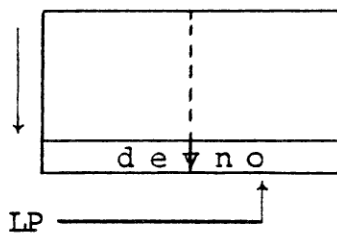
The operand is interpreted as a device number. The result is the contents of the data\_in register of the device. On end of information, the value of the eoi register is transferred to the eoi field of the ps register.

10.5.1      IORW

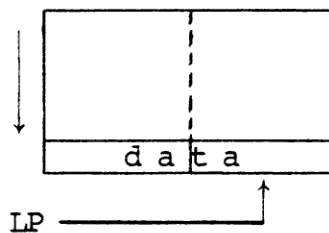
10.5.1

Input/Output Read WordValue: 23<sub>Hex</sub>IP → IORW

STACK BEFORE:



STACK AFTER:



devices (devno):

|             |
|-------------|
|             |
|             |
|             |
|             |
| ( d a t a ) |
|             |
|             |
|             |

(\* IORW \*)

VAR

  devno : integer;

BEGIN

  devno := stack (lp-1);

  device(devno).function := read\_data;

  IF device(devno).eoi = 1 THEN ps.eoi := 1;

  stack(lp-1) := device(devno).data\_in;

  ip := ip + 1

END; (\* IORW \*)



10.6 General Input

10.6

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 transferred to the status\_out register of the device, and the resulting word is obtained, according to the function, from the data\_in or status\_in register.

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

10.6.1 IOGI

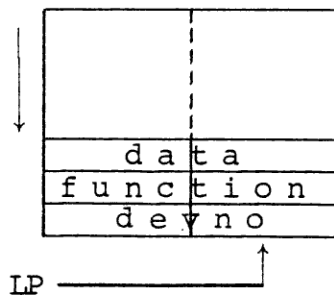
10.6.1

Input/Output General Input

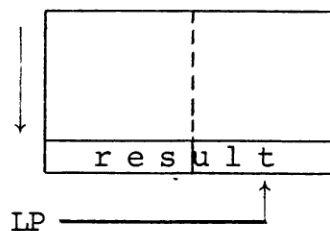
Value: 26<sub>Hex</sub>

IP → IOGI

STACK BEFORE:



STACK AFTER:



devices(devno):

|               |            |
|---------------|------------|
|               |            |
| read_status → | ( result ) |
|               | data       |
| read_data →   | ( result ) |
|               |            |
|               |            |
|               |            |

(\* IOGI \*)

VAR

  devno : integer;

  function : function\_type;

  data : word;

BEGIN

  devno := stack (lp-1);

  function := stack(lp-3);

  data := stack(lp-5);

  device(devno).function := function;

  device(devno).status\_out := data;

  IF device(devno).eoi = 1 THEN ps.eoi := 1;

  stack(lp-5) := device(devno).status\_in

  lp := lp - 4;

  ip := ip + 1

END; (\* IOGI \*)

### 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 level defined in the incarnation descriptor as its device number is assigned the value = 0.

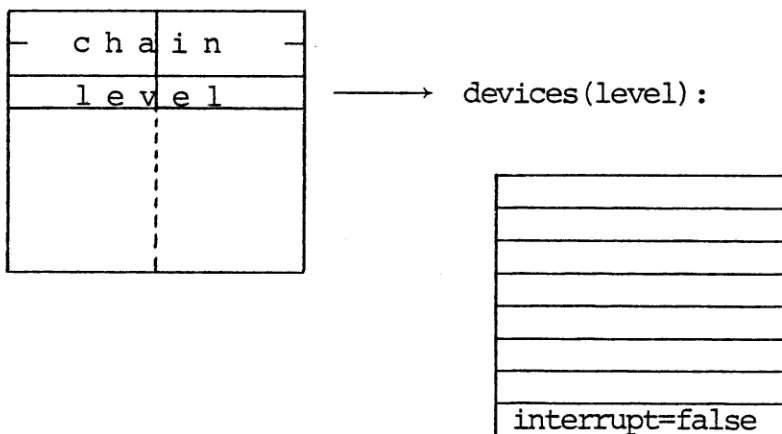
### 10.7.1 IOCCI

### 10.7.1

Input/Output Clear Current Interrupt    Value: 29<sub>Hex</sub>

IP → IOCCI

incarnation descriptor:



(\* IOCCI \*)

```

VAR
    devno: byte;
BEGIN
    devno := membyte(gf+level);
    IF ps.to = 0 THEN
        device(devno).interrupt := 0;
        ip := ip + 1
    END;    (* IOCCI *)

```

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 level defined in the incarnation descriptor as its device number is assigned the value 0, whereupon the next instruction on this level is executed.

10.8.1      IONCI

10.8.1

Input/Output Execute NextValue: 27<sub>Hex</sub>Instruction After Clearing InterruptIP → 

|       |      |
|-------|------|
| IONCI | next |
|-------|------|

incarnation descriptor:

|           |
|-----------|
| c h a i n |
| l e v e l |
|           |

→ devices(level):

|                 |
|-----------------|
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
| interrupt=false |

(\* IONCI \*)

VAR

devno : byte;

BEGIN

devno := membyte(gf+level);

IF ps.to = 0 THEN

device(devno).interrupt := 0;

ip := ip + 1;

GOTO execute\_next (\* next instruction on same level \*)

END; (\* IONCI \*)

10.9 Get Current Device Address

10.9

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.

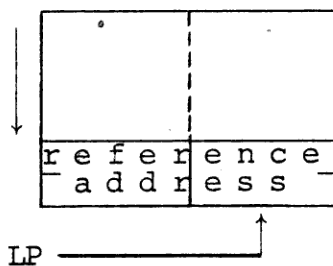
10.9.1 IOCDA

10.9.1

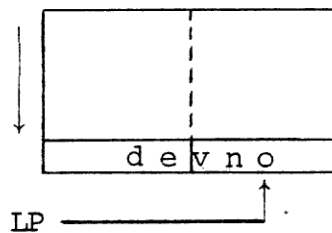
Input/Output Get Current Device Address Value: 2A<sub>Hex</sub>

IP → IOCDA

STACK BEFORE:



STACK AFTER:



(\* IOCDA \*)

```

VAR
  ref_addr : addr; (* address of reference *)
  ref      : addr; (* address of message *)
  kind     : integer;
  devno    : integer;
BEGIN
  ref_addr := memaddr (lp-3);
  ref      := memaddr (ref_addr);
  IF ref.base.nilbit = 1 THEN exception(nil)
  kind := mem(ref+messagekind);
  IF kind < 0 THEN
    BEGIN
      devno := kind AND 127;
      stack(lp-3) := devno;
      lp := lp-2;
      ip := ip + 1
    END;
  ELSE exception(not channel message)
END; (* IOCDA *)

```

10.10 Initialize Block Transfer

10.10

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

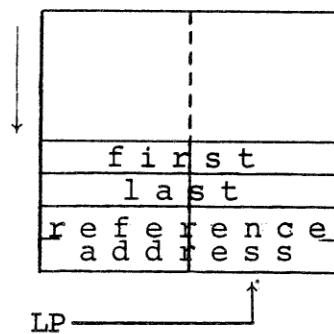
10.10.1 IOIBX

10.10.1

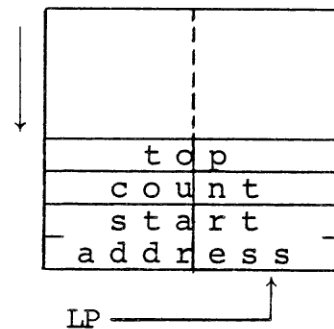
Input/Output Initialize Block Xfer      Value: 2B<sub>Hex</sub>

IP → IOIBX

STACK BEFORE:



STACK AFTER:



(\* IOIBX \*)

```

VAR
  first      : word;
  ref_addr   : addr;
  top        : word;
  count      : word;
  start_addr : addr;
  ref        : addr;
  kind       : integer;
  m_size     : word;
BEGIN
  ref_addr := memaddr(lp-3);
  ref      := memaddr(ref_addr);
  IF ref.base.nilbit = 1 THEN
    exception(nil)
  top := stack(lp-5)+1; (* last+1 *)
  first := stack(lp-7);
  kind := mem(ref+messagekind);
  IF kind < 0 THEN
    exception(not data message)
  m_size := mem(ref+size);
  IF m_size < ((top+1)DIV2) THEN (* unsigned *)
    exception(size too small)
  IF first < top THEN (* unsigned *)
    BEGIN
      count := top-first;
      start_addr := memaddr(ref+start);
      start_addr.disp := start_addr.disp + first;
      memaddr(lp-3) := start_addr;
      stack(lp-5) := count;
      stack(lp-7) := top;
      ip := ip+1
    END;
  ELSE
    exception(last < first)
END; (* IOIBX *)

```

10.11 Write Block of Bytes

10.11

A block of bytes is transferred from a buffer, one byte at a time, to the data\_out register of the device with the current interrupt level as its device number, until the buffer is empty or a timeout occurs or the eoi register of the device contains the value 1.

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

Finally, 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 written.

For each byte transferred the instruction code and the stack parameters remain in the register set. The next instruction to be executed is selected in accordance with the interrupt condition.



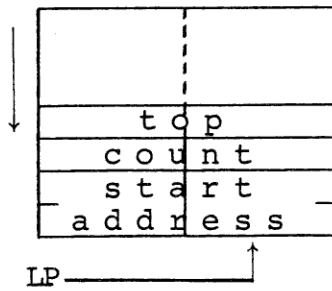
10.11.1 IOWBB

10.11.1

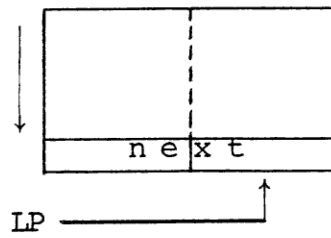
Input/Output Write Block of BytesValue: 73<sub>Hex</sub>

IP → IOWBB

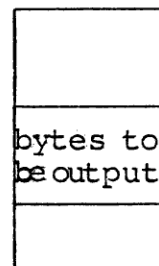
STACK BEFORE:



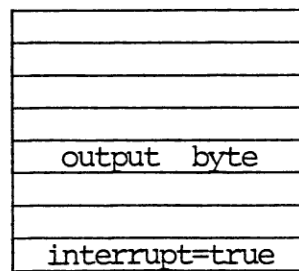
STACK AFTER:



MEMORY:



device (current level):



```
(* IOWBB, single step *)
```

```
CONST
    iowbb = #h8373;
VAR
    next : addr;
    count : integer;
    top : word;
PROCEDURE dumpregisters(level : integer);
EXTERNAL;
LABEL
    ioblockfinis, ioblocktimeout;
BEGIN
    IF ip.base <> iowbb THEN
        BEGIN (* first step *)
            IF curlevel = 0 THEN
                exception(block i/o level 0)
            ps.eoi := 0;
            dumpregisters(curlevel);
            (* change mode *)
            ib := iowbb; (* ip.base *)
            lm := stack(lp-7); (* top = last+1 *)
            ic := stack(lp-5); (* count *)
            sf := stack(lp-3); (* next_address.base *)
            lu := stack(lp-1)-1; (* next_address.disp-1 *)
            GOTO fetch1
        END
    ELSE
        BEGIN
            WITH device(curlevel) DO
                BEGIN
                    IF ps.to = 1 THEN GOTO ioblocktimeout;
                    interrupt := 0; (* clear interrupt *)
                    next.base := sf;
                    next.disp := lu;
                    count := ic;
                    top := lm;
                    function := write_data;
                    data_out := membyte(next+1);
                    next := next+1;
                    IF eoi = 1 THEN
                        BEGIN
                            ps.eoi := 1;
                            GOTO ioblockfinis
                        END;
                    count := count-1;
                    IF count > 0 THEN
                        BEGIN
                            lu := next.disp;
                            ic := count;
                            GOTO fetch
                        END;
                    ioblockfinis:
                        interrupt := 1;
```

```

ioblocktimeout:
    lm := mem(gf+dump1m);
    lu := mem(gf+dump1u)-6;
    sf := mem(gf+dumpsf);
    ib := mem(gf+dumpib);
    ic := mem(gf+dumpic)+1;
    stack(lp-1) := top-count
END (* WITH *)
END (* ELSE *)
END; (* IOWBB, single step *)

```

10.12 Write Block of Bytes and Clear

10.12

This instruction resembles the Write Block of Bytes instruction except that the current interrupt level is cleared after the last write function is performed.

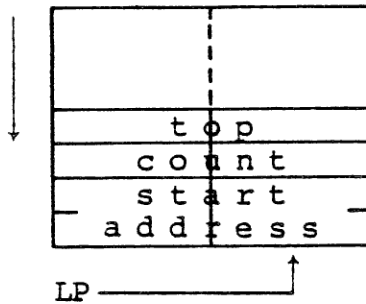
10.12.1 IOWBBC

10.12.1

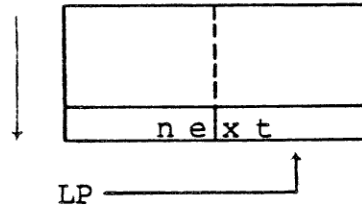
Input/Output Write Block of Bytes and Clear Value: 72<sub>Hex</sub>

IP → IOWBBC

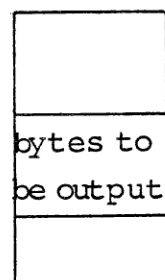
STACK BEFORE:



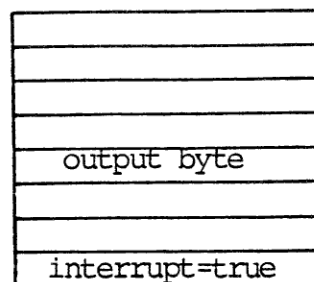
STACK AFTER:



MEMORY:



device (current level):



(\* IOWBBC, single step \*)

```

CONST
  iowbbc = #h8372;
VAR
  next      : addr;
  count     : integer;
  top       : word;
PROCEDURE dumpregisters(level : integer);
EXTERNAL;
LABEL
  ioblockfinis;
BEGIN
  IF ip.base <> iowbbc THEN
    BEGIN (* first step *)
      IF curlevel = 0 THEN
        exception(block i/o level 0)
      ps.eoi := 0;
      dumpregisters(curlevel);
      (* change mode *)
      ib := iowbbc; (* ip.base *)
      lm := stack(lp-7); (* top = last+1 *)
      ic := stack(lp-5); (* count *)
      sf := stack(lp-3); (* next_address.base *)
      lu := stack(lp-1)-1; (* next_address.disp-1 *)
      GOTO fetch1
    END
  ELSE
    BEGIN
      WITH device(curlevel) DO
        BEGIN
          IF ps.to = 1 THEN GOTO ioblockfinis;
          interrupt := 0; (* clear interrupt *)
          next.base := sf;
          next.disp := lu;
          count := ic;
          top := lm;
          function := write_data;
          data_out := membyte(next+1);
          next := next+1;
          IF eoi = 1 THEN
            BEGIN
              ps.eoi := 1
              GOTO ioblockfinis
            END;
          count := count-1;
          IF count > 0 THEN
            BEGIN
              lu := next.disp;
              ic := count;
              GOTO fetch
            END;
          END;
        END
      END
    END
  END

```

```
      ioblockfinis:
          lm := mem(gf+dump1m);
          lu := mem(gf+dump1u)-6;
          sf := mem(gf+dumpsf);
          ib := mem(gf+dumpib);
          ic := mem(gf+dumpic)+1;
          stack(lp-1) := top-count
      END (* WITH *)
  END (* ELSE *)
END; (* IOWBBC, single step *)
```

10.13 Write Block of Words

10.13

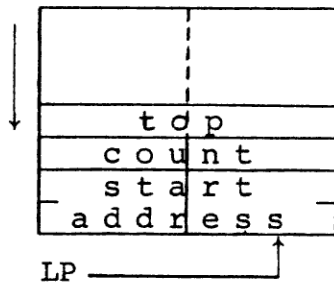
This instruction resembles the Write Block of Bytes instruction (section 10.11). A block of words is transferred, one word at a time, to the data\_out register of the device from the buffer.

10.13.1 IOWBW

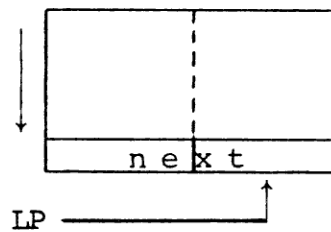
10.13.1

Input/Output Write Block of WordsValue: 77<sub>Hex</sub>IP → IOWBW

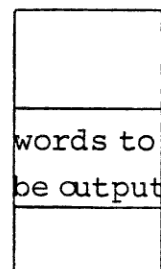
STACK BEFORE:



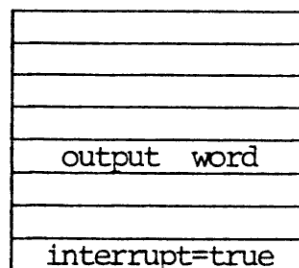
STACK AFTER:



MEMORY:



device (current level):



(\* IOWBW, single step \*)

```

CONST
    iowbw = #h8377;
VAR
    next : addr;
    count : integer;
    top : word;
PROCEDURE dumpregisters(level : integer);
EXTERNAL;
LABEL
    ioblockfinis, ioblocktimeout;
BEGIN
    IF ip.base <> iowbw THEN
        BEGIN (* first step *)
            IF curlevel = 0 THEN
                exception(block i/o level 0)
            ps.eoi := 0;
            dumpregisters(curlevel);
            (* change mode *)
            ib := iowbw; (* ip.base *)
            lm := stack(lp-7); (* top = last+1 *)
            ic := stack(lp-5); (* count *)
            sf := stack(lp-3); (* next_address.base *)
            lu := stack(lp-1)-1; (* next_address.disp-1 *)
            GOTO fetch1
        END
    ELSE
        BEGIN
            WITH device(curlevel) DO
                BEGIN
                    IF ps.to = 1 THEN GOTO ioblocktimeout;
                    interrupt := 0; (* clear interrupt *)
                    next.base := sf;
                    next.disp := lu;
                    count := ic;
                    top := lm;
                    IF count = 1 THEN
                        GOTO ioblockfinis;
                    function := write data;
                    data_out := membyte(next+1);
                    next := next+2;
                    IF eoi = 1 THEN
                        BEGIN
                            ps.eoi := 1
                            GOTO ioblockfinis
                        END;
                    count := count-2;
                    IF count > 1 THEN
                        BEGIN
                            lu := next.disp;
                            ic := count;
                            GOTO fetch
                        END;
                    ioblockfinis:
                        interrupt := 1;
                END
            END
        END
    END

```



```
        ioblocktimeout:
            lm := mem(gf+dumplm);
            lu := mem(gf+dumplu)-6;
            sf := mem(gf+dumpsf);
            ib := mem(gf+dumpib);
            ic := mem(gf+dumpic)+1;
            stack(lp-1) := top-count
        END (* WITH *)
    END (* ELSE *)
END; (* IOWBW, single step *)
```

10.14 Write Block of Words and Clear

10.14

This instruction resembles the Write Block of Bytes instruction except that the current interrupt level is cleared after the last write function is performed.

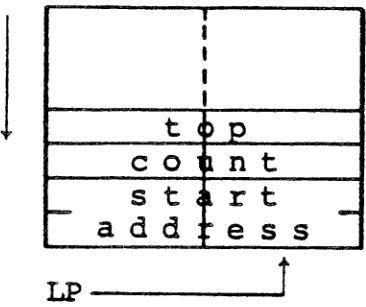
10.14.1 IOWBWC

10.14.1

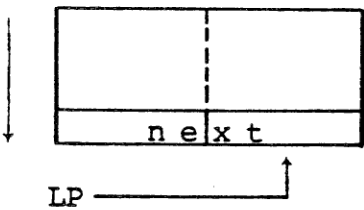
Input/Output Write Block of Words and Clear Value: 76<sub>Hex</sub>

IP → IOWBWC

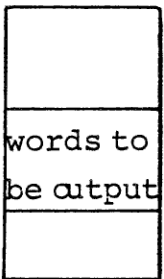
STACK BEFORE:



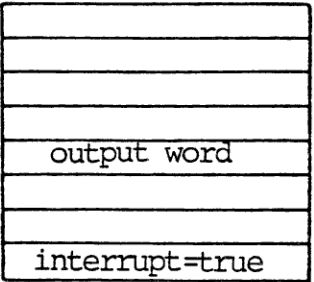
STACK AFTER:



MEMORY:



device (current level):



```
(* IOBWC, single step *)
```

```
CONST
```

```
  iowbwc = #h8376;
```

```
VAR
```

```
  next      : addr;
```

```
  count     : integer;
```

```
  top       : word;
```

```
PROCEDURE dumpregisters(level : integer);
```

```
EXTERNAL;
```

```
LABEL
```

```
  ioblockfinis;
```

```
BEGIN
```

```
  IF ip.base <> iowbwc THEN
```

```
    BEGIN (* first step *)
```

```
      IF curlevel = 0 THEN
```

```
        exception(block i/o level 0)
```

```
        ps.eoi := 0;
```

```
        dumpregisters(curlevel);
```

```
        (* change mode *)
```

```
        ib := iowbwc; (* ip.base *)
```

```
        lm := stack(lp-7); (* top = last+1 *)
```

```
        ic := stack(lp-5); (* count *)
```

```
        sf := stack(lp-3); (* next_address.base *)
```

```
        lu := stack(lp-1)-1; (* next_address.disp-1 *)
```

```
        GOTO fetch1
```

```
      END
```

```
    ELSE
```

```
      BEGIN
```

```
        WITH device(curlevel) DO
```

```
          BEGIN
```

```
            IF ps.to = 1 THEN GOTO ioblockfinis;
```

```
            interrupt := 0; (* clear interrupt *)
```

```
            next.base := sf;
```

```
            next.disp := lu;
```

```
            count := ic;
```

```
            top := lm;
```

```
            IF count = 1 THEN
```

```
              GOTO ioblockfinis;
```

```
            function := write_data;
```

```
            data_out := memword(next+1);
```

```
            next := next+2;
```

```
            IF eoi = 1 THEN
```

```
              BEGIN
```

```
                ps.eoi := 1
```

```
                GOTO ioblockfinis
```

```
              END;
```

```
            count := count-2;
```

```
            IF count > 1 THEN
```

```
              BEGIN
```

```
                lu := next.disp;
```

```
                ic := count;
```

```
                GOTO fetch
```

```
              END;
```

```
      ioblockfinis:
        lm := mem(gf+dumplm);
        lu := mem(gf+dumplu)-6;
        sf := mem(gf+dumpsf);
        ib := mem(gf+dumpib);
        ic := mem(gf+dumpic)+1;
        stack(lp-1) := top-count
      END (* WITH *)
    END (* ELSE *)
  END; (* IOWBWC, single step *)
```

10.15 Read Block of Bytes

10.15

A block of bytes is transferred to a buffer, one byte at a time, from the data\_in register of the device with the current interrupt 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. This value is then 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.

Finally, 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 read.

For each byte transferred the instruction code and the stack parameters remain in the register set. The next instruction to be executed is selected in accordance with the interrupt condition.

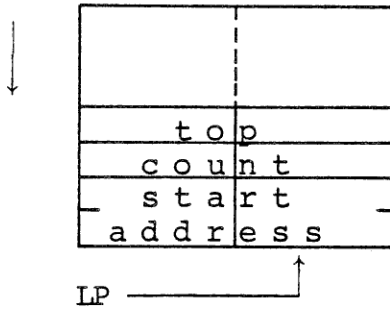
## 10.15.1 IORBB

10.15.1

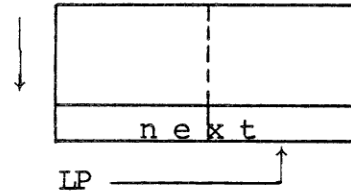
Input/Output Read Block of BytesValue: 71<sub>Hex</sub>

IP → IORBB

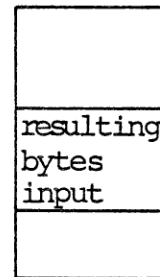
STACK BEFORE:



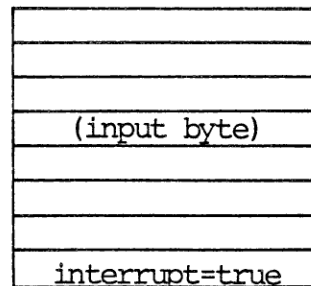
STACK AFTER:



MEMORY:



device (current level):



```
(* IORBB, single step *)
```

```
CONST
```

```
iorbb = #h8371;
```

```
VAR
```

```
next : addr;
```

```
count : integer;
```

```
top : word;
```

```
data: word;
```

```
PROCEDURE dumpregisters(level : integer);
```

```
EXTERNAL;
```

```
LABEL
```

```
iorblockfinis, ioblocktimeout;
```

```
BEGIN
```

```
IF ip.base <> iorbb THEN
```

```
  BEGIN (* first step *)
```

```
    IF curlevel = 0 THEN
```

```
      exception(block i/o level 0)
```

```
    ps.eoi := 0;
```

```
    dumpregisters(curlevel);
```

```
    (* change mode *)
```

```
    ib := iorbb; (* ip.base *)
```

```
    lm := stack(lp-7); (* top = last+1 *)
```

```
    ic := stack(lp-5); (* count *)
```

```
    sf := stack(lp-3); (* next_address.base *)
```

```
    lu := stack(lp-1)-1; (* next_address.disp-1 *)
```

```
    GOTO fetch1
```

```
  END
```

```
ELSE
```

```
  BEGIN
```

```
    WITH device(curlevel) DO
```

```
      BEGIN
```

```
        IF ps.to = 1 THEN GOTO ioblocktimeout;
```

```
        interrupt := 0; (* clear interrupt *)
```

```
        next.base := sf;
```

```
        next.disp := lu;
```

```
        count := ic;
```

```
        top := lm;
```

```
        function := read_data;
```

```
        data := data_in;
```

```
        IF eoi = 1 THEN
```

```
          BEGIN
```

```
            ps.eoi := 1;
```

```
            GOTO ioblockfinis
```

```
          END;
```

```
        membyte(next+1) := data AND 255;
```

```
        next := next+1;
```

```
        count := count-1;
```

```
        IF count > 0 THEN
```

```
          BEGIN
```

```
            lu := next.disp;
```

```
            ic := count;
```

```
            GOTO fetch
```

```
          END;
```

```

ioblockfinis:
    interrupt := 1;
ioblocktimeout:
    lm := mem(gf+dumplm);
    lu := mem(gf+dumplu)-6;
    sf := mem(gf+dumpsf);
    ib := mem(gf+dumpib);
    ic := mem(gf+dumpic)+1;
    stack(lp-1) := top-count
END (* WITH *)
END (* ELSE *)
END; (* IORBB, single step *)

```



10.16 Read Block of Bytes and Clear

10.16

This instruction resembles the Read Block of Bytes instruction except that the current interrupt level is cleared after the last read function is performed.

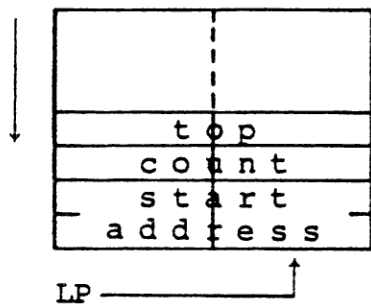
10.16.1 IORBBC

10.16.1

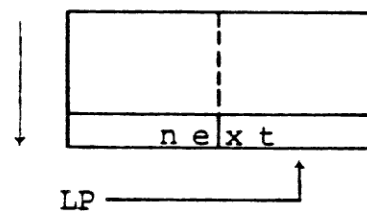
Input/Output Read Block of Bytes and Clear Value: 70<sub>Hex</sub>

IP → IORBBC

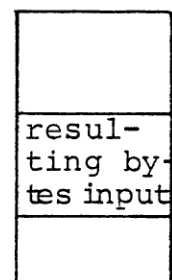
STACK BEFORE:



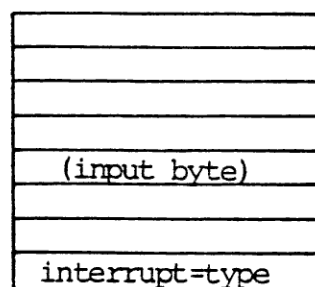
STACK AFTER:



MEMORY:



device (current level):



```
(* IORBBC, single step *)
```

```
CONST
```

```
iorbbc = #h8370;
```

```
VAR
```

```
next      : addr;
```

```
count     : integer;
```

```
top       : word;
```

```
data      : word;
```

```
PROCEDURE dumpregisters(level : integer);
```

```
EXTERNAL;
```

```
LABEL
```

```
  ioblockfinis;
```

```
BEGIN
```

```
  IF ip.base <> iorbbc THEN
```

```
    BEGIN (* first step *)
```

```
      IF curlevel = 0 THEN
```

```
        exception(block i/o level 0)
```

```
      ps.eoi := 0;
```

```
      dumpregisters(curlevel);
```

```
      (* change mode *)
```

```
      ib := iorbbc; (* ip.base *)
```

```
      lm := stack(lp-7); (* top = last+1 *)
```

```
      ic := stack(lp-5); (* count *)
```

```
      sf := stack(lp-3); (* next_address.base *)
```

```
      lu := stack(lp-1)-1; (* next_address.disp-1 *)
```

```
      GOTO fetch1
```

```
    END
```

```
  ELSE
```

```
    BEGIN
```

```
      WITH device(curlevel) DO
```

```
        BEGIN
```

```
          IF ps.to = 1 THEN GOTO ioblockfinis;
```

```
          interrupt := 0; (* clear interrupt *)
```

```
          next.base := sf;
```

```
          next.disp := lu;
```

```
          count := ic;
```

```
          top := lm;
```

```
          function := read_data;
```

```
          data := data in;
```

```
          IF eoi = 1 THEN
```

```
            BEGIN
```

```
              ps.eoi := 1; interrupt := 1; (* set interrupt *)
```

```
              GOTO ioblockfinis
```

```
            END;
```

```
          membyte(next+1) := data AND 255;
```

```
          next := next+1;
```

```
          count := count-1;
```

```
          IF count > 0 THEN
```

```
            BEGIN
```

```
              lu := next.disp;
```

```
              ic := count;
```

```
              GOTO fetch
```

```
            END;
```

```
      ioblockfinis:
        lm := mem(gf+dump1m);
        lu := mem(gf+dumplu)-6;
        sf := mem(gf+dumpsf);
        ib := mem(gf+dumpib);
        ic := mem(gf+dumpic)+1;
        stack(lp-1) := top-count
      END (* WITH *)
    END (* ELSE *)
  END; (* IORBEC, single step *)
```

10.17    Read Block of Words

10.17

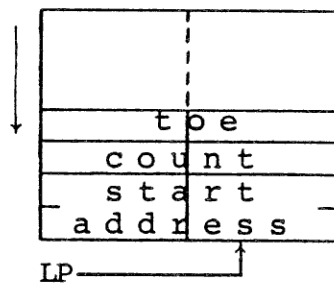
This instruction resembles the Read Block of Bytes instruction (section 10.15). A block of words is transferred, one word at a time, from the data\_in register of the device to the buffer.

10.17.1    IORBW

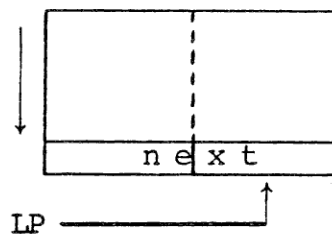
10.17.1

Input/Output Read Block of WordsValue: 75<sub>Hex</sub>IP → IOREW

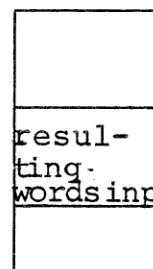
STACK BEFORE:



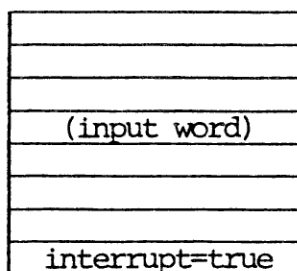
STACK AFTER:



MEMORY:



device (current level):



```
(* IORBW, single step *)
```

```
CONST
```

```
iorbw = #h8375;
```

```
VAR
```

```
next : addr;
```

```
count : integer;
```

```
top : word;
```

```
data: word;
```

```
PROCEDURE dumpregisters(level : integer);
```

```
EXTERNAL;
```

```
LABEL
```

```
iorblockfinis, iorblocktimeout;
```

```
BEGIN
```

```
IF ip.base <> iorbw THEN
```

```
  BEGIN (* first step *)
```

```
    IF curlevel = 0 THEN
```

```
      exception(block i/o level 0)
```

```
    ps.eoi := 0;
```

```
    dumpregisters(curlevel);
```

```
    (* change mode *)
```

```
    ib := iorbw; (* ip.base *)
```

```
    lm := stack(lp-7); (* top = last+1 *)
```

```
    ic := stack(lp-5); (* count *)
```

```
    sf := stack(lp-3); (* next_address.base *)
```

```
    lu := stack(lp-1)-1; (* next_address.disp-1 *)
```

```
    GOTO fetch1
```

```
  END
```

```
ELSE
```

```
  BEGIN
```

```
    WITH device(curlevel) DO
```

```
      BEGIN
```

```
        IF ps.to = 1 THEN GOTO iorblocktimeout;
```

```
        interrupt := 0; (* clear interrupt *)
```

```
        next.base := sf;
```

```
        next.disp := lu;
```

```
        count := ic;
```

```
        top := lm;
```

```
        IF count = 1 THEN
```

```
          GOTO iorblockfinis;
```

```
        function := read_data;
```

```
        data := data in;
```

```
        IF eoi = 1 THEN
```

```
          BEGIN
```

```
            ps.eoi := 1;
```

```
            GOTO iorblockfinis
```

```
          END;
```

```
        memword(next+1) := data;
```

```
        next := next+2;
```

```
        count := count-2;
```

```
        IF count > 0 THEN
```

```
          BEGIN
```

```
            lu := next.disp;
```

```
            ic := count;
```

```
            GOTO fetch
```

```
          END;
```

```

ioblockfinis:
    interrupt := 1;
ioblocktimeout:
    lm := mem(gf+dump1m);
    lu := mem(gf+dump1u)-6;
    sf := mem(gf+dumpsf);
    ib := mem(gf+dumpib);
    ic := mem(gf+dumpic)+1;
    stack(lp-1) := top-count
END (* WITH *)
END (* ELSE *)
END; (* IORBW, single step *)

```

10.18 Read Block of Words and Clear

10.18

This instruction resembles the Read Block of Bytes instruction except that the current interrupt level is cleared after the last read function is performed.

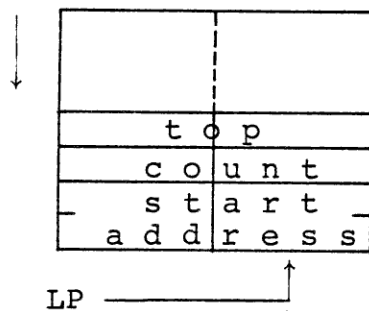
10.18.1 IORBWC

10.18.1

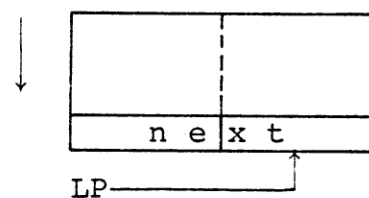
Input/Output Read Block of Words and Clear Value: 74<sub>Hex</sub>

IP → IORBWC

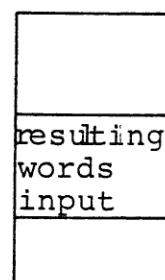
STACK BEFORE:



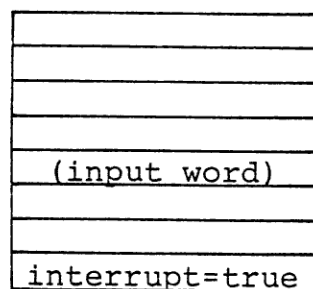
STACK AFTER:



MEMORY:



device (current level):



(\* IORBWC, single step \*)

```

CONST
    iorbwc = #h8374;
VAR
    next      : addr;
    count     : integer;
    top       : word;
    data      : word;
PROCEDURE dumpregisters(level : integer);
EXTERNAL;
LABEL
    ioblockfinis;
BEGIN
    IF ip.base <> iorbbc THEN
        BEGIN (* first step *)
            IF curlevel = 0 THEN
                exception(block i/o level 0)
            ps.eoi := 0;
            dumpregisters(curlevel);
            (* change mode *)
            ib := iorbwc; (* ip.base *)
            lm := stack(lp-7); (* top = last+1 *)
            ic := stack(lp-5); (* count *)
            sf := stack(lp-3); (* next_address.base *)
            lu := stack(lp-1)-1; (* next_address.disp-1 *)
            GOTO fetch1
        END
    ELSE
        BEGIN
            WITH device(curlevel) DO
                BEGIN
                    IF ps.to = 1 THEN GOTO ioblockfinis;
                    interrupt := 0; (* clear interrupt *)
                    next.base := sf;
                    next.disp := lu;
                    count := ic;
                    top := lm;
                    IF count = 1 THEN
                        GOTO ioblockfinis
                    function := read_data;
                    data := data in;
                    IF eoi = 1 THEN
                        BEGIN
                            ps.eoi := 1; interrupt := 1;
                            GOTO ioblockfinis
                        END;
                    membyte(next+1) := data;
                    next := next+2;
                    count := count-2;
                    IF count > 1 THEN
                        BEGIN
                            lu := next.disp;
                            ic := count;
                            GOTO fetch
                        END;
                END;
            END;
        END;
    END;

```



```
      ioblockfinis:
          lm := mem(gf+dump1m);
          lu := mem(gf+dump1u)-6;
          sf := mem(gf+dumpsf);
          ib := mem(gf+dumpib);
          ic := mem(gf+dumpic)+1;
          stack(lp-1) := top-count
      END (* WITH *)
  END (* ELSE *)
END; (* IOREWC, single step *)
```

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:

- register set (see subsection 3.6.1)
- incarnation stack (see subsection 3.7.3.1)
- chain structures (see subsection 3.7.1.2)
- scheduler type (see below)
- semaphores (see subsection 3.7.2.2)
- message headers (see subsection 3.7.2.1)

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.

#### Data Structures

TYPE

```

counter_type = RECORD
    high : word;
    low  : word;
END;

scheduler_type = RECORD
    active_queue0 : addr;
    n, (* highest active_queue index *)
    m, (* lowest active_queue index *)
    k, (* current active_queue index *)
    nxt : integer;
    dummy_counter : counter_type;
END;

wait_cause_type = PACKED RECORD
    unused : 0..63;
    intrpt,
    sem,
    tim,
    writecntrl : boolean;
    unused1 : 0..15;
END;

```

Common Global Variables

VAR

```

head_a, last_a, first_a,
head_b, last_b, next : addr; (* used in queue manipulation *)
sem_addr, sem, ref_addr, ref, msg : addr;
reg : register_set_type;
register_sets : ARRAY(0..123) OF register_set_type;
cur_level : 0..123;
wcause : wait_cause_type;
device : ... (* cf. chapter 10 *)

```

The term `memaddr(a)` is introduced to defined two consecutive words in memory pointed out by the address `a`, for example:

```

memaddr(a) := b; means
memword(a) := b.base; and memword(a+2) := b.disp;

```

Routines

```

PROCEDURE get_two_addresses;
BEGIN
  head_a := memaddr(lp-3);
  ref_addr := memaddr(lp-7)
END; (* get_two_addresses *)

```

```

PROCEDURE dump_registers(VAR reg : register_set_type);
VAR
  inc_desc_ptr : addr;
BEGIN
  IF reg.ib <> -1 THEN
    BEGIN
      inc_desc_ptr.base := reg.pb;
      inc_desc_ptr.disp := reg.pr;
      memword(inc_desc_ptr+dumpib) := reg.ib; (* entry_point base *)
      memword(inc_desc_ptr+dumpic) := reg.ic; (* entry_point disp *)
      memword(inc_desc_ptr+dumpsf) := reg.sf;
      memword(inc_desc_ptr+dumplu) := reg.lu;
      memword(inc_desc_ptr+dumplm) := reg.lm;
      memword(inc_desc_ptr+dumpps) := reg.ps;
      reg.ib := -1
    END
  END; (* dump_registers *)

```

```

PROCEDURE load_registers;
BEGIN (* entry requirement : next = incarnation descriptor *)
  dump_registers(reg);
  reg.lm := memword(next+dump1m);
  reg.ps := memword(next+dumpps);
  reg.lu := memword(next+dumplu);
  reg.sf := memword(next+dumpsf);
  reg.ib := memword(next+dumpib); (* entry_point base *)
  reg.ic := memword(next+dumpic) (* entry_point disp *)

END; (* load_registers *)

```

```

PROCEDURE get_message;
BEGIN
  IF ref_addr.base.nilbit = 1 THEN
    exception(pointer = nil);
  ref := memaddr(ref_addr);
  IF ref.base.lockbit = 1 THEN exception(reference locked)
END; (* get_message *)

```

```

PROCEDURE reserve_a;
BEGIN
  IF head_a.base.nilbit = 1 THEN exception(pointer = nil);
  last_a := memaddr(head_a)
END; (* reserve_a *)

```

```

PROCEDURE reserve_b;
BEGIN
  IF head_b.base.nilbit = 1 THEN exception(pointer = nil);
  last_b := memaddr(head_b)
END; (* reserve_b *)

```

```

PROCEDURE unchain;
BEGIN (* unlink next from last_a with head head a *)
  IF last_a = next THEN (* chain goes empty *)
    memaddr(head_a).base.nilbit := 1
  ELSE
    BEGIN (* queue remains non-empty *)
      first_a := memaddr(next);
      memaddr(last_a) := first_a
    END
  END; (* unchain *)

```

```

PROCEDURE chain_first(start_monitor : boolean);
BEGIN (* put next into the chain of head b *)
  IF last_b.base.nilbit = 1 THEN chain(start_monitor) (* see below *)
  ELSE
    BEGIN
      memaddr(next) := memaddr(last_b);
      memaddr(last_b) := next
    END
  END; (* chain_first *)

```

```

PROCEDURE put_elem_ptr;
BEGIN
    memaddr(ref_addr) := next
END; (* put_elem_ptr *)

```

```

PROCEDURE change_a_b;
BEGIN
    head_b := head_a;
    last_b := last_a
END; (* change_a_b *)

```

```

PROCEDURE set_ps_wait(wcause : wait_cause_type);
BEGIN
    reg := register_sets(level);
    reg.ps.wait := reg.ps.wait OR wcause
END; (* set_ps_wait *)

```

```

PROCEDURE send_control;
VAR
    oldlp : word;
BEGIN
    oldlp := lp;
    IF wcause.sem THEN lp := lp - 8;
    device(cur_level).function := write_control;
    device(cur_level).control := memword(lp-1);
    lp := oldlp
END; (* send_control *)

```

```

PROCEDURE set_ref_message_waited;
VAR
    inc, ref_addr : addr;
BEGIN
    inc := memaddr(next);
    ref_addr := memaddr(inc+msg_waited);
    memaddr(ref_addr) := ref
END; (* set_ref_message_waited *)

```

```

PROCEDURE set_chain_head;
VAR
    inc : addr;
BEGIN
    inc := memaddr(next);
    memaddr(inc+chainhead) := head_b
END; (* set_chain_head *)

```

```

PROCEDURE chain(start_monitor : boolean);
(* put an incarnation/message (next) in an active/semaphore queue
   (head b) as the last element; schedule if start_monitor and
   the queue was empty *)
BEGIN
  IF memaddr(head_b).base.nilbit = 0 THEN
    BEGIN (* non empty queue *)
      memaddr(next) := memaddr(last_b);
      memaddr(last_b) := next
    END
  ELSE
    BEGIN (* queue empty *)
      memaddr(next) := next;
      IF start_monitor THEN
        set_interrupt(monitor_level); (* see below *)
        memaddr(head_b) := next
      END
    END
  END; (* chain *)

```

PROCEDURE signal;

VAR

inc : addr;

BEGIN (\* defined variables at entry:

head a is the address of the semaphore,

ref is the address of the message,

ref addr is the address of ref \*)

reserve\_a; (\* last\_a := memaddr(head a), i.e. the last  
element of the chain \*)

IF (last\_a.base.nilbit=0) AND

(last\_a.base.bit15=1) THEN

BEGIN (\* semaphore locked \*)

next := memaddr(last\_a); (\* first incarnation in the  
chain \*)

(\* now next is the address of an incarnation descriptor \*)

inc := memaddr(next);

(\* now inc is a global frame pointer \*)

IF membyte(inc+level) = 0 THEN

BEGIN

head b := memaddr(inc+activequeue);

reserve b;

set\_ref\_message\_waited; (\* inc msg\_waited) := msg \*)

set\_chain head;

unchain; (\* unlink incarnation from semaphore \*)

chain(true)

END

ELSE

BEGIN (\* level of receiving incarnation > 0 \*)

set\_ref\_message\_waited;

set\_interrupt(level); (\* see below \*)

memaddr(inc+chainhead).base.nilbit := 1;

unchain

END (\* level > 0 \*)

END (\* semaphore locked \*)

ELSE

BEGIN (\* semaphore open or passive \*)

change\_a\_b;

chain(false) (\* chain the message to the queue of the  
semaphore \*)

END;

memaddr(ref\_addr).base.nilbit := 1

END; (\* signal \*)

PROCEDURE set\_interrupt(level : integer);

BEGIN

set\_external(level);

IF level <= 7 THEN

set\_internal(level)

END; (\* set\_interrupt \*)

```

PROCEDURE clear_interrupt(level : integer);
BEGIN
    clear_external(level);
    IF level <= 7 THEN
        clear_internal(level)
    END; (* clear_interrupt *)

PROCEDURE wait(multiple : boolean; VAR effwait : boolean);
BEGIN (* defined variables at entry:
        head_a is the address of the semaphore,
        ref_addr is the address of the ref to receive a mes-
        sage *)
    get_message;
    IF ref.base.nilbit = 0 THEN
        exception(wait : reference <> nil);
    reserve a;
    IF (last_a.base.nilbit=0) AND
        (last_a.base.bit15=0) THEN
        BEGIN (* semaphore open *)
            next := memaddr(last_a);
            unchain;
            effwait := false;
            put_elem_ptr
        END;
    ELSE
        BEGIN (* semaphore passive or locked *)
            IF multiple AND (last_a.base.bit15=1) THEN
                exception(multiple wait on locked semaphore);
            next := gf; (* current process *)
            change_a_b;
            memaddr(gf+msg_waited) := ref_addr;
            IF cur_level < 0 THEN
                clear_interrupt
            ELSE
                BEGIN
                    head_a := memaddr(gf+chainhead);
                    deschedule10 (* see below *)
                END;
            effwait := true;
            set_chain head;
            chain(false)
        END (* semaphore passive or locked *)
    END; (* wait *)

```



```

PROCEDURE deschedule0;
BEGIN (* entry requirements: head_a = active queue,
                                next = process *)
    reserve_a;
    unchain;
    set_interrupt(monitor_level)
END; (* deschedule0 *)

```

```

FUNCTION test_interrupt : boolean;
BEGIN
    test_interrupt := cur_level < next_level
END; (* test_interrupt *)

```

```

PROCEDURE deschedule;
BEGIN
    IF cur_level < 0 THEN
        clear_interrupt(cur_level)
    ELSE
        BEGIN
            next := gf;
            head_a := memaddr(gf+chainhead);
            memaddr(gf+chainhead).base.nilbit := 1
        END
    END; (* deschedule *)

```

## 11.1 Signal

11.1

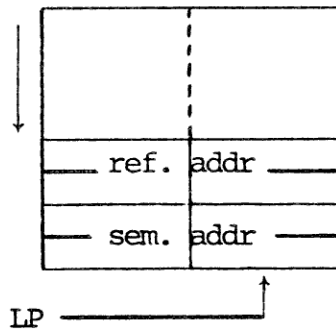
The instructions in this group signal a message to a semaphore. If the semaphore is closed, a process incarnation is moved from the semaphore to an active queue with a reference to the message. If the semaphore is open or passive, the message is linked as the last element in the semaphore queue.

11.1.1 CSIGN

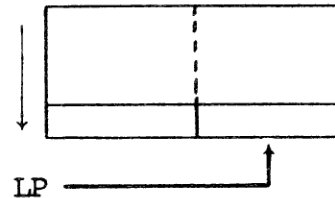
11.1.1

Control SIGNValue: 11<sub>Hex</sub>IP → CSIGN

STACK BEFORE:



STACK AFTER:



The state of the semaphore determines the chaining and unchaining performed. These data structure modifications are not shown.

(\* CSIGN \*)

```

BEGIN
  get_two_addresses;
  get_message;
  IF ref.base.nilbit = 1 THEN exception(pointer = nil);
  signal;
  lp := lp - 8;
  ip := ip + 1
END; (* CSIGN *)

```

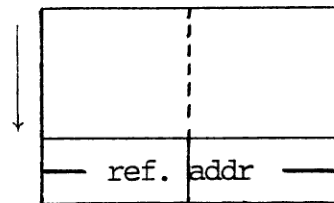
## 11.1.2 CRELE

11.1.2

Control RELEase MessageValue: 12<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| CRELE | parameter |
|-------|-----------|

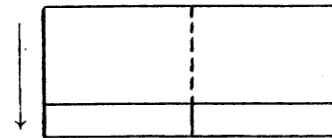
STACK BEFORE:



LP →

(\* CRELE \*)

STACK AFTER:



LP →

VAR

semrel : word;

BEGIN

semrel := membyte(ip+1);

ref\_addr := memaddr(lp-3);

get\_message;

head\_a.disp := ref.disp + semrel;

head\_a.base := ref.base;

IF rel.base.nilbit = 1 THEN exception(pointer = nil);

reserve\_a;

signal;

lp := lp - 4;

ip := ip + 2

END; (\* CRELE \*)

11.2 Wait

11.2

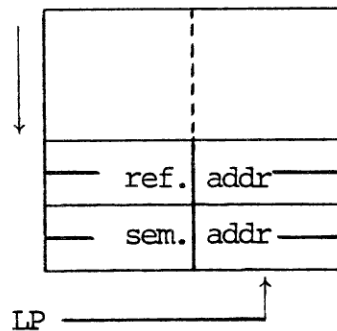
The instructions in this group wait for a message from a semaphore. Depending on which of the instructions is used, the process incarnation executing the instruction will/will not be removed from the active queue if the semaphore is closed or passive.

11.2.1 CSENS

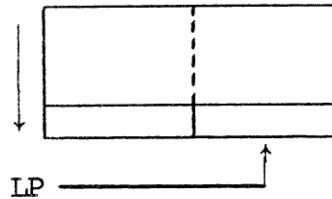
11.2.1

Control SENSE SemaphoreValue: 15<sub>Hex</sub>IP → CSENS

STACK BEFORE:



STACK AFTER:



(\* CSENS \*)

```

BEGIN
  get_two_addresses;
  get_message;
  IF ref.base.nilbit = 1 THEN exception(reference <> nil);
  reserve_a; (* last_a := sem *)
  IF (last_a.base.nilbit=1) OR (last_a.base.bit15=1) THEN
    BEGIN (* not open *)
      next := nil;
      put_elem_ptr
    END
  ELSE
    BEGIN
      next := memaddr(last_a);
      unchain; (* unlink message from semaphore *)
      put_elem_ptr
    END;
  lp := lp - 8;
  ip := ip + 1
END; (* CSENS *)

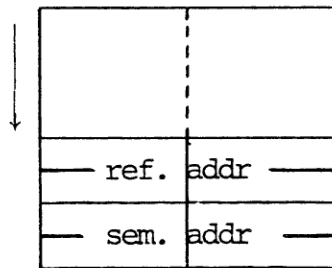
```

11.2.2 CWAIT

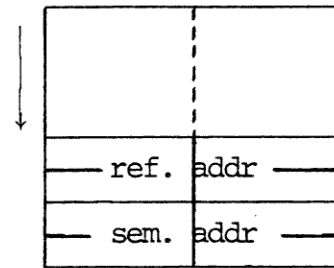
11.2.2

Control WAITValue: 10<sub>Hex</sub>IP → CWAIT

STACK BEFORE:



STACK AFTER:



(\* CWAIT \*)

```

VAR
  dummy : boolean;
BEGIN
  get_two_addresses;
  wait(false, dummy);
  ip := ip + 1
END;  (* CWAIT *)

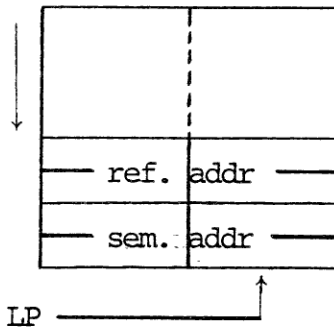
```

## 11.2.3 CWTAC

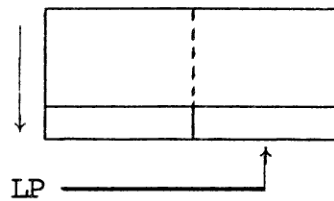
11.2.3

Control Wait And ClearValue: 16<sub>Hex</sub>IP → CWTAC

STACK BEFORE:



STACK AFTER: unchanged or



(\* CWTAC \*)

```

BEGIN
  get_two_addresses;
  ref := memaddr(ref_addr);
  IF ref.base.nilbit = 1 THEN
    clear_interrupt(cur_level)
  ELSE
    BEGIN
      lp := lp - 8;
      ip := ip + 1
    END
  END;
  (* CWTAC *)

```

11.3 Link/Unlink

11.3

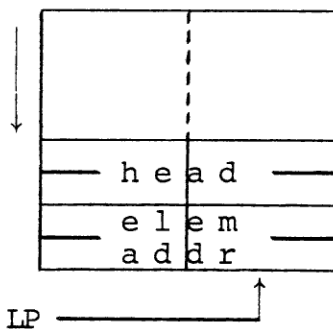
The instructions of this group manipulate elements of a chained structure (see section 3.7).

11.3.1 CLLST

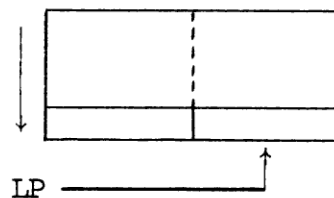
11.3.1

Control Link LaSTValue: 13<sub>Hex</sub>IP → CLLST

STACK BEFORE:



STACK AFTER:



The structure pointed out by elemaddr is linked as the last element in the chain pointed out by head.

(\* CLLST \*)

BEGIN

```

  get_two_addresses;  (* head_a := head; ref_addr := elem_addr *)
  head_b := head_a;
  reserve_b;
  get_message;
  next := ref;
  chain(false);
  lp := lp - 8;
  ip := ip + 1
END;  (* CLLST *)

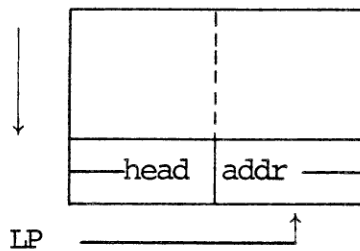
```

## 11.3.2 CSKIP

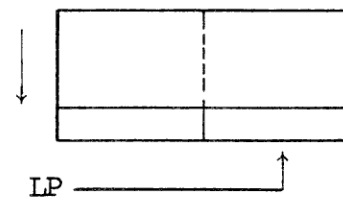
11.3.2

Control SKIP to NextValue: 14<sub>Hex</sub>IP → CSKIP

STACK BEFORE:



STACK AFTER:



(\* CSKIP \*)

```

VAR
  head_addr, link : addr;
BEGIN
  head_addr := memaddr(lp-3);
  IF head_addr.base.nilbit = 1 THEN
    exception(nil exception);
  link := memaddr(head_addr);
  IF link.base.nilbit = 0 THEN
    memaddr(head_addr) := memaddr(link)
  END; (* CSKIP *)

```



11.4      Select Level

11.4

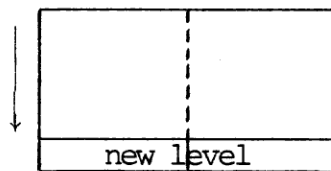
This instruction is used to select a new level with corresponding register set as the current level for a process incarnation.

11.4.1      CSELL

11.4.1

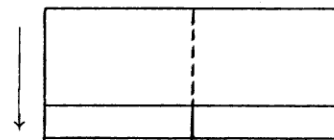
Control SElect LevelValue: 1A<sub>Hex</sub>IP → CSELL

STACK BEFORE:



LP —————↑

STACK AFTER:



LP —————↑

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.

```
(* CSELL *)
```

```
VAR
  new_level : integer;
BEGIN
  new_level := stack(lp-1);
  next := gf; (* global frame *)
  head_a := memaddr(next+activequeue);
  reserve a;
  clear_interrupt(cur_level);
  IF new_level = 0 THEN
    BEGIN
      IF level < 0 THEN
        BEGIN (* put current process into active queue *)
          change_a_b;
          set_chain_head;
          chain_first(false)
        END
      END
    ELSE
      BEGIN (* new_level > 0 *)
        set_interrupt(new_level);
        IF level = 0 THEN
          BEGIN (* unlink own from active queue *)
            head_b.base.nilbit := 1;
            set_chain_head;
            unchain;
            set_interrupt(monitor_level)
          END
        END; (* new_level > 0 *)
        (* common select level action *)
        membyte(next+level) := new_level;
        dump_registers(reg);
        load_registers;
        reg := register_sets(new_level*8+7);
        reg.ps.to := 0;
        level := new_level;
        lp := lp - 2;
        ip := ip + 1
      END; (* CSELL *)
```

11.5 Stop

11.5

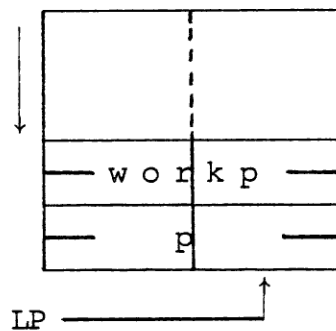
This instruction stops a process incarnation by removing it from the chained structures. The contents of the register set will have been dumped in the process incarnation descriptor after execution. The instruction must be executed from level 0.

11.5.1 CSTOP

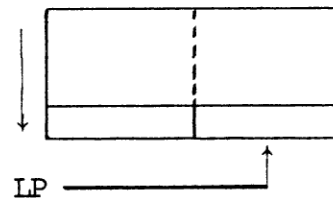
11.5.1

Control STOPValue: 1B<sub>Hex</sub>IP → CSTOP

STACK BEFORE:



STACK AFTER:



The instruction may be executed only on level 0.

(\* CSTOP \*)

```

VAR
  reg : 0..123;
  wpchain_head, p, workp, pred, next : addr;
BEGIN
  get_two_addresses;
  workp := head_a;
  p := ref addr;
  head_b := memaddr(ref_addr+chain_head);
  IF head_b.base.nilbit = 0 THEN
    BEGIN (* unlink p from its chain *)
      reserve_b;
      IF workp.base.nilbit = 0 THEN
        BEGIN (* continue after interrupt *)
          wpchain_head := memaddr(workp + chain_head);
          IF wpchain_head <> head_b THEN head_a := last_b
        END
      ELSE
        head_a := last_b; (* first loop *)
      rep: pred := head_a;
        head_a := memaddr(pred);
        IF head_a = p THEN GOTO found;
        IF test_interrupt THEN
          BEGIN
            memaddr(lp-7) := head_a; (* rewrite workp *)
            instruction_fetch
          END;
          GOTO rep;
        found: (* p = head_a, pred = preceding process,
          head b = head of chain, last b = last of chain *)
          IF p = pred THEN memaddr(head_b).base.nilbit := 1
        ELSE
          BEGIN
            next := memaddr(head_a);
            memaddr(pred) := next;
            IF last_b = head_a THEN memaddr(head_b) := pred
          END
        END; (* unlink p from its chain *)
        reg := membyte(p+level);
        IF reg <> 0 THEN dump_registers(register_sets(reg));
        ps := memword(p+dumpp);
        memword(p+dumpp) := ps AND (NOT ps.wait); (* clear wait bit *)
        lp := lp - 8;
        ip := ip + 1
      END; (* CSTOP *)

```

11.6 Multiple Wait

11.6

The instructions in this group form together a multiple wait on combinations of three different events: a message from a semaphore, an interrupt, and a timeout. This results in seven different wait combinations.

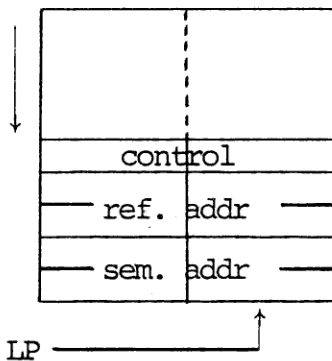
11.6.1 MCIS

11.6.1

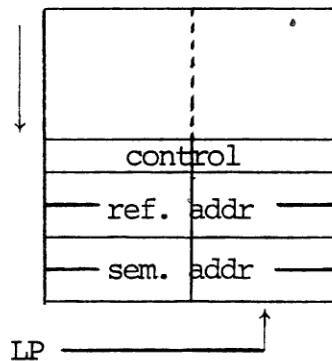
Multiple Control Interrupt and Semaphore Value: 58<sub>Hex</sub>

IP → MCIS

STACK BEFORE:



STACK AFTER:



(\* MCIS \*)

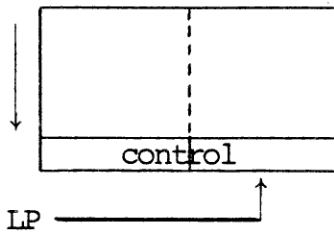
```

VAR
  oldlp : addr;
  dummy_result : boolean;
BEGIN
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := true;
      tim := false;
      writectrl := true
    END;
    set_ps_wait(wcause);
    get_two_addresses;
    wait(true, dummy_result);
    send_control;
    ip := ip + 1
  END; (* MCIS *)

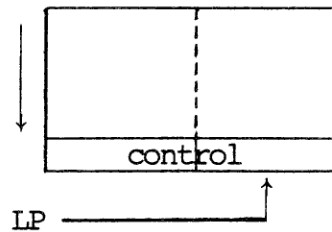
```

Multiple Control Interrupt and TimerValue: 68<sub>Hex</sub>IP → MCIT

STACK BEFORE:



STACK AFTER:



(\* MCIT \*)

```

BEGIN
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := false;
      tim := true;
      writecntrl := true
    END;
    set_ps_wait(wcause);
    IF memword(gf+timer) > 0 THEN deschedule;
    send control;
    ip := ip + 1
  END; (* MCIT *)

```

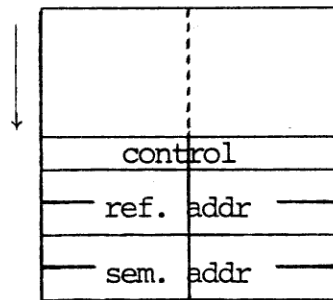
## 11.6.3 MCIST

11.6.3

Multiple Control Interrupt, Semaphore, and Timer Value: 78<sub>Hex</sub>

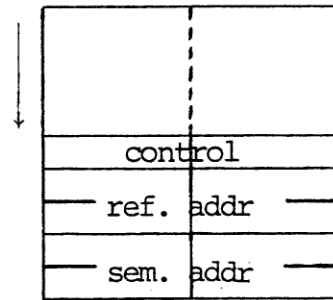
IP → MCIST

STACK BEFORE:



LP

STACK AFTER:



LP

(\* MCIST \*)

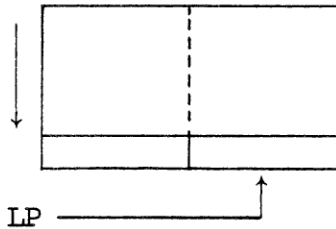
```

VAR
  oldlp : addr;
  dummy_result : boolean;
BEGIN
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := true;
      tim := true;
      writecntrl := true
    END;
    set_ps_wait(wcause);
    IF memword(gf+timer) > 0 THEN
      BEGIN
        get_two_addresses;
        wait(true, dummy_result)
      END;
    send_control;
    ip := ip + 1
  END; (* MCIST *)

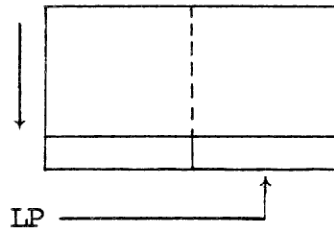
```

Multiple Wait InterruptValue: 88<sub>Hex</sub>IP → MWI

STACK BEFORE:



STACK AFTER:



The stack is not involved.

(\* MWI \*)

```

BEGIN (* suicide if executed on level 0 *)
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := false;
      tim := false;
      writecntrl := false;
    END;
    set_ps_wait(wcause);
    deschedule;
    ip := ip + 1
  END; (* MWI *)

```

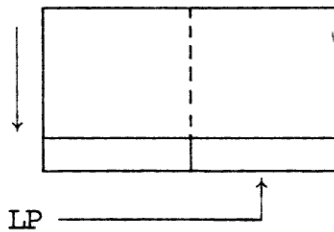


## 11.6.5 MWT

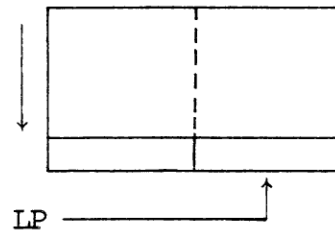
11.6.5

Multiple Wait TimerValue: 20<sub>Hex</sub>IP → MWT

STACK BEFORE:



STACK AFTER:



The stack is not involved.

(\* MWT \*)

```

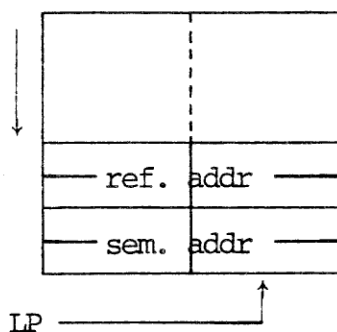
BEGIN
  WITH wcause DO
    BEGIN
      intrpt := false;
      sem := false;
      tim := true;
      writecntrl := false
    END;
    set_ps_wait(wcause);
    IF memword(gf+timer) > 0 THEN deschedule;
    ip := ip + 1
  END;  (* MWT *)

```

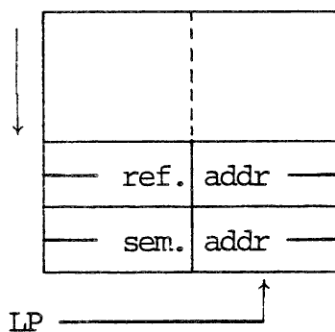
Multiple Wait Interrupt and Semaphore Value: 18<sub>Hex</sub>

IP → MWIS

STACK BEFORE:



STACK AFTER:



(\* MWIS \*)

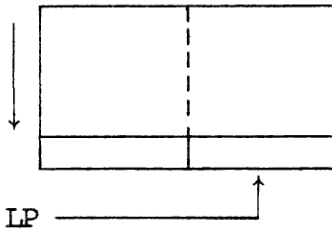
```

VAR
  oldlp : addr;
  dummy_result : boolean;
BEGIN
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := true;
      tim := false;
      writecntrl := false
    END;
    set_ps_wait(wcause);
    get_two_addresses;
    wait(true, dummy_result);
    ip := ip + 1
  END; (* MWIS *)

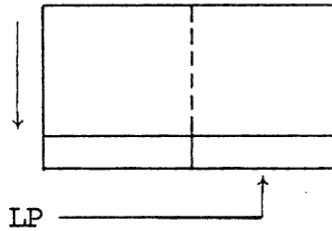
```

Multiple Wait Interrupt and TimerValue: 28<sub>Hex</sub>IP → MWIT

STACK BEFORE:



STACK AFTER:



The stack is not involved.

(\* MWIT \*)

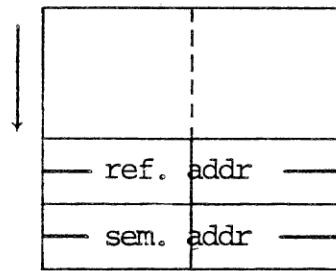
```

BEGIN
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := false;
      tim := true;
      writecntrl := false
    END;
    set_ps wait(wcause);
    IF mem(gf+timer) > 0 THEN deschedule;
    ip := ip + 1
  END;  (* MWIT *)

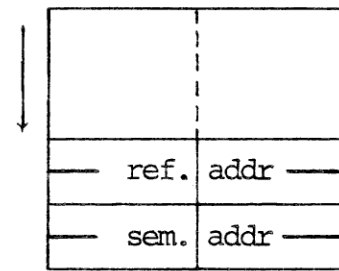
```

Multiple Wait Semaphore and TimerValue: 30<sub>Hex</sub>IP → MWST

STACK BEFORE:



STACK AFTER:



(\* MWST \*)

```

VAR
  dummy_result : boolean;
BEGIN
  WITH wcause DO
    BEGIN
      intrpt := false;
      sem := true;
      tim := true;
      writecntrl := false
    END;
    set_ps wait(wcause);
    IF mem(gf+timer) > 0 THEN
      BEGIN
        get_two_addresses;
        wait(true, dummy_result)
      END;
      ip := ip + 1
    END; (* MWST *)
  
```

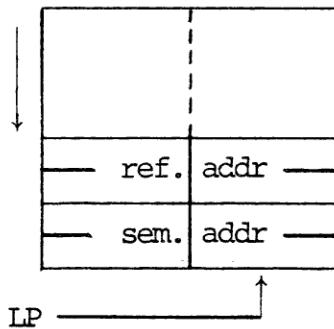
## 11.6.9 MWIST

11.6.9

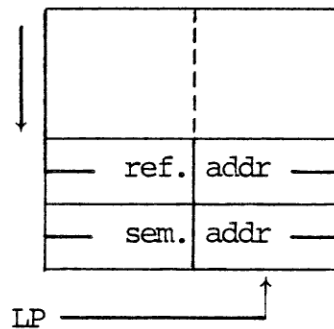
Multiple Wait Interrupt, Semaphore, and Timer Value: 38<sub>Hex</sub>

IP → MWIST

STACK BEFORE:



STACK AFTER:



(\* MWIST \*)

```

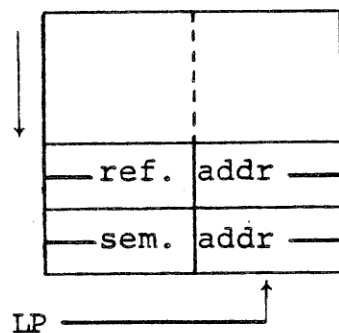
VAR
  dummy_result : boolean;
BEGIN
  WITH wcause DO
    BEGIN
      intrpt := true;
      sem := true;
      tim := true;
      writecntrl := false
    END;
    set_ps wait(wcause);
    IF mem(gf+timer) > 0 THEN
      BEGIN
        get_two_addresses;
        wait(true, dummy_result);
      END;
      ip := ip + 1
    END; (* MWIST *)
  
```

Multiple Wait And Clear

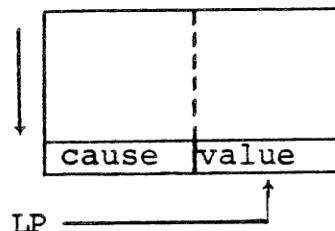
Value: 17<sub>Hex</sub>

IP → MWTAC

STACK BEFORE:

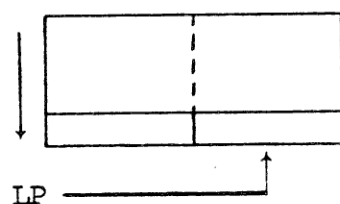


STACK AFTER:

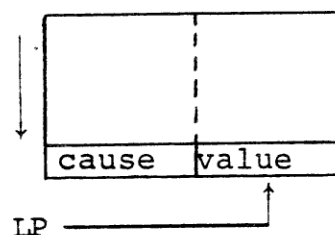


or:

STACK BEFORE:



STACK AFTER:



The stack before execution is shown in two situations, either containing semaphore and reference addresses or being empty. Furthermore the stack may contain a control word in both situations (placed at the lowest stack address). In all situations the stack is emptied (0, 2, 8 or 10 bytes are removed), and the cause value is placed as the result.

The cause value may be one of the following: 0 for interrupt; 1 for message; or 2 for timer.

(\* MWTAC \*)

VAR

causeval : (interrupt, message, timer);  
t : integer;

BEGIN

IF register\_sets(cur\_level).ps.wait.sem = false THEN GOTO testtimer;  
ref\_addr := memaddr(lp-7);  
ref\_base := mem(ref\_addr);  
IF ref\_base.nilbit = 1 THEN GOTO testtimer;  
causeval := message;

pfetch: IF register\_sets(cur\_level).ps.wait.sem THEN lp := lp - 8;

IF register\_sets(cur\_level).ps.wait.wrtcntrl THEN lp := lp - 2;

WITH wcause DO

BEGIN

intrp := false;  
sem := false;  
tim := false;  
writecntrl := false

END;

set\_ps\_wait(wcause);  
mem(lp+1) := causeval;

lp := lp + 2;

ip := ip + 1;

GOTO end\_mwtac;

testtimer: (\* if timeout, p is unlinked from semaphore  
by MTIME instruction \*)

IF register\_sets(cur\_level).ps.wait.tim = false THEN GOTO inttest;

t := mem(gf+timer);

IF t > 0 THEN GOTO inttest;

causeval := timer;

GOTO pfetch;

inttest: IF register\_sets(cur\_level).ps.wait.intrpt = false THEN

GOTO repeatinst;

IF registers(cur\_level).ps.wait.sem THEN

BEGIN (\* unlink current process from semaphore \*)

next := gf;

get\_two\_addresses;

reserve\_a;

unchain;

head\_b.base.nilbit := 1;

set\_chain\_head

END; (\* unlink own from semaphore \*)

causeval := interrupt;

GOTO pfetch;

repeatinst: deschedule; (\* repeat instruction on next activation \*)

end\_mwtac:

END; (\* MWTAC \*)

11.7 Time

11.7

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

11.7.1 MTIME

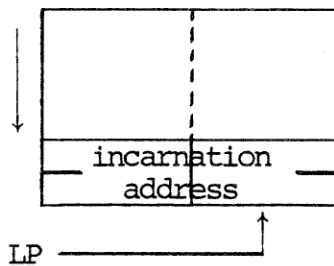
11.7.1

Monitor function TIME count down

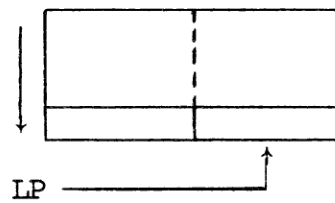
Value: 19<sub>Hex</sub>

IP → MTIME

STACK BEFORE:



STACK AFTER:



This instruction must be executed on level 0!



(\* MTIME \*)

VAR

p : addr;  
t, l : integer;  
ps : word;

BEGIN

p := memaddr(lp-3);

rep:

IF p.base.nilbit = 1 THEN

BEGIN

lp := lp - 4;

ip := ip + 1;

GOTO end\_mtime

END;

t := mem(p+timer);

IF t <= 0 THEN GOTO nextp;

mem(p+timer) := t - 1;

IF t - 1 <> 0 THEN GOTO nextp;

(\* time out \*)

l := membyte(p+level);

IF l <> 0 THEN

BEGIN (\* driver timeout \*)

IF register\_sets(l).ib = - 1 THEN GOTO already\_scheduled;  
(\* p is stopped \*)

ps := register\_sets(l).ps;

register\_sets(l).ps := ps OR to; (\* time out \*)

set\_interrupt(l)

END

ELSE

ps := mem(p+dumps);

IF ps.wait.tim = false THEN GOTO already\_scheduled;

head\_a := memaddr(p+chainhead);

next := p;

IF (ps.wait.sem) AND (head\_a.base.nilbit=0) AND

(head\_a.base.bit15=0) THEN

BEGIN (\* unlink process from semaphore \*)

reserve a;

unchain;

memaddr(p+chainhead).base.nilbit := 1;

head\_a.base.nilbit := 1

END;

(\* schedule p \*)

IF l = 0 THEN (\* level = 0 \*)

BEGIN (\* link p to active queue if and only if not already  
linked \*)

IF (head\_a.base.nilbit=0) AND (head\_a.base.bit15=1) THEN

GOTO already\_scheduled;

head\_b := memaddr(p+active\_queue);

reserve b;

set\_chain\_head;

chain(true)

END; (\* else interrupt already set \*)

already\_scheduled:

nextp:

p := memaddr(p+tchain);

IF NOT test\_interrupt THEN GOTO rep;

stack(lp-3) := paddr.base;

stack(lp-1) := paddr.disp;

end\_mtime:

END; (\* MTIME \*)

11.8 Schedule Next Incarnation to Run

11.8

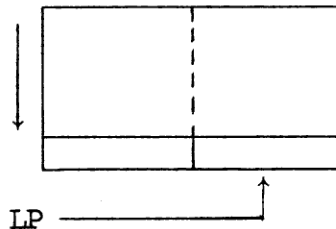
This instruction prepares the scheduling of the next process incarnation to run. The actual selection of the incarnation is integrated in the instruction fetch.

11.8.1 SCHED

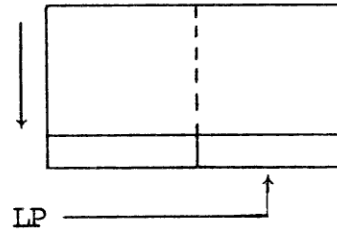
11.8.1

SCHEDULEValue: 1D<sub>Hex</sub>IP → SCHED

STACK BEFORE:



STACK AFTER:



The stack is not involved.

(\* SCHED \*)

```

VAR
  scheduler_data : scheduler_type;
BEGIN
  scheduler_data.nxt := scheduler_data.n;
  dump_registers(register_sets(0));
  clear_interrupt(monitor_level);
  ip := ip + 1
END; (* SCHED *)

```

11.9 Driver Activation

11.9

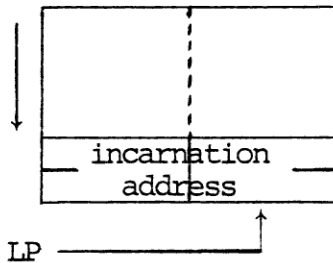
The instructions are used to start a driver on its associated interrupt level, and to choose a new current level.

11.9.1 CSTDR

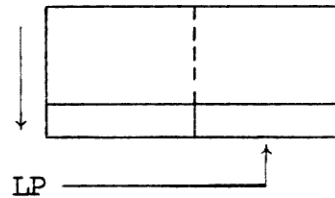
11.8.1

Control Start DriverValue: 1C<sub>Hex</sub>IP → CSTDR

STACK BEFORE:



STACK AFTER:



(\* CSTDR \*)

```

VAR
  l : integer;
BEGIN
  next := memaddr(lp-3);
  load_registers;
  l := membyte(next+level);
  set_interrupt(l);
  lp := lp - 4;
  ip := ip + 1
END;  (* CSTDR *)

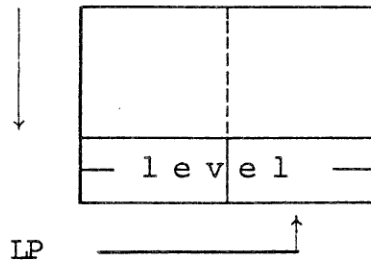
```

## 11.9.2 CSLEV

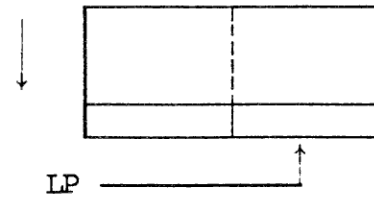
11.9.2

Control Set LevelValue: 1E<sub>Hex</sub>IP → CSLEV

STACK BEFORE:



STACK AFTER:



(\* CSLEV \*)

```

VAR
  level : integer;
BEGIN
  level := memword(lp-1);
  cur_level := level;
  set_interrupt(level);
  lp := lp - 2;
  ip := ip + 1
END;  (* CSLEV *)

```

## 11.10 Register Array Operations

11.10

The instructions in this group are used to manipulate the register array (section 3.6) and the RAM memory of the control microprocessor (section 3.3).

### 11.10.1 CRGET

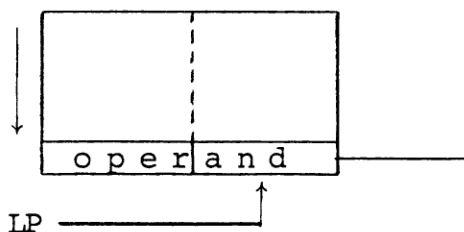
11.10.1

Control Register GET

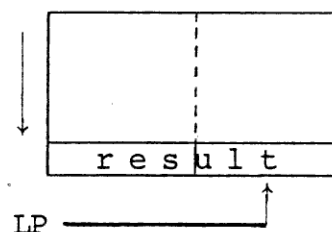
Value: 91<sub>Hex</sub>

IP → CRGET

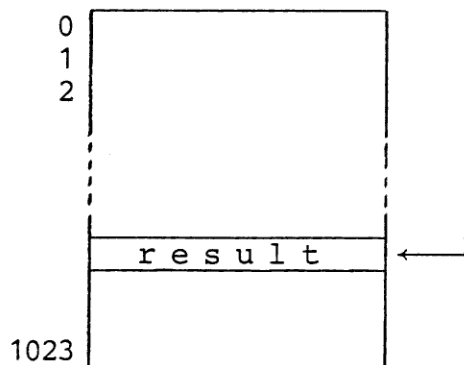
STACK BEFORE:



STACK AFTER:



REGISTER ARRAY:

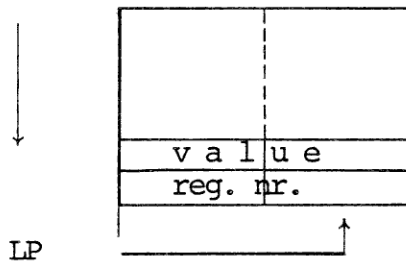


(\* CRGET \*)

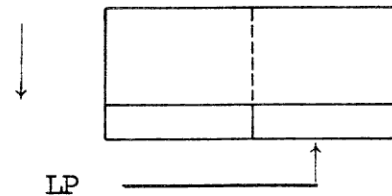
```
VAR
  registers : ARRAY(0..1023) OF word;
BEGIN
  stack(lp-1) := registers(stack(lp-1));
  ip := ip + 1
END; (* CRGET *)
```

Control Register PUTValue: B<sub>Hex</sub>IP → CRPUT

STACK BEFORE:

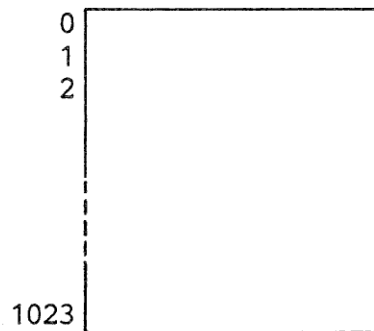


STACK AFTER:

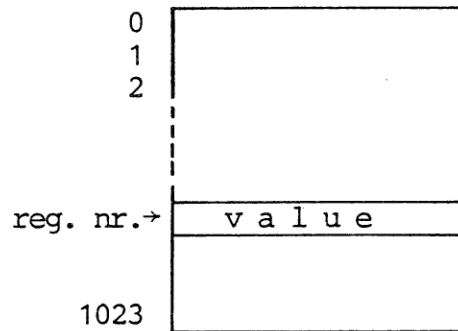


or:

REGISTER ARRAY BEFORE:



REGISTER ARRAY AFTER:



The stack is not involved.

(\* CRPUT \*)

```

VAR
  registers : ARRAY(0..1023) OF word;
BEGIN
  registers(memword(lp-1)) := memword(lp-3);
  lp := lp - 4;
  ip := ip + 1
END;  (* CRPUT *)

```

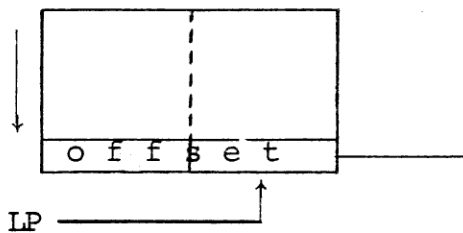
## 11.10.3 CRRAM

11.10.3

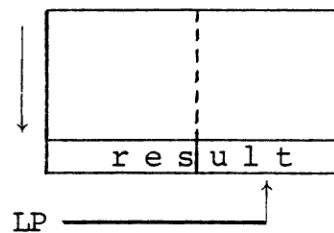
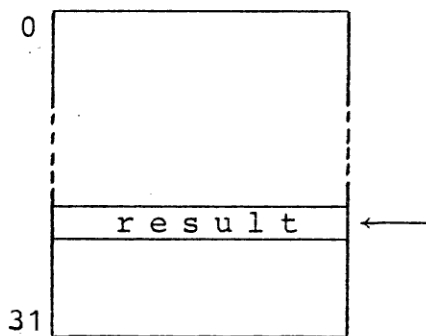
Control Read RAM of Control ProcessorValue: D0<sub>Hex</sub>

IP → CRRAM

STACK BEFORE:



STACK AFTER:

CONTROL  
PROCESSOR RAM COPY:

(\* CRRAM \*)

VAR

```
control_byte_addr : addr;
control_processor_ram_copy : ARRAY(0..31) OF byte; (* placed as
first part
of the first
ram module *)
```

BEGIN

```
control_byte_addr.base := #hC0;
control_byte_addr.disp := stack(lp - 1);
stack(lp - 1) := word(membyte(control_byte_addr));
ip := ip + 1
END; (* CRRAM *)
```

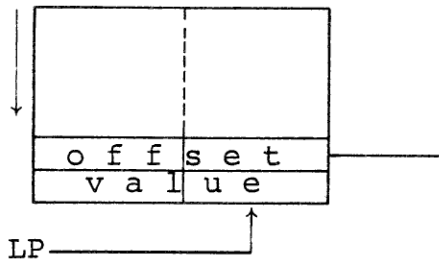
## 11.10.4 CWRAM

11.10.4

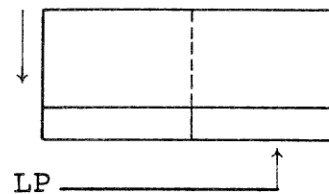
Control Write RAM of Control Processor      Value: F0<sub>Hex</sub>

IP → CWRAM

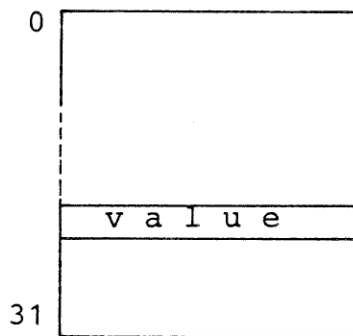
STACK BEFORE:



STACK AFTER:



CONTROL  
PROCESSOR RAM COPY:



(\* CWRAM \*)

```

VAR
  control_byte_addr : addr;
  value : byte;
BEGIN
  control_byte_addr.base := #hC0;
  value := stack(lp-1);
  control_byte_addr.disp := stack(lp - 3);
  IF register_sets(com8085).cow = 0 THEN
    (* 8085 debugger not busy *)
    BEGIN
      membyte(control_byte_addr) := value;
      send_interrupt_to_the_8085_debugger
      lp := lp - 4;
      ip := ip + 1
    END
  END;
  (* CWRAM *)

```

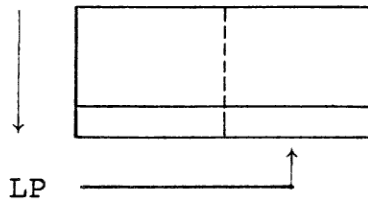


## 11.10.5 CGREG

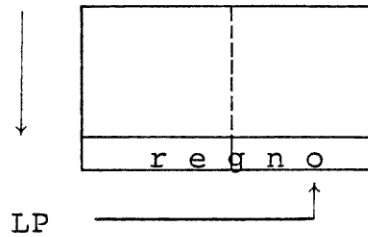
11.10.5

Control Get REGisterValue: 1F<sub>Hex</sub>IP → CGREG

STACK BEFORE:



STACK AFTER:



(\* CGREG \*)

BEGIN

```

  IF lp > ms-2 THEN exception(stack overflow);
  memword(lp+1) := cur_level * 8 + 7;
  lp := lp + 2;
  ip := ip + 1
END;  (* CGREG *)

```

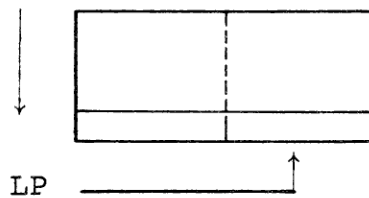
## 11.10.6 MBTES

11.10.6

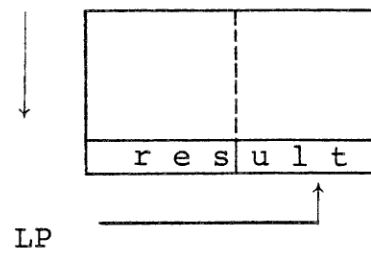
Miscellaneous Bits TEST in Status Register Value: D1<sub>Hex</sub>

IP → MBTES

STACK BEFORE:



STACK AFTER:



(\* MBTES \*)

BEGIN

```

  IF lp > ms - 2 THEN exception(stack overflow);
  IF reg.ps AND memword(ip+1) = 0 THEN
    memword(lp+1) := 0 (* false *)
  ELSE
    memword(lp+1) := 1;
    lp := lp + 2;
    ip := ip + 3
  END; (* MBTES *)

```

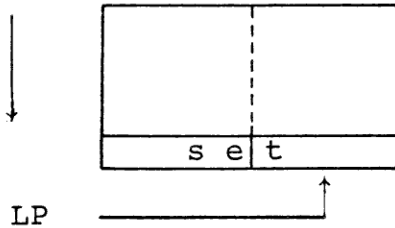
Miscellaneous Bits SET or Clear in Status Register

F<sub>1</sub>Hex

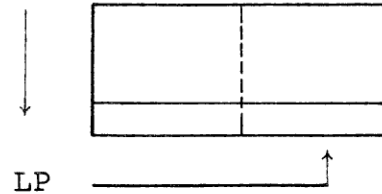
IP → 

|       |   |   |   |   |
|-------|---|---|---|---|
| MBSET | m | a | s | k |
|-------|---|---|---|---|

STACK BEFORE:



STACK AFTER:



(\* MBSET \*)

VAR

set : boolean;

mask : word;

BEGIN

mask := memword(ip + 1);

set := memword(lp - 1);

reg.ps := reg.ps AND NOT mask; (\* clear mask bit \*)

IF set THEN

reg.ps := reg.ps OR mask;

lp := lp - 2;

ip := ip + 3

END; (\* MBSET \*)

11.11 Push and Pop

11.11

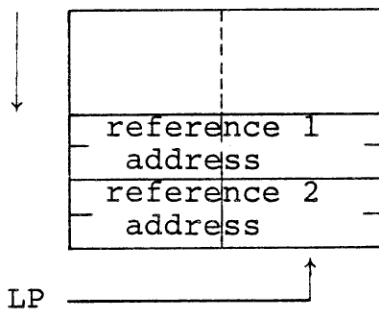
The instructions in this group are used for manipulation of message stacks, i.e. they support the language constructs push and pop.

11.11.1 LPUSH

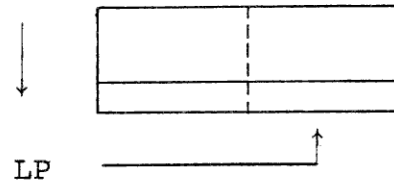
11.11.1

Language Support PUSHValue: 7C<sub>Hex</sub>IP → LPUSH

STACK BEFORE:



STACK AFTER:



(\* LPUSH \*)

CONST

```

  headerkind = 0;
  msgkind = 4;      (* offset in message header *)
  msgsize = 6;      (* offset in message header *)
  msgstart = 8;     (* offset in message header *)
  stackchain = 18;  (* offset in message header *)

```

VAR

```

  r1a, r2a, r1stack, r1, r2, r2saddr : addr;
  rekind, r2size : word;

```

BEGIN

```

  get_two_addresses; (* r1a = ref_addr, r2a = head_a *)
  get_message; (* r1 = ref *)
  IF r1.base.nilbit = 1 THEN exception(push : nil r1);
  r1stack := memaddr(r1+stackchain);
  IF r1stack.base.nilbit = 0 THEN exception(push : not empty r1);
  r2 := memaddr(r2a);
  IF r1 = r2 THEN exception(push : identical arguments);
  IF r2.base.lockbit = 1 THEN exception(push : locked r2);
  memaddr(r1a).base.nilbit := 1;
  memaddr(r1+stackchain) := r2;
  memaddr(r2a) := r1;
  r1kind := memword(r1+kind);
  IF (r2.base.nilbit=0) AND (r1kind = headerkind) THEN
    BEGIN (* copy buffer description *)
      r2size := memword(r2+msgsize);
      r2saddr := memaddr(r2+msgstart);
      memaddr(r1+msgstart) := r2saddr;
      memword(r1+msgsize) := r2size
    END;
  lp := lp - 8;
  ip := ip + 1
END; (* LPUSH *)

```

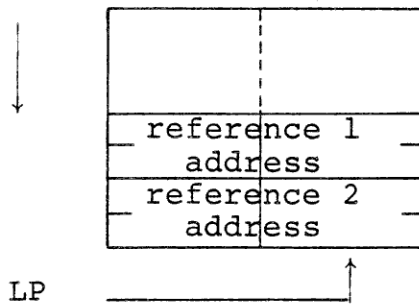
11.11.2 LPOP

11.11.2

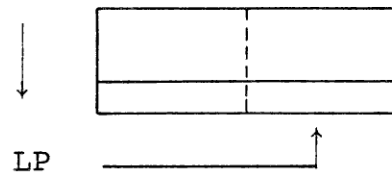
Language Support POPValue: 7D<sub>Hex</sub>IP → 

|      |
|------|
| LPOP |
|------|

STACK BEFORE:



STACK AFTER:



(\* LPOP \*)

CONST

```

  headerkind = 0;
  msgkind = 4;      (* offset in message header *)
  msgsize = 6;      (* offset in message header *)
  msgstart = 8;     (* offset in message header *)
  stackchain = 18;  (* offset in message header *)

```

VAR

```

  r1, r2, r1a, r2a, r2stack : addr;
  r2kind : word;

```

BEGIN

```

  get_two_addresses; (* r1a = ref_addr, r2a = head_a *)
  r1 := memaddr(r1a);
  IF r1.base.nilbit = 1 THEN exception(pop : r1 not nil);
  r2 := memaddr(r2a);
  IF r2.base.lockbit = 1 THEN exception(pop : locked r2);
  IF r2.base.nilbit = 1 THEN exception(pop : nil r2);
  memaddr(r1a) := r2;
  r2stack := memaddr(r2 + stackchain);
  memaddr(r2+stackchain).base.nilbit := 1;
  memaddr(r2a) := r2stack;
  r2kind := memword(r2+msgkind);
  IF r2kind = headerkind THEN
    BEGIN (* copy buffer description of msg 1 *)
      memword(r2+msgsize) := 0;
      memword(r2+msgstart).base.nilbit := 1
    END;
  lp := lp - 8;
  ip := ip + 1
END; (* LPOP *)

```

11.12 Lock Support

11.12

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

11.12.1 LLOCK

11.12.1

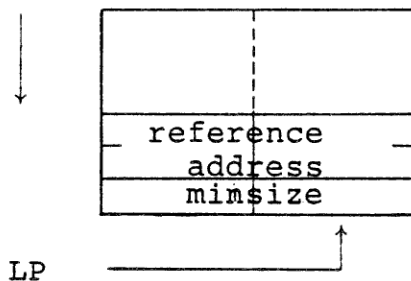
Language Support LLOCK Type on Message

Value: 7F<sub>Hex</sub>

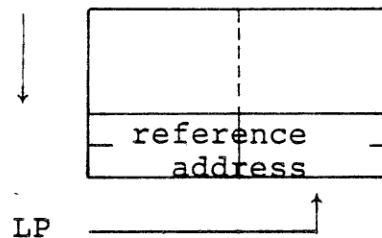
IP → 

|       |           |
|-------|-----------|
| LLOCK | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



(\* LLOCK \*)

CONST

```
msgkind = 4;    (* offset in message header *)
msgsize = 6;    (* offset in message header *)
msgstart = 8;   (* offset in message header *)
```

VAR

```
minsize, ldisp : word;
ref, ref_addr : addr;
```

BEGIN

```
ldisp := memword(ip+1);
minsize := memword(lp-1);
ref_addr := memaddr(lp-5);
ref := memaddr(ref_addr);
IF ref.base.nilbit = 1 THEN exception(nil address);
IF ref.base.lockbit = 1 THEN exception(reference locked);
IF memword(ref+msgkind) < 0 THEN exception(lock : type error);
IF memword(ref+msgsize) < minsize THEN exception(lock : size error);
memaddr(lf+ldisp) := memaddr(ref+msgstart);
lp := lp - 2;
ip := ip + 3
END;  (* LLOCK *)
```

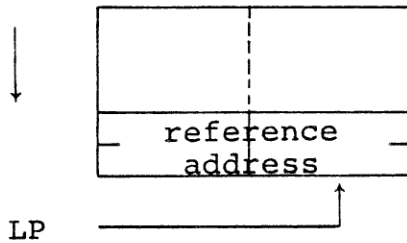


11.12.2 LRESE

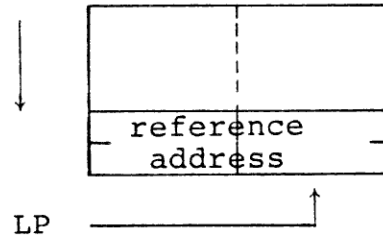
11.12.2

Language Support RESErve ReferenceValue: 7E<sub>Hex</sub>IP → LRESE

STACK BEFORE:



STACK AFTER:



(\* LRESE, reserve reference, i.e. set lockbit \*)

```

BEGIN
  ref_addr := memaddr(lp-3);
  ref := memaddr(refaddr);
  IF ref.base.nilbit = 1 THEN
    exception(nil address);
  IF ref.base.lockbit = 1 THEN
    exception(reference locked);
  ref.base.lockbit := 1;
  ip := ip + 1
END; (* LRESE *)

```

## 12. INDEXING AN ARRAY

12.

## 12.1 Range Test

12.1

This instruction has two operands. Operand 1 is a range descriptor, which describes a subrange of integers. If the value of operand 2 does not belong to the subrange described by operand 1, an exception occurs.

## 12.1.1 INTRS

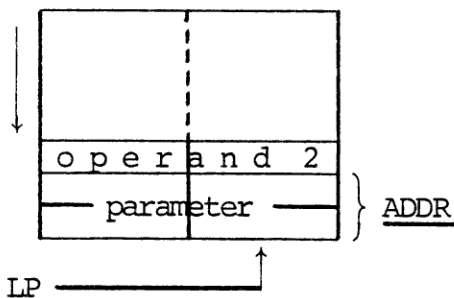
12.1.1

Index Test Range via Stack

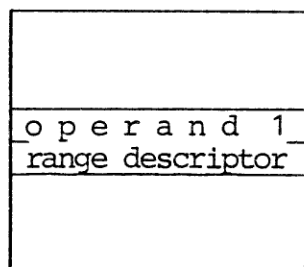
Value: 6C<sub>Hex</sub>

IP → INTRS

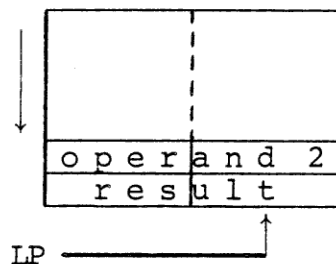
STACK BEFORE:



MEMORY:

access  
path:

STACK AFTER:



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.

```
(* INTRS *)
```

```
VAR
```

```
  range_descriptor : RECORD
```

```
    lower : integer;
```

```
    upper : integer
```

```
  END;
```

```
  index : integer;
```

```
  address : addr;
```

```
BEGIN
```

```
  index := stack(lp-5);
```

```
  address.base := stack(lp-3);
```

```
  address.disp := stack(lp-1);
```

```
  WITH range_descriptor DO
```

```
    BEGIN
```

```
      lower := mem(address);
```

```
      upper := mem(address+2);
```

```
      IF (index < lower) OR (upper < index) THEN
```

```
        exception(subrange out of bounds)
```

```
    END;
```

```
  lp := lp - 4;
```

```
  ip := ip + 1
```

```
END; (* INTRS *)
```

## 12.2 Indexing an Array

This instruction has three operands:

Operand 1 is interpreted as the address (ADDR) of the first byte of the first element of the array to be indexed.

Operand 2 is interpreted as a dope vector.

Operand 3 is interpreted as a signed integer. The value of this operand is the index.

The result is the address (ADDR) of the first byte of the array element selected by operand 3.

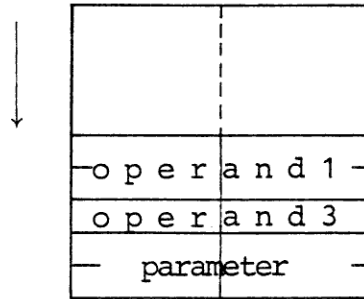
## 12.2.1 INDEX

12.2.1

INDEXValue: 6D<sub>Hex</sub>

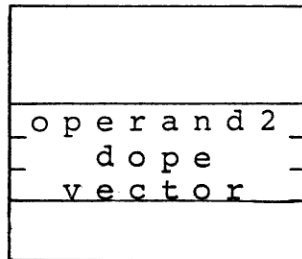
IP → INDEX

STACK BEFORE:

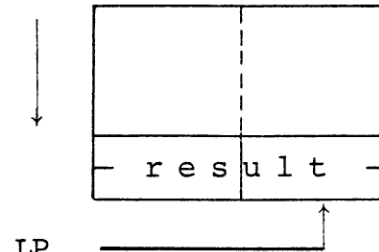


LP

MEMORY:

access  
path:SA  
0

STACK AFTER:



LP

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.

(\* INDEX \*)

```

VAR
  operand : addr;
  dope_vector : RECORD
    lower : integer;
    upper : integer;
    size : integer;
  END;
  address : addr;
  index : integer;
BEGIN
  index := stack(lp-5);
  address.base := stack(lp-3);
  address.disp := stack(lp-1);
  WITH dope_vector DO
    BEGIN
      lower := mem(address);
      upper := mem(address+2);
      size := mem(address+4);
      IF (index < lower) OR (upper < index) THEN
        exception(index error);
      operand.disp := stack(lp-7);
      stack(lp-7) := operand.disp + size * (index - lower)
    END
  END
  lp := lp - 6;
  ip := ip + 1
END      (* INDEX *)

```

12.3 Push an Element of a Packed Array

12.3

This instruction pushes an element of a packed (one-dimensional) array on the stack. The instruction has three operands:

Operand 1 is interpreted as the address (ADDR) of the first byte of the first element of the array to be indexed.

Operand 2 is interpreted as a packed dope vector.

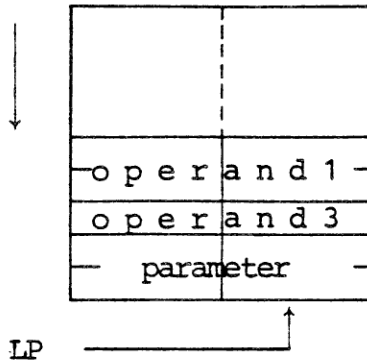
Operand 3 is interpreted as a signed integer. The value of this operand is the index.

The result is a WORD. It is the value of the array element selected by operand 3. The value is right justified with zero extension.

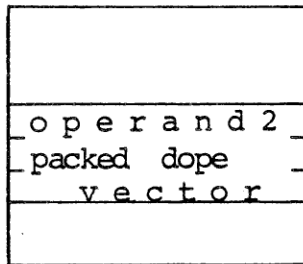
INdex Packed Array Retrieve via Stack Value: 6E<sub>Hex</sub>

IP → INPRS

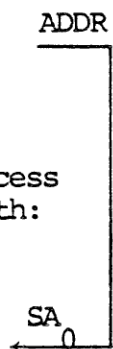
STACK BEFORE:



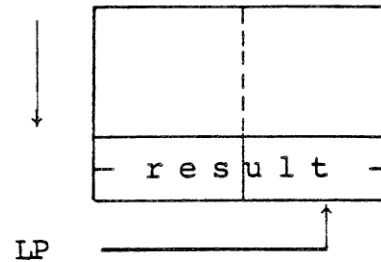
MEMORY:



access  
path:



STACK AFTER:



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.



```
(* INPRS *)
```

```
VAR
```

```
  packed_dope_vector : RECORD
    lower : integer;
    upper : integer;
    no : byte;
    size : byte
  END;
```

```
  index : integer;
  result : word;
  addr1 : addr;
  addr2 : addr;
```

```
BEGIN
```

```
  index := stack(lp-5);
  addr1.base := stack(lp-3);
  addr1.disp := stack(lp-1);
  WITH packed_dope_vector DO
```

```
    BEGIN
```

```
      lower := mem(addr1);
      upper := mem(addr1+2);
      no := membyte(addr1+4);
      size := membyte(addr1+5);
      IF (index < lower) OR (upper < index) THEN (* signed compare *)
        exception(index error);
      index := index - lower;
      addr2.base := stack(lp-9);
      addr2.disp := stack(lp-7) + (index DIV no) * 2;
      result := memword(addr2);
      result := result SHIFT ((index MOD no) * size);
      result := result SHIFT size;
      result := result AND (1 shift size - 1);
      stack(lp-9) := result
```

```
    END;
```

```
  lp := lp - 8;
```

```
  ip := ip + 1
```

```
END;      (* INPRS *)
```

12.4 Pop and Store a Value in an Element of a Packed Array

12.4

This instruction removes a value from the stack and stores the value in an element of a packed (one-dimensional) array. The instruction has four operands:

Operand 1 is interpreted as the address (ADDR) of the first byte of the first element of the array to be indexed.

Operand 2 is interpreted as a packed dope vector.

Operand 3 is interpreted as a signed integer. The value of this operand is the index.

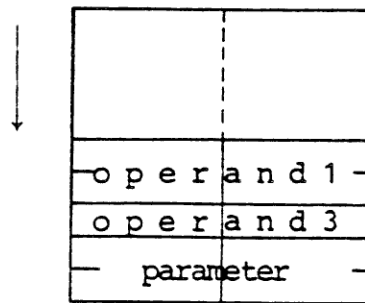
Operand 4 is the value to be stored in the array element selected by operand 3. The value of this operand is truncated from the left (high-order bits) to fit the size of the array element.

12.4.1 INPSS

12.4.1

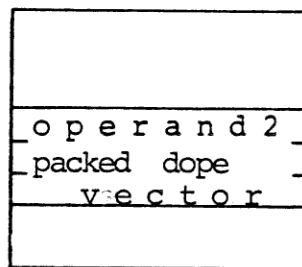
INdex Packed Array Store via StackValue: 6F<sub>Hex</sub>IP → INPSS

STACK BEFORE:



LP

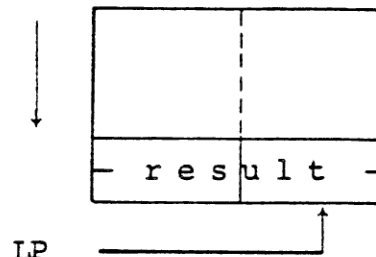
MEMORY:

access  
path:

SA

0

STACK AFTER:



LP

For a description of a packed dope vector, see subsection 12.3.1.

```
(* INPSS *)
```

```
VAR
```

```
mask : ARRAY(0..15) OF bit;
packed_dope_vector : RECORD
    lower : integer;
    upper : integer;
    no : byte;
    size : byte
END;
```

```
index : integer;
addr1 : addr;
addr2 : addr;
operand : word;
no_of_shifts : integer;
```

```
BEGIN
```

```
index := stack(lp-7);
addr1.base := stack(lp-5);
addr1.disp := stack(lp-3);
WITH packed_dope_vector DO
    BEGIN
        lower := mem(addr1);
        upper := mem(addr1+2);
        no := membyte(addr1+4);
        size := membyte(addr1+5);
        IF (index < lower) OR (upper < index) THEN
            exception(index error);
        index := index - lower;
        addr2.base := stack(lp-11);
        addr2.disp := stack(lp-9) + (index DIV no) * 2;
        operand := (stack(lp-1));
        mask := 1 SHIFT size - 1;
        no_of_shifts := 16 - (index MOD no + 1) * size;
        operand := operand shift no of shifts;
        mask := mask SHIFT no of shifts;
        result := memword(addr2);
        result := result AND (NOT mask);
        memword(addr2) := result OR operand
    END;
```

```
lp := lp - 12;
```

```
ip := ip + 1
```

```
END; (* INPSS *)
```

13. MISCELLANEOUS

13.

13.1 No Operation

13.1

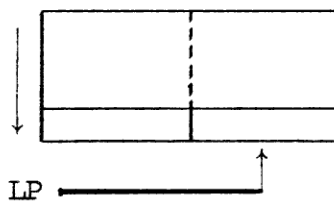
The execution of this instruction has no effect. The instruction has no operands.

13.1.1 MNOOP

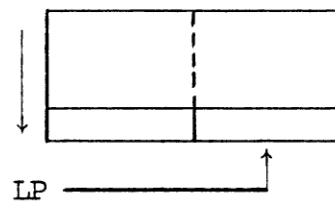
13.1.1

Miscellaneous NO OperationValue: 2F<sub>Hex</sub>IP → MNOOP

STACK BEFORE:



STACK AFTER:



(\* MNOOP \*)

BEGIN

```

    ip := ip + 1
END;  (* MNOOP *)

```

13.2 Exception

13.2

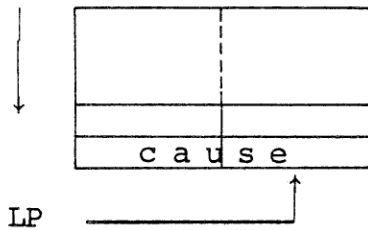
This instruction performs an unconditional jump to the program point defined in the exception point field of the incarnation descriptor. Before the jump, the multiple-wait flags are cleared in the ps register of the register set and the maxstack pointer reset to the value defined in the incarnation descriptor.

13.2.1 MXEPT

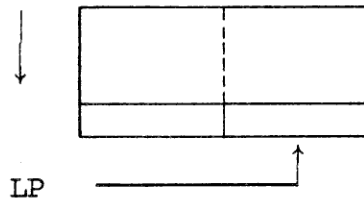
13.2.1

Miscellaneous eXcEPTIONValue: 2E<sub>Hex</sub>IP → MXEPT

STACK BEFORE:



STACK AFTER:



(\* MXEPT \*)

```

VAR
  cause : integer;
BEGIN
  cause := stack(lp-1);
  lp := lp - 2;
  exception(cause);
END;  (* MXEPT *)

```

13.3 Trap

13.3

Two instructions, with deliberately illegal operation codes, are reserved for software purposes. The parameter is interpreted by software as a routine number.

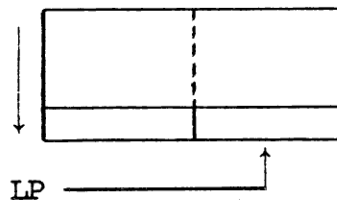
13.3.1 TRAPE

13.3.1

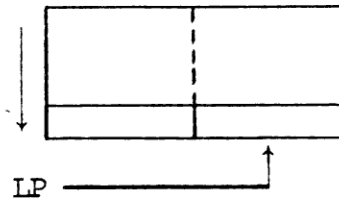
TRAP EntryValue: 2C<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| TRAPE | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



(\* TRAPE \*)

```

BEGIN
  exception(illegal opcode);
END; (* TRAPE *)

```

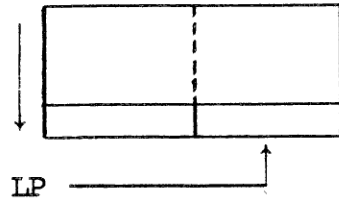
## 13.3.2 TRAPR

13.3.2

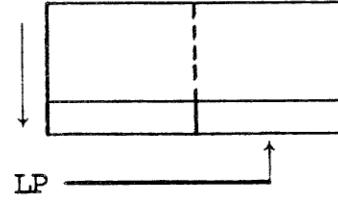
TRAP ReturnValue: 2D<sub>Hex</sub>IP → 

|       |           |
|-------|-----------|
| TRAPR | parameter |
|-------|-----------|

STACK BEFORE:



STACK AFTER:



(\* TRAPR \*)

```

BEGIN
  exception(illegal opcode);
END; (* TRAPR *)

```



This chapter contains an algorithmic description and a flow chart of the instruction fetch performed by the microprogram.

The fetch point is the point to which all instructions return after execution. A few instructions return to special points in the fetch algorithm, viz. the block input/output instructions and the 'Execute Next after Clearing Interrupt' (IONCI) instruction.

fetch: (\* instruction fetch \*)

```

IF debug_request THEN
  execute debug_request;
IF stop_mode THEN
  BEGIN
    IF breakpoint_mode = 0 THEN
      GOTO fetch;
    IF curlevel < nextlevel THEN (* interrupt with higher
                                  priority *)
      GOTO fetch1;
    IF breakpoint_mode > 0 THEN
      breakpoint_mode := breakpoint_mode - 1
    ELSE
      IF reg_set(curlevel).ip = breakpoint_ip THEN
        GOTO fetch
      END;
    GOTO execute_next;
  
```

fetch1:

```

IF curlevel < nextlevel THEN
  curlevel := nextlevel;
IF reg_set(curlevel).ib.mode = block_io_mode THEN
  BEGIN
    GOTO CASE reg_set(curlevel).ib OF
      (iorbbc, iorbb, iorbwc, iorbw, iowbbc,
       iowbb, iowbwc, iowbw)
    END;
  
```

execute\_next:

```

IF reg_set(curlevel).ib.mode = dummy_mode THEN
  BEGIN
    IF curlevel <> 0 THEN
      BEGIN
        dummy_interrupt_last := curlevel;
        dummy_interrupt_count := dummy_interrupt_count + 1;
        device(curlevel).interrupt := 0; (* clear interrupt level *)
        GOTO fetch
      END;
      (* schedule; see subsection 3.7.5 *)
      WITH monitor_reg_set DO
        BEGIN
          IF (n = -1) OR (nxt = m-1) THEN
            BEGIN
              dummy_loop_count := dummy_loop_count + 1; (* 32 bits *)
              GOTO fetch
            END;
            IF (nxt = -1) THEN
              next := k
            ELSE
              IF (k <= nxt) AND (nxt < -1) THEN
                next := nxt + 1
              ELSE
                nxt := nxt;
              IF active_queue(next).base.nilbit = 1 THEN
                BEGIN
                  nxt := nxt - 1;
                  GOTO fetch
                END;
                loadregisters(memaddr(active_queue(next)));
                GOTO fetch
              END (* with monitor_reg_set *)
            END; (* dummy_mode *)
            instruction := membyte(reg_set(curlevel).ip);
            GOTO map (instruction);
            (* end fetch *)

```



Fig. 16

RC3502 Microprogram Flow

## 15. AUTOLOAD

The autoloader 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 16.2.1) and the CPU initializes the registers, whereafter control is passed to the autoloader program residing in the first memory module.

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 autoloader 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 autoloader 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 3.3.4).

The following algorithm is executed by the microprogram:

auto:

```
FOR i := 0 TO 123 DO
  WITH reg_set (i) DO
    BEGIN
      lm := -1;
      ps := -1;
      pb := -1;
      lu := -1;
      sf := -1;
      pr := -1;
      ib := -1;
      ic := -1;
    END;
  curlevel := 1;
  WITH reg_set (curlevel) DO
    BEGIN
      ps := 0;
      pb := #h c0;
      ib := #h e0;
      ic := 2;
    END;
  device (curlevel).interrupt := 1;
  GOTO fetch;
```

16. SWITCHES AND INDICATORS

16.

16.1 Operator Control Panel

16.1

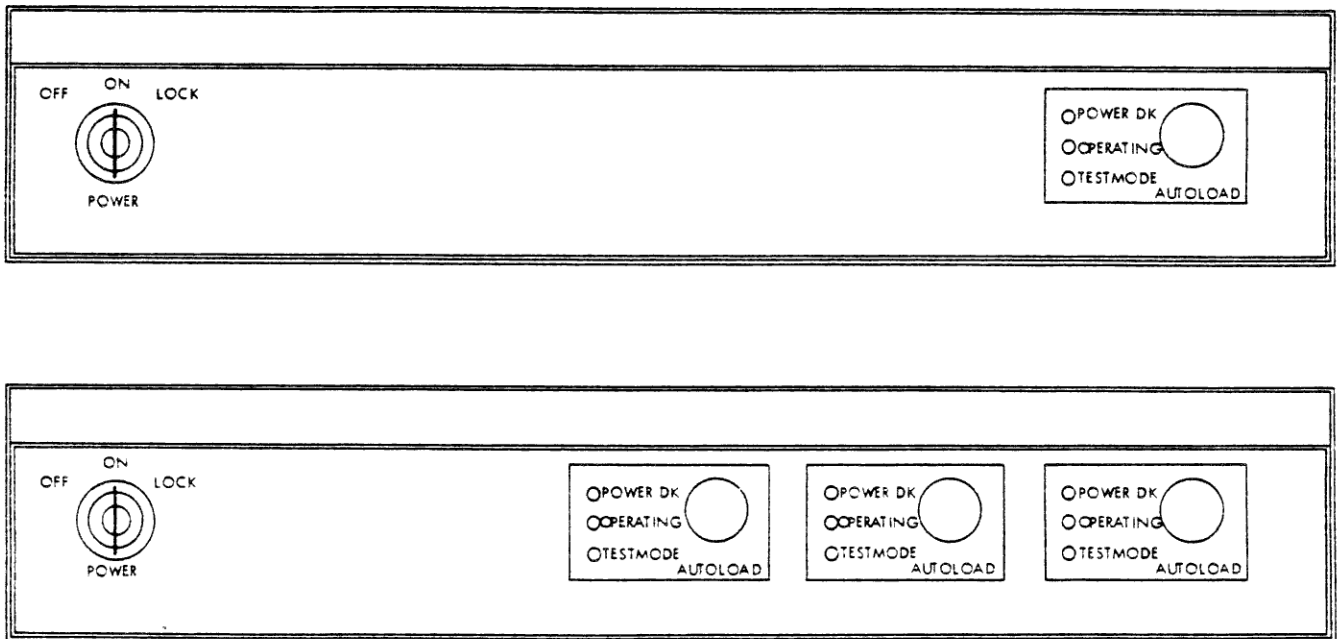


Figure 17: OCP for Rack with One RC3502 or Three RC3502's.

Power is removed from the RC3502('s) by turning the POWER key to the OFF position.

Power is applied to the RC3502('s) by turning the POWER key to the ON position (or further to the LOCK position).

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

The AUTOLOAD button initiates autoloading of the relevant RC3502.

The POWER OK lamp, when lit, indicates that the power is as it should be on the RC3502.

The OPERATING lamp, when lit, indicates that the RC3502 is running normally.

The TEST MODE lamp, when lit, indicates that the rc3502 is executing the built-in test programs.

## 16.2 Processor Front Panel

16.2

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

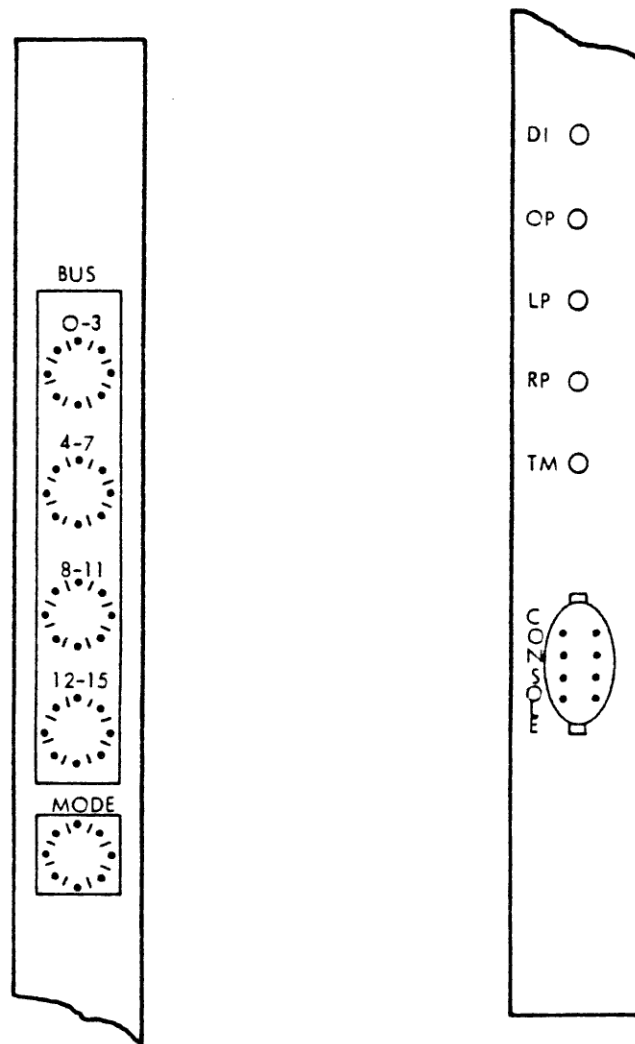


Figure 18: Processor Front Panel, Switches, Indicators, and Jack.

16.2.1 Switches

16.2.1

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

16.2.1.1 Bus Switches

16.2.1.1

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.

16.2.1.2 Debug Mode Switch

16.2.1.2

The switch marked MODE is used to enable activation of the debug console and to control the execution of the built-in test programs (subsection 16.2.2).

The switch settings 0 to 7 select a baud rate for the debug console and a program execution mode, whereas the switch settings 8 to F select only a program execution mode, i.e. the console cannot be activated, to debug mode (chapter 17).

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



Test Program Execution Modes

|           |                                                                    |
|-----------|--------------------------------------------------------------------|
| run test  | The programs are executed whenever the AUTOLOAD button is pressed. |
| skip test | The programs are not executed.                                     |
| loop      | The programs are executed in an endless loop.                      |
| no loop   | The programs are executed once.                                    |

16.2.2 Indicators

16.2.2

DI Disabled Interrupt

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

OP Operation

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 power restart or 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 power restart or 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 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 messages described below will be displayed on the debug console. For a description of the display commands, see section 17.2.

1 8085 Communication Test

Message: err 1 <dummy> <dummy>

Y5D displays 6 bytes of transmitted data.

Y70 displays 6 bytes of received data.

3 Communication Test

Microprogram interrupt of debug microprocessor.

No message is displayed, but RP and TM are lit.

5 Working Register Address Test

Message: err 5 <address> <data read>

W<address> displays data read.

7 Working Register Data Test

Message: err 7 <address> <data read>

W<address> displays data read.

9 Memory Address Test

Message: err 9 <address.displacement> <data read> <err.type>

err.type: + 1 = left parity error

+ 2 = right parity error

+ 4 = data error

Y40 displays memory module number.

M<memory module.displacement> displays data read.

B Memory Data Test

Pattern 5555.

Message: err b <address.displacement> <data read> <err.type>

err.type: see 9

Y40 displays memory module number.

M<memory module.displacement> displays data read.

D Memory Data Test

Pattern AAAA.

Message: err d<address.displacement> <data read> <err.type>

err.type: see 9

Y40 displays memory module number.

M<memory module.displacement> displays data read.

16.2.3 Jack

16.2.3

The jack, marked CONSOLE, is a 9-pin CANON jack for connection of a debug console (Teletype compatible device). The console, which is further described in chapter 17, must have the following characteristics:

8 data bits per character  
no parity bit  
1 stop bit  
300 or 1200 bps

The electrical/logical characteristics of the signals in the jack must conform to Recommendation V.24. The signals are as follows:

| <u>Pin No</u> | <u>Signal Name</u> |
|---------------|--------------------|
| 1             | +5 V               |
| 2             | +12 V              |
| 3             | received data      |
| 4             | -12 V              |
| 5             | 0 V                |
| 6             | transmitted data   |
| 7             | 0 V                |
| 8             | (not used)         |
| 9             | 0 V                |

16.3 Power Supply

16.3

The power supply (POW201) is supplied with a number of controls:

AC MAINS: Manual and automatic circuit breaker.

LOCK: Spring loaded lock switch, which inhibits manual intervention in AC MAINS and AUTO functions.

POWER OK: Indicator, which stays lit as long as power is as it should be.

AUTO: Push-button for manual generation of an autoload signal.

TEMP: Indicator, which remains on after a power break caused by overheating.

## 17. DEBUG CONSOLE

17.

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.

### 17.1 Activation of the Console

17.1

If the MODE switch (subsection 16.2.1.2) 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.

### 17.2 Display Commands

17.2

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

M <gaddr>    Modify Memory

Displays the contents of the 8 memory locations starting at <gaddr>.

W <register> Modify Working Registers

Displays the contents of the 8 working registers starting at <register>.

L <level>    Modify Working Registers

Displays the level number and the contents of the 8 working registers belonging to <level>.

Y <yaddr>    Modify Control Microprocessor RAM

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

Display commands are executed in the following situations:

1) 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.

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

- CR            The CR key terminates the current display command. The debug console will await the next command.
- +            The + key terminates the current display command and executes a display command for the succeeding 8 words (M or Y) 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 or Y) or the 8 registers on the preceding level (W).
- ESC          The ESC key terminates the current display command, but no data modification takes place. The text <ESC> is displayed. The debug console will await the next command.

2) When a control command (section 17.3) is terminated

The last executed display command is repeated, but modification of the displayed data is not allowed. The debug console will await the next command.

## 17.3 Control Commands

17.3

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 reactivate the debug console.
- S <steps>   Multi-Instruction Step  
The processor will execute <steps> instructions, stop, and reactivate the debug console.

#### 17.4      Command Parameters

17.4

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.

A global address (<gaddr>) is entered using one of the following formats:

<base> : <disp>

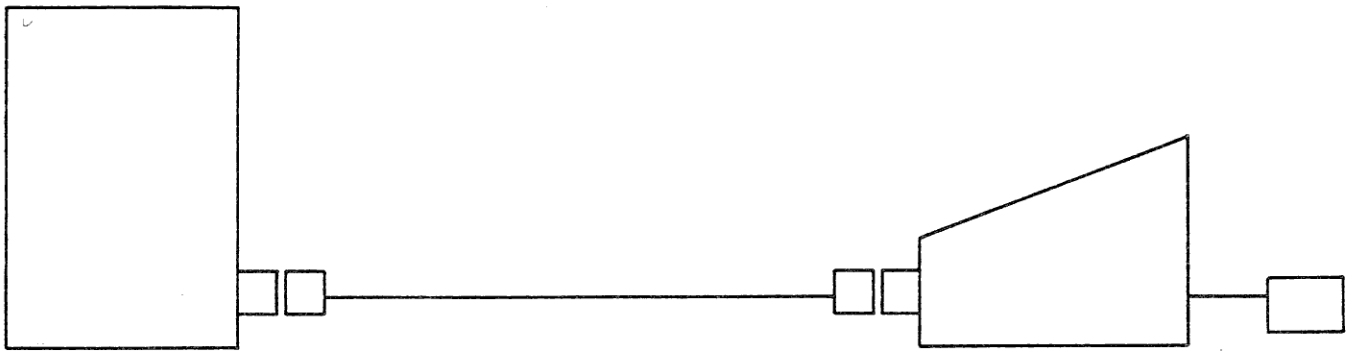
or

: <disp>

<base> is the leftmost 16 bits of the 32-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.



(CBL312 for RC822)

(CBL588 for RC831)

Console Jack  
(subsection 16.2.3)

Teletype Compatible  
Device (RC822 or RC831)

Figure 19: Connection of the Console.



The Base Instruction Set (BIS) described in chapters 4 to 14 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. If several parameters follow the operation code, the distribution of byte consumption on operation code and parameters is shown (e.g. 3 bytes (1,1,1) indicates that the instruction occupies 3 bytes, distributed on operation code: 1 byte, parameter 1: 1 byte, and parameter 2: 1 byte).

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 |
|---------------|-------------------|---------------------------------------------------|--------------------------|
| REAGDS        | CE                | REAGD,X; $X \in [0..255]$                         | 2                        |
| REALDS        | CF                | REALD,X; $X \in [0..255]$                         | 2                        |
| REC0          | 60                | 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; $X \in [0..255]$                         | 2                        |
| REVGBS        | 89                | REVGb,X; $X \in [0..255]$                         | 2                        |
| REVGDS        | E9                | REVGd,X; $X \in [0..255]$                         | 2                        |
| REVGFS        | C9                | REVGf,X; $X \in [0..255]$                         | 3 (1,1,1)                |
| REVGWS        | A9                | REVGw,X; $X \in [0..255]$                         | 2                        |
| REVLBS        | 8B                | REVLb,X; $X \in [0..255]$                         | 2                        |
| REVLDS        | EB                | REVLd,X; $X \in [0..255]$                         | 2                        |
| REVLFS        | CB                | REVLf,X; $X \in [0..255]$                         | 3 (1,1,1)                |
| REVLWS        | AB                | REVLw,X; $X \in [0..255]$                         | 2                        |
| RVSBO         | 81                | REVSb,0                                           | 1                        |
| RVSB12        | 8D                | REVSb,12                                          | 1                        |
| RVSB2         | 83                | REVSb,2                                           | 1                        |
| RVSB4         | 85                | REVSb,4                                           | 1                        |
| RVSB6         | 87                | REVSb,6                                           | 1                        |
| RVSD12        | ED                | REVSD,12                                          | 1                        |
| RVSD2         | E3                | REVSD,2                                           | 1                        |
| RVSD4         | E5                | REVSD,4                                           | 1                        |
| RVSD6         | E7                | REVSD,6                                           | 1                        |
| RVSFO         | C1                | REVSf,0,X                                         | 2                        |
| RVSF12        | CD                | REVSf,12,X                                        | 2                        |
| RVSF2         | C3                | REVSf,2,X                                         | 2                        |
| RVSF4         | C5                | REVSf,4,X                                         | 2                        |
| RVSF6         | C7                | REVSf,6,X                                         | 2                        |
| RVSW0         | A1                | REVSw,0                                           | 1                        |
| RVSW12        | AD                | REVSw,12                                          | 1                        |
| RVSW2         | A3                | REVSw,2                                           | 1                        |
| RVSW4         | A5                | REVSw,4                                           | 1                        |
| RVSW6         | A7                | REVSw,6                                           | 1                        |

| Symbolic Name | Op-Code Hex Value | Corresponding BIS Instruction and Parameter Value | Number of Bytes Occupied |
|---------------|-------------------|---------------------------------------------------|--------------------------|
| STVLBS        | 8A                | STVLB,X; $X \in [0..255]$                         | 2                        |
| STVLDS        | EA                | STVLD,X; $X \in [0..255]$                         | 2                        |
| STVLFS        | CA                | STVLF,X,Y; $X \in [0..255]$                       | 3 (1,1,1)                |
| STVLWS        | AA                | STVLW,X; $X \in [0..255]$                         | 2                        |
| SVSB0         | 80                | STVSB,0                                           | 1                        |
| SVSB2         | 82                | STVSB,2                                           | 1                        |
| SVSB28        | 9C                | STVSB,28                                          | 1                        |
| SVSB29        | 9D                | STVSB,29                                          | 1                        |
| SVSB30        | 9E                | STVSB,30                                          | 1                        |
| SVSB31        | 9F                | STVSB,31                                          | 1                        |
| SVSB4         | 84                | STVSB,4                                           | 1                        |
| SVSB6         | 86                | STVSB,6                                           | 1                        |
| SVSF0         | C0                | STVSF,0,X                                         | 2                        |
| SVSF2         | C2                | STVSF,2,X                                         | 2                        |
| SVSF28        | DC                | STVSF,28,X                                        | 2                        |
| SVSF29        | DD                | STVSF,29,X                                        | 2                        |
| SVSF30        | DE                | STVSF,30,X                                        | 2                        |
| SVSF31        | DF                | STVSF,31,X                                        | 2                        |
| SVSF4         | C4                | STVSF,4,X                                         | 2                        |
| SVSF6         | C6                | STVSF,6,X                                         | 2                        |
| SVSW0         | A0                | STVSW,0                                           | 1                        |
| SVSW2         | A2                | STVSW,2                                           | 1                        |
| SVSW28        | BC                | STVSW,28                                          | 1                        |
| SVSW29        | BD                | STVSW,29                                          | 1                        |
| SVSW30        | BE                | STVSW,30                                          | 1                        |
| SVSW31        | BF                | STVSW,31                                          | 1                        |
| SVSW4         | A4                | STVSW,4                                           | 1                        |
| SVSW6         | A6                | STVSW,6                                           | 1                        |

19. INSTRUCTION EXECUTION TIMES

19.

The calculated execution times of the RC3502 Rev. 5 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 execution time includes a non-interrupt instruction fetch, with normal microinstruction flow. It is assumed that block I/O is stopped by count, and not because of end of medium.

For each memory write, 2 steps are added, and for each memory read, 1 step or 0.5 step when there are microinstructions between the request (address) and the read.

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. This deviation is indicated by percentages in the table.

On level shift add:

|           |           |           |
|-----------|-----------|-----------|
| level ≤ 7 | 12.0 step | 2.6 μsec. |
| level > 7 | 18.0 step | 3.1 μsec. |

Note: On block I/O do not add level shifts.

Scheduling on level 0 is done by a level shift to level 1, where the SCHED instruction is executed.

The following instructions are interruptable:

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

The following abbreviations are employed:

|            |                                                                           |
|------------|---------------------------------------------------------------------------|
| X          | time required for one microinstruction, viz. 217.0 nanoseconds            |
| SH         | shiftout + shiftin time (controller dependent)                            |
| W          | number of words                                                           |
| Weq        | number of equal words before difference                                   |
| B          | number of bytes                                                           |
| Beq        | number of equal bytes before difference                                   |
| E          | element size                                                              |
| S          | element number in word                                                    |
| F          | first bit number in a field instruction bit;<br>0 is most significant bit |
| L          | last bit number in a field instruction bit;<br>0 is most significant bit  |
| R          | static procedure level difference                                         |
| NF         | number of empty queues before the SCHEDULEd process is found              |
| P          | number of processes in queue before the CSTOPped process                  |
| $\mu$ sec. | microsecond(s)                                                            |

| Instruction                                      | Step  | μsec. |     |
|--------------------------------------------------|-------|-------|-----|
| ABS                                              | 25.0  | 5.4   | 4%  |
| ADD                                              | 30.0  | 6.5   |     |
| AND                                              | 29.0  | 6.3   |     |
| CEXCH                                            | 99.0  | 21.5  |     |
| CGREG                                            | 17.0  | 3.7   |     |
| CLLST                                            |       |       |     |
| empty chain                                      | 112.0 | 24.3  |     |
| not empty chain                                  | 132.0 | 28.6  |     |
| COMPL                                            | 22.0  | 4.8   |     |
| CRC16                                            | 62.0  | 13.5  | 13% |
| CRELE                                            |       |       |     |
| open semaphore                                   | 157.8 | 34.2  | 2%  |
| passive semaphore                                | 136.8 | 29.7  | 2%  |
| closed semaphore<br>start level = 0              |       |       |     |
| active queue empty,<br>semaphore                 |       |       |     |
| going passive                                    | 251.3 | 54.5  | 1%  |
| staying closed                                   | 266.3 | 57.8  | 1%  |
| active queue not empty,<br>semaphore             |       |       |     |
| going passive                                    | 257.3 | 55.8  | 1%  |
| staying closed                                   | 272.3 | 59.1  | 1%  |
| closed semaphore<br>start level > 0<br>semaphore |       |       |     |
| going passive                                    | 179.8 | 39.0  | 3%  |
| staying closed                                   | 194.8 | 42.3  | 2%  |
| CRGET                                            | 23.0  | 5.0   |     |
| CRPUT                                            | 30.0  | 6.5   |     |
| CRRAM                                            | 33.5  | 7.3   | 1%  |
| CSELL                                            |       |       |     |
| level = 0, new level > 0,<br>active queue        |       |       |     |
| going empty                                      | 292.0 | 63.4  | 2%  |
| staying not empty                                | 307.0 | 66.6  | 2%  |
| level > 0, new level = 0,<br>active queue        |       |       |     |
| empty                                            | 299.5 | 65.0  | 1%  |
| not empty                                        | 302.5 | 65.6  | 1%  |

| Instruction                                      | Step  | µsec. |    |
|--------------------------------------------------|-------|-------|----|
| level > 0,<br>new level > 0                      | 249.0 | 54.0  | 2% |
| level = 0,<br>new level = 0                      | 194.5 | 42.2  | 2% |
| CSENS                                            |       |       |    |
| open semaphore<br>going passive                  | 121.0 | 26.3  |    |
| open semaphore<br>staying open                   | 136.0 | 29.5  |    |
| passive semaphore                                | 96.0  | 20.8  |    |
| closed semaphore                                 | 97.0  | 21.0  |    |
| CSIGN                                            |       |       |    |
| open semaphore                                   | 151.0 | 32.8  |    |
| passive semaphore                                | 130.0 | 28.2  |    |
| closed semaphore<br>start level = 0              |       |       |    |
| active queue empty,<br>semaphore                 |       |       |    |
| going passive                                    | 244.5 | 53.1  |    |
| staying closed                                   | 259.5 | 56.3  |    |
| active queue not empty,<br>semaphore             |       |       |    |
| going passive                                    | 250.5 | 54.4  |    |
| staying closed                                   | 265.5 | 57.6  |    |
| closed semaphore<br>start level > 0<br>semaphore |       |       |    |
| going passive                                    | 173.0 | 37.5  | 1% |
| staying closed                                   | 188.0 | 40.8  | 1% |
| CSKIP                                            |       |       |    |
| active queue empty                               | 35.0  | 7.6   |    |
| active queue not empty                           | 58.0  | 12.6  |    |
| CSLEV                                            | 26.5  | 5.8   | 6% |
| CSTD                                             | 136.0 | 29.5  | 1% |
| CSTOP                                            |       |       |    |
| process not in queue                             |       |       |    |
| level = 0                                        | 97.5  | 21.2  | 1% |
| level > 0                                        | 146.5 | 31.8  |    |
| process in queue                                 |       |       |    |
| going empty                                      |       |       |    |
| level = 0                                        | 150.5 | 32.7  |    |
| level > 0                                        | 199.5 | 43.3  |    |

| Instruction                                                            | Step        | µsec.     |    |
|------------------------------------------------------------------------|-------------|-----------|----|
| process in queue<br>staying not empty,<br>last process<br>in queue     |             |           |    |
| level = 0                                                              | 16.0P+177.5 | 3.5P+38.5 |    |
| level > 0                                                              | 16.0P+226.5 | 3.5P+49.2 |    |
| process in queue<br>staying not empty,<br>not last process<br>in queue |             |           |    |
| level = 0                                                              | 16.0P+162.5 | 3.5P+35.3 |    |
| level > 0                                                              | 16.0P+211.5 | 3.5P+45.9 |    |
| CWAIT                                                                  |             |           |    |
| open semaphore<br>going passive                                        | 120.0       | 26.0      |    |
| open semaphore<br>staying open                                         | 135.0       | 29.3      |    |
| passive semaphore,<br>level = 0,<br>active queue                       |             |           |    |
| going empty                                                            | 207.0       | 44.9      |    |
| staying not empty                                                      | 222.0       | 48.2      |    |
| passive semaphore,<br>level > 0                                        | 167.0       | 36.2      | 2% |
| closed semaphore,<br>level = 0,<br>active queue                        |             |           |    |
| going empty                                                            | 229.0       | 49.7      |    |
| staying not empty                                                      | 244.0       | 52.9      |    |
| closed semaphore,<br>level > 0                                         | 189.0       | 41.0      | 2% |
| CWRAM                                                                  | 36.0        | 7.8       |    |
| CWTAC                                                                  |             |           |    |
| reference nil                                                          | 64.0        | 13.9      | 5% |
| reference not nil                                                      | 55.0        | 11.9      |    |
| DIV                                                                    | 99.0        | 21.5      | 9% |
| EQ                                                                     | 30.5        | 6.6       | 2% |
| GE                                                                     | 31.0        | 6.7       | 3% |
| GT                                                                     | 31.0        | 6.7       | 3% |
| INDEX                                                                  |             |           |    |
| elementsize <= 60                                                      | E+84.0      | 0.2E+18.2 |    |
| elementsize > 60                                                       | 144.0       | 31.2      | 6% |



| Instruction | Step                | $\mu\text{sec.}$    |     |
|-------------|---------------------|---------------------|-----|
| INPRS       | $2SE+3S+E+156.5$    | $.2(2SE+3S+E)+34.0$ | 7%  |
| INPSS       | $SE+3S+2E+187.0$    | $.2(SE+3S+2E)+40.6$ | 8%  |
| INTRS       | 46.0                | 10.0                |     |
| IOCCI       | 21.0                | 4.6                 |     |
| IOCDA       | 46.5                | 10.1                |     |
| IOGI        | $SH/X+46.0$         | $SH+10.0$           |     |
| IOGO        | 37.5                | 8.1                 |     |
| IOIBX       | 103.5               | 22.5                |     |
| IONCI       | 18.0                | 3.9                 |     |
| IORBB       | $(SH/X+40.5)B+96.5$ | $(SH+8.8)B+20.9$    | 1%  |
| IORBBC      | $(SH/X+40.5)B+89.5$ | $(SH+8.8)B+19.4$    | 1%  |
| IORBW       | $(SH/X+47.0)W+96.5$ | $(SH+10.2)W+20.9$   | 9%  |
| IORBWC      | $(SH/X+47.0)W+89.5$ | $(SH+10.2)W+19.4$   | 9%  |
| IORS        | $SH/X+36.5$         | $SH+7.9$            |     |
| IORW        | $SH/X+36.5$         | $SH+7.9$            |     |
| IOWBB       | $(SH/X+40.5)B+96.5$ | $(SH+8.8)B+20.9$    | 1%  |
| IOWBBC      | $(SH/X+40.5)B+89.5$ | $(SH+8.8)B+19.4$    | 1%  |
| IOWBW       | $(SH/X+45.5)W+96.5$ | $(SH+9.9)W+20.9$    | 8%  |
| IOWBWC      | $(SH/X+45.5)W+89.5$ | $(SH+9.9)W+19.4$    | 8%  |
| IOWC        | 28.5                | 6.2                 |     |
| IOWW        | 32.5                | 7.1                 |     |
| JMCHT       | 66.0                | 14.3                | 6%  |
| JMPHC       | 29.0                | 6.3                 | 10% |
| JMPPD       | 23.0                | 5.0                 |     |
| JMPRW       | 18.0                | 3.9                 | 8%  |
| JMZEQ       | 25.5                | 5.5                 | 8%  |
| JMZGE       | 25.5                | 5.5                 | 8%  |
| JMZGT       | 26.0                | 5.6                 | 10% |
| JMZLE       | 26.5                | 5.8                 | 8%  |

| Instruction                                                     | Step  | μsec. |    |
|-----------------------------------------------------------------|-------|-------|----|
| JMZLT                                                           | 25.5  | 5.5   | 8% |
| JMZNE                                                           | 25.5  | 5.5   | 8% |
| LE                                                              | 31.0  | 6.7   | 3% |
| LLOCK                                                           | 87.5  | 19.0  | 2% |
| LPOP                                                            |       |       |    |
| no description clear                                            | 129.0 | 28.0  |    |
| description clear                                               | 146.0 | 31.7  |    |
| LPUSH                                                           |       |       |    |
| no description copy                                             | 136.0 | 29.5  |    |
| description copy                                                | 177.0 | 38.4  |    |
| LRESE                                                           | 40.0  | 8.7   | 9% |
| LT                                                              | 31.0  | 6.7   | 3% |
| MADD                                                            | 29.0  | 6.3   |    |
| MBSET                                                           | 26.0  | 5.6   | 8% |
| MBTES                                                           | 26.5  | 5.8   | 8% |
| MCIS                                                            |       |       |    |
| open semaphore<br>going passive                                 | 165.5 | 35.9  |    |
| open semaphore<br>staying open                                  | 180.5 | 39.2  |    |
| passive semaphore,<br>level = 0,<br>active queue<br>going empty | 252.5 | 54.8  |    |
| staying not empty                                               | 267.5 | 58.0  |    |
| passive semaphore,<br>level > 0                                 | 212.5 | 46.1  | 1% |
| timer = 0                                                       | 60.5  | 13.1  |    |
| MCIST                                                           |       |       |    |
| open semaphore<br>going passive                                 | 165.5 | 35.9  |    |
| open semaphore<br>staying open                                  | 180.5 | 39.2  |    |
| passive semaphore,<br>level = 0,<br>active queue<br>going empty | 252.5 | 54.8  |    |
| staying not empty                                               | 267.5 | 58.0  |    |

| Instruction                                                           | Step       | µsec.     |     |
|-----------------------------------------------------------------------|------------|-----------|-----|
| passive semaphore,<br>level > 0                                       | 212.5      | 46.1      | 1%  |
| timer = 0                                                             | 60.5       | 13.1      |     |
| MCIT                                                                  |            |           |     |
| level = 0,<br>active queue<br>going empty                             | 128.5      | 27.9      |     |
| staying not empty                                                     | 143.5      | 31.1      |     |
| level > 0                                                             | 71.5       | 15.5      | 4%  |
| timer = 0                                                             | 59.5       | 12.9      |     |
| MNOOP                                                                 | 10.0       | 2.2       |     |
| MOD                                                                   | 98.5       | 21.4      | 10% |
| MOVEB                                                                 | 18.0B+40.0 | 3.9B+8.7  |     |
| MOVEG                                                                 |            |           |     |
| even bytes                                                            |            |           |     |
| both even                                                             | 17.5W+41.0 | 3.8W+8.9  |     |
| one odd                                                               | 24.0W+41.0 | 5.2W+8.9  | 2%  |
| both odd                                                              | 30.5W+41.0 | 6.6W+8.9  |     |
| odd bytes                                                             |            |           |     |
| both even                                                             | 30.5W+56.0 | 6.6W+12.2 |     |
| one odd                                                               | 24.0W+56.0 | 5.2W+12.2 | 2%  |
| both odd                                                              | 17.5W+56.0 | 3.8W+12.2 |     |
| MSUB                                                                  | 29.0       | 6.3       |     |
| MTIME                                                                 |            |           |     |
| empty timer chain                                                     | 21.0       | 4.6       |     |
| For each process in<br>timer chain add one<br>of the following times: |            |           |     |
| own.timer = 0                                                         | 23.0       | 5.0       |     |
| own.timer > 0                                                         | 31.0       | 6.7       |     |
| own.timer = 1,<br>level = 0,<br>semaphore                             |            |           |     |
| not used                                                              | 175.5      | 38.1      |     |
| going passive                                                         | 215.5      | 46.8      |     |
| staying closed                                                        | 230.5      | 50.0      |     |
| own.timer = 1,<br>level > 0,<br>semaphore                             |            |           |     |
| not used                                                              | 89.0       | 19.3      | 2%  |
| going passive                                                         | 129.0      | 28.0      | 2%  |
| staying closed                                                        | 144.0      | 31.2      | 1%  |
| MUL                                                                   | 96.5       | 20.9      | 10% |

| Instruction                                      | Step  | μsec. |    |
|--------------------------------------------------|-------|-------|----|
| MWI                                              |       |       |    |
| level = 0,<br>active queue                       |       |       |    |
| going empty                                      | 93.0  | 20.2  |    |
| staying not empty                                | 108.0 | 23.4  |    |
| level > 0                                        | 36.0  | 7.8   | 8% |
| MWIS                                             |       |       |    |
| open semaphore,<br>going passive                 | 142.0 | 30.8  |    |
| open semaphore,<br>staying open                  | 157.0 | 34.1  |    |
| passive semaphore,<br>level = 0,<br>active queue |       |       |    |
| going empty                                      | 229.0 | 49.7  |    |
| staying not empty                                | 244.0 | 52.9  |    |
| passive semaphore,<br>level > 0                  | 189.0 | 41.0  | 2% |
| MWIST                                            |       |       |    |
| open semaphore,<br>going passive                 | 142.0 | 30.8  |    |
| open semaphore,<br>staying open                  | 157.0 | 34.1  |    |
| passive semaphore,<br>level = 0,<br>active queue |       |       |    |
| going empty                                      | 229.0 | 49.7  |    |
| staying not empty                                | 244.0 | 52.9  |    |
| passive semaphore,<br>level > 0                  | 189.0 | 41.0  | 2% |
| timer = 0                                        | 37.0  | 8.0   |    |
| MWIT                                             |       |       |    |
| level = 0,<br>active queue                       |       |       |    |
| going empty                                      | 106.0 | 23.0  |    |
| staying not empty                                | 121.0 | 26.3  |    |
| level > 0                                        | 49.0  | 10.6  | 6% |
| timer = 0                                        | 37.0  | 8.0   |    |
| MWST                                             |       |       |    |
| open semaphore,<br>going passive                 | 142.0 | 30.8  |    |
| open semaphore,<br>staying open                  | 157.0 | 34.1  |    |

| Instruction                                      | Step  | µsec. |    |
|--------------------------------------------------|-------|-------|----|
| passive semaphore,<br>level = 0,<br>active queue |       |       |    |
| going empty                                      | 229.0 | 49.7  |    |
| staying not empty                                | 244.0 | 52.9  |    |
| passive semaphore,<br>level > 0                  | 189.0 | 41.0  | 2% |
| timer = 0                                        | 37.0  | 8.0   |    |
| MWT                                              |       |       |    |
| level = 0,<br>active queue                       |       |       |    |
| going empty                                      | 106.0 | 23.0  |    |
| staying not empty                                | 121.0 | 26.3  |    |
| level > 0                                        | 49.0  | 10.6  | 6% |
| timer = 0                                        | 37.0  | 8.0   |    |
| MWTAC                                            |       |       |    |
| Wait Start                                       |       |       |    |
| Causes Reason Level                              |       |       |    |
| I I -                                            | 40.5  | 8.8   | 1% |
| S S -                                            | 54.5  | 11.8  | 1% |
| S not S 0                                        |       |       |    |
| active queue                                     |       |       |    |
| going empty                                      | 110.0 | 23.9  |    |
| staying not empty                                | 125.0 | 27.1  |    |
| S not S > 0                                      | 53.0  | 11.5  | 6% |
| T T -                                            | 46.5  | 10.1  | 1% |
| T not T 0                                        |       |       |    |
| active queue                                     |       |       |    |
| going empty                                      | 103.0 | 22.4  |    |
| staying not empty                                | 118.0 | 25.6  |    |
| T not T > 0                                      | 46.0  | 10.0  | 7% |
| IS I -                                           |       |       |    |
| semaphore                                        |       |       |    |
| going passive                                    | 138.5 | 30.1  |    |
| staying closed                                   | 153.5 | 33.3  |    |
| IS S -                                           | 54.5  | 11.8  | 1% |
| IT I -                                           | 51.5  | 11.2  | 1% |
| IT T -                                           | 46.5  | 10.1  | 1% |
| TS T -                                           | 65.5  | 14.2  | 1% |
| TS S -                                           | 54.5  | 11.8  | 1% |
| TS I 0                                           |       |       |    |
| active queue                                     |       |       |    |
| going empty                                      | 121.0 | 26.3  |    |
| staying not open                                 | 136.0 | 29.5  |    |
| TS I > 0                                         | 64.0  | 13.9  | 5% |
| IST I -                                          |       |       |    |
| semaphore                                        |       |       |    |
| going passive                                    | 149.5 | 32.4  |    |
| staying closed                                   | 164.5 | 35.7  |    |
| IST S -                                          | 54.5  | 11.8  | 1% |
| IST T -                                          | 65.5  | 14.2  | 1% |

| Instruction  | Step      | $\mu$ sec. |     |
|--------------|-----------|------------|-----|
| NE           | 30.5      | 6.6        | 2%  |
| NEG          | 24.0      | 5.2        |     |
| NOT          | 24.5      | 5.3        | 2%  |
| OR           | 29.0      | 6.3        |     |
| PCALD        | 81.0      | 17.6       |     |
| PCALS        | 7.0R+86.3 | 1.5R+18.7  | 7%  |
| PEXIT        | 44.5      | 9.7        | 3%  |
| REAAD        | 41.0      | 8.9        | 7%  |
| READB        | 37.5      | 8.1        |     |
| READW        | 38.5      | 8.4        |     |
| REAGD        | 33.0      | 7.2        | 5%  |
| REAGDS       | 31.8      | 6.9        | 9%  |
| READD        | 7.0R+43.3 | 1.5R+9.4   | 11% |
| REALD        | 33.0      | 7.2        | 5%  |
| REALDS       | 31.8      | 6.9        | 9%  |
| REARD        | 32.0      | 6.9        | 5%  |
| READS        | 45.0      | 9.8        | 3%  |
| REAXD        | 25.0      | 5.4        |     |
| RECO         | 16.0      | 3.5        |     |
| REC1-REC15   | 18.0      | 3.9        |     |
| RECHD        | 41.0      | 8.9        | 7%  |
| RECHW        | 25.0      | 5.4        | 6%  |
| RECHWS       | 22.8      | 4.9        | 12% |
| RENHB        | 21.0      | 4.6        | 7%  |
| RENPB        | 18.0      | 3.9        |     |
| REVAB        | 42.0      | 9.1        | 7%  |
| REVAD        |           |            |     |
| even address | 55.0      | 11.9       | 5%  |
| odd address  | 67.0      | 14.5       | 4%  |

| Instruction  | Step        | $\mu$ sec.    |     |
|--------------|-------------|---------------|-----|
| REVAF        |             |               |     |
| even address | -F+79.3     | -.2F+17.2     | 9%  |
| odd address  | -F+85.3     | -.2F+18.5     | 8%  |
| REVAW        |             |               |     |
| even address | 41.5        | 9.0           | 7%  |
| odd address  | 47.5        | 10.3          | 6%  |
| REVGB        | 34.0        | 7.4           | 4%  |
| REVGBS       | 32.8        | 7.1           | 9%  |
| REVGD        |             |               |     |
| even address | 47.0        | 10.2          | 3%  |
| odd address  | 59.0        | 12.8          | 3%  |
| REVGDS       |             |               |     |
| even address | 45.8        | 9.9           | 6%  |
| odd address  | 57.8        | 12.5          | 5%  |
| REVGF        |             |               |     |
| even address | -F+71.3     | -.2F+15.5     | 8%  |
| odd address  | -F+77.3     | -.2F+16.8     | 7%  |
| REVGFS       |             |               |     |
| even address | -F+70.0     | -.2F+15.2     | 10% |
| odd address  | -F+76.0     | -.2F+16.5     | 9%  |
| REVGW        |             |               |     |
| even address | 33.5        | 7.3           | 4%  |
| odd address  | 39.5        | 8.6           | 4%  |
| REVGWS       |             |               |     |
| even address | 32.3        | 7.0           | 9%  |
| odd address  | 38.3        | 8.3           | 7%  |
| REVIB        | 7.0R+43.3   | 1.5R+9.4      | 11% |
| REVID        |             |               |     |
| even address | 7.0R+56.3   | 1.5R+12.2     | 9%  |
| odd address  | 7.0R+68.3   | 1.5R+14.8     | 7%  |
| REVIF        |             |               |     |
| even address | 7.0R-F+80.5 | 1.5R-.2F+17.5 | 11% |
| odd address  | 7.0R-F+86.5 | 1.5R-.2F+18.8 | 10% |
| REVIW        |             |               |     |
| even address | 7.0R+42.8   | 1.5R+9.3      | 11% |
| odd address  | 7.0R+48.8   | 1.5R+10.6     | 10% |
| REVLB        | 34.0        | 7.4           | 4%  |
| REVLBS       | 32.8        | 7.1           | 9%  |
| REVL D       |             |               |     |
| even address | 47.0        | 10.2          | 3%  |
| odd address  | 59.0        | 12.8          | 3%  |

| Instruction  | Step       | $\mu$ sec. |     |
|--------------|------------|------------|-----|
| REVLDS       |            |            |     |
| even address | 45.8       | 9.9        | 6%  |
| odd address  | 57.8       | 12.5       | 5%  |
| REVL F       |            |            |     |
| even address | -F+71.3    | -.2F+15.5  | 8%  |
| odd address  | -F+77.4    | -.2F+16.8  | 7%  |
| REVLFS       |            |            |     |
| even address | -F+70.0    | -.2F+15.2  | 10% |
| odd address  | -F+76.0    | -.2F+16.5  | 9%  |
| REVLW        |            |            |     |
| even address | 33.5       | 7.3        | 4%  |
| odd address  | 39.5       | 8.6        | 4%  |
| REVLWS       |            |            |     |
| even address | 32.3       | 7.0        | 9%  |
| odd address  | 38.3       | 8.3        | 7%  |
| REVPD        | 34.5       | 7.5        |     |
| REVPW        | 22.5       | 4.9        |     |
| REVSB        | 46.0       | 10.0       | 3%  |
| REVSD        |            |            |     |
| even address | 59.0       | 12.8       | 3%  |
| odd address  | 71.0       | 15.4       | 2%  |
| REVSF        |            |            |     |
| even address | -F+83.3    | -.2F+18.1  | 6%  |
| odd address  | -F+89.3    | -.2F+19.4  | 6%  |
| REVSM        |            |            |     |
| even address | 17.5W+34.0 | 3.8W+7.4   |     |
| odd address  | 23.5W+34.0 | 5.1W+7.4   |     |
| REVSW        |            |            |     |
| even address | 45.5       | 9.9        | 3%  |
| odd address  | 51.5       | 11.2       | 3%  |
| RVSB0        | 39.0       | 8.5        |     |
| RVSB2-RVSB12 | 40.0       | 8.7        |     |
| RVSD0        |            |            |     |
| even address | 52.0       | 11.3       |     |
| odd address  | 64.0       | 13.9       |     |
| RVSD2-RVSD12 |            |            |     |
| even address | 53.0       | 11.5       |     |
| odd address  | 65.0       | 14.1       |     |
| RVSF0        |            |            |     |
| even address | -F+76.3    | -.2F+16.6  | 5%  |
| odd address  | -F+82.3    | -.2F+17.9  | 4%  |



| Instruction  | Step          | $\mu$ sec.   |     |
|--------------|---------------|--------------|-----|
| RVSF2-RVSF12 |               |              |     |
| even address | -F+77.3       | -.2F+16.8    | 4%  |
| odd address  | -F+83.3       | -.2F+18.1    | 4%  |
| RVSW0        |               |              |     |
| even address | 38.5          | 8.4          |     |
| odd address  | 44.5          | 9.7          |     |
| RVSW2-RVSW12 |               |              |     |
| even address | 39.5          | 8.6          |     |
| odd address  | 45.5          | 9.9          |     |
| SCHED        | 34.5NF+192.0  | 7.5NF+41.7   | 4%  |
| SETAD        |               |              |     |
| even address | 36.0W+79.5    | 7.8W+17.3    | 10% |
| odd address  | 42.0W+79.5    | 9.1W+17.3    | 8%  |
| SETCR        | 108.5W+85.5   | 23.5W+18.6   | 18% |
| SETDI        |               |              |     |
| both even    | 45.0W+80.5    | 9.8W+17.5    | 8%  |
| one odd      | 51.0W+80.5    | 11.1W+17.5   | 7%  |
| both odd     | 57.0W+80.5    | 12.4W+17.5   | 6%  |
| SETEQ        |               |              |     |
| result true  |               |              |     |
| both even    | 38.5W+81.5    | 8.4W+17.7    | 4%  |
| one odd      | 44.5W+81.5    | 9.7W+17.7    | 4%  |
| both odd     | 50.5W+81.5    | 11.0W+17.7   | 4%  |
| result false |               |              |     |
| both even    | 38.5Weq+106.0 | 8.4Weq+23.0  | 3%  |
| one odd      | 44.5Weq+112.0 | 9.7Weq+24.3  | 3%  |
| both odd     | 50.5Weq+118.0 | 11.0Weq+25.6 | 3%  |
| SETIN        |               |              |     |
| both even    | 45.0W+80.5    | 9.8W+17.5    | 4%  |
| one odd      | 51.0W+80.5    | 11.1W+17.5   | 4%  |
| both odd     | 57.0W+80.5    | 12.4W+17.5   | 4%  |
| SETSB        |               |              |     |
| result true  |               |              |     |
| both even    | 38.5W+81.5    | 8.4W+17.7    | 4%  |
| one odd      | 44.5W+81.5    | 9.7W+17.7    | 4%  |
| both odd     | 50.5W+81.5    | 11.0W+17.7   | 4%  |
| result false |               |              |     |
| both even    | 38.5Weq+106.0 | 8.4Weq+23.0  | 3%  |
| one odd      | 44.5Weq+112.0 | 9.7Weq+24.3  | 3%  |
| both odd     | 50.5Weq+118.0 | 11.0Weq+25.6 | 3%  |

| Instruction  | Step          | μsec.         |     |
|--------------|---------------|---------------|-----|
| SETSP        |               |               |     |
| result true  |               |               |     |
| both even    | 38.5W+81.5    | 8.4W+17.7     | 4%  |
| one odd      | 44.5W+81.5    | 9.7W+17.7     | 4%  |
| both odd     | 50.5W+81.5    | 17.7W+17.7    | 4%  |
| result false |               |               |     |
| both even    | 38.5Weq+106.0 | 8.4Weq+23.0   | 3%  |
| one odd      | 44.5Weq+112.0 | 9.7Weq+24.3   | 3%  |
| both odd     | 50.5Weq+118.0 | 11.0Weq+25.6  | 3%  |
| SETST        |               |               |     |
| even address | 35.0W+75.0    | 7.6W+16.3     | 10% |
| odd address  | 41.0W+75.0    | 8.9W+16.3     | 9%  |
| SETTM        |               |               |     |
| even address | 56.5          | 12.3          | 14% |
| odd address  | 62.5          | 13.6          | 13% |
| SETUN        |               |               |     |
| both even    | 45.0W+80.5    | 9.8W+17.5     | 8%  |
| one odd      | 51.0W+80.5    | 11.1W+17.5    | 7%  |
| both odd     | 57.0W+80.5    | 12.4W+17.5    | 6%  |
| SHC          | 39.5          | 8.6           | 22% |
| SHC8         | 24.0          | 5.2           |     |
| STCEA        |               |               |     |
| result true  | 18.0B+48.0    | 3.9B+10.4     |     |
| result false | 18.0Beq+64.0  | 3.9Beq+13.9   |     |
| STNHB        | 19.0          | 4.1           | 8%  |
| STVAB        | 38.5          | 8.4           | 8%  |
| STVAD        |               |               |     |
| even address | 50.5          | 11.0          | 6%  |
| odd address  | 64.5          | 14.0          | 5%  |
| STVAF        |               |               |     |
| even address | -F-L+101.3    | -.2F-.2L+22.0 | 9%  |
| odd address  | -F-L+114.3    | -.2F-.2L+24.8 | 7%  |
| STVAW        |               |               |     |
| even address | 38.5          | 8.4           | 8%  |
| odd address  | 45.5          | 9.9           | 7%  |
| STVGB        | 30.5          | 6.6           | 5%  |
| STVGD        |               |               |     |
| even address | 42.5          | 9.2           | 4%  |
| odd address  | 56.5          | 12.3          | 3%  |
| STVGF        |               |               |     |
| even address | -F-L+93.3     | -.2F-.2L+20.2 | 8%  |
| odd address  | -F-L+106.3    | -.2F-.2L+23.1 | 6%  |

| Instruction  | Step           | $\mu$ sec.        |     |
|--------------|----------------|-------------------|-----|
| STVGW        |                |                   |     |
| even address | 30.5           | 6.6               | 5%  |
| odd address  | 37.5           | 8.1               | 4%  |
| STVIB        | 7.0R+39.8      | 1.5R+8.6          | 12% |
| STVID        |                |                   |     |
| even address | 7.0R+51.8      | 1.5R+11.2         | 9%  |
| odd address  | 7.0R+65.8      | 1.5R+14.3         | 7%  |
| STVIF        |                |                   |     |
| even address | 7.0R-F-L+102.5 | 1.5R-.2F-.2L+22.2 | 11% |
| odd address  | 7.0R-F-L+115.5 | 1.5R-.2F-.2L+25.1 | 9%  |
| STVIW        |                |                   |     |
| even address | 7.0R+39.8      | 1.5R+8.6          | 12% |
| odd address  | 7.0R+46.8      | 1.5R+10.2         | 10% |
| STVLB        | 30.5           | 6.6               | 5%  |
| STVLBS       | 29.3           | 6.4               | 10% |
| STVLD        |                |                   |     |
| even address | 42.5           | 9.2               | 4%  |
| odd address  | 56.5           | 12.3              | 3%  |
| STVLDS       |                |                   |     |
| even address | 41.3           | 9.0               | 7%  |
| odd address  | 55.3           | 12.0              | 5%  |
| STVLF        |                |                   |     |
| even address | -F-L+93.3      | -.2F-.2L+20.2     | 8%  |
| odd address  | -F-L+106.3     | -.2F-.2L+23.1     | 6%  |
| STVLFS       |                |                   |     |
| even address | -F-L+92.0      | -.2F-.2L+20.0     | 10% |
| odd address  | -F-L+105.0     | -.2F-.2L+22.8     | 8%  |
| STVLW        |                |                   |     |
| even address | 30.5           | 6.6               | 5%  |
| odd address  | 37.5           | 8.1               | 4%  |
| STVLWS       |                |                   |     |
| even address | 29.3           | 6.4               | 10% |
| odd address  | 36.3           | 7.9               | 8%  |
| STVSB        | 44.0           | 9.5               | 3%  |
| STVSD        |                |                   |     |
| even address | 57.0           | 12.4              | 3%  |
| odd address  | 71.0           | 15.4              | 2%  |
| STVSF        |                |                   |     |
| even address | -F-L+105.8     | -.2F-.2L+23.0     | 6%  |
| odd address  | -F-L+118.8     | -.2F-.2L+25.8     | 5%  |
| STVSW        |                |                   |     |
| even address | 44.0           | 9.5               | 3%  |
| odd address  | 51.0           | 11.1              | 3%  |

| Instruction  | Step       | $\mu\text{sec.}$ |    |
|--------------|------------|------------------|----|
| SUB          | 30.0       | 6.5              |    |
| SVSB0        | 38.0       | 8.2              |    |
| SVSB2-SVSB31 | 39.0       | 8.5              |    |
| SVSF0        |            |                  |    |
| even address | -F-L+99.8  | -.2F-.2L+21.7    | 5% |
| odd address  | -F-L+112.8 | -.2F-.2L+24.5    | 4% |
| SVSF2-SVSF31 |            |                  |    |
| even address | -F-L+100.8 | -.2F-.2L+21.9    | 5% |
| odd address  | -F-L+113.8 | -.2F-.2L+24.7    | 4% |
| SVSW0        |            |                  |    |
| even address | 38.0       | 8.2              |    |
| odd address  | 45.0       | 9.8              |    |
| SVSW2-SVSW31 |            |                  |    |
| even address | 39.0       | 8.5              |    |
| odd address  | 46.0       | 10.0             |    |
| TEQAD        | 43.0       | 9.3              | 2% |
| TLOCK        | 41.0       | 8.9              | 2% |
| TNILL        | 40.5       | 8.8              | 1% |
| TOPEN        | 42.0       | 9.1              | 2% |
| UADD         | 30.0       | 6.5              |    |
| UADHW        | 45.0       | 9.8              | 3% |
| UDIV         | 93.0       | 20.2             | 9% |
| ULT          | 30.5       | 6.6              | 2% |
| UMOD         | 93.0       | 20.2             | 9% |
| UMUL         | 92.0       | 20.0             | 9% |
| USUB         | 30.0       | 6.5              |    |
| XOR          | 29.0       | 6.5              |    |

A. INSTRUCTION TABLES

A.

A.1 Instructions Listed by Operation Code

A.1

For each instruction the following is given: operation code (hexadecimal), symbolic name, and number of bytes in each parameter (if any).

|             |             |               |
|-------------|-------------|---------------|
| 00: except  | 2b: ioibx   | 56: setcr     |
| 01: recl    | 2c: trape 1 | 57: settm     |
| 02: rec2    | 2d: trapr 1 | 58: mcis      |
| 03: rec3    | 2e: mxept   | 59: seteq     |
| 04: rec4    | 2f: mnoop   | 5a: setsb     |
| 05: rec5    | 30: mwst    | 5b: setsp     |
| 06: rec6    | 31: ult     | 5c: setun     |
| 07: rec7    | 32: eq      | 5d: setin     |
| 08: rec8    | 33: ne      | 5e: setdi     |
| 09: rec9    | 34: lt      | 5f: setad     |
| 0a: recl0   | 35: gt      | 60: rec0      |
| 0b: recl1   | 36: le      | 61: jmzeq 2   |
| 0c: recl2   | 37: ge      | 62: jmzne 2   |
| 0d: recl3   | 38: mwist   | 63: jmzlt 2   |
| 0e: recl4   | 39: tnill   | 64: jmzgt 2   |
| 0f: recl5   | 3a: topen   | 65: jmzle 2   |
| 10: cwait   | 3b: tlock   | 66: jmzge 2   |
| 11: csign   | 3c: tegad   | 67: jmprw 2   |
| 12: crele 1 | 3d: not     | 68: mcit      |
| 13: cllst   | 3e: except  | 69: jmpphc 4  |
| 14: cskip   | 3f: except  | 6a: jmppd     |
| 15: csens   | 40: madd    | 6b: jmcht 4   |
| 16: cwtac   | 41: msub    | 6c: intrs     |
| 17: mwtac   | 42: uadd    | 6d: index     |
| 18: mwis    | 43: usub    | 6e: inprs     |
| 19: mtime   | 44: add     | 6f: inps      |
| 1a: csell   | 45: sub     | 70: iorbbs    |
| 1b: cstop   | 46: umul    | 71: iorbb     |
| 1c: cstdr   | 47: udiv    | 72: iowbbs    |
| 1d: sched   | 48: umod    | 73: iowbb     |
| 1e: cslev   | 49: mul     | 74: iorbwc    |
| 1f: cgreg   | 4a: div     | 75: iorbw     |
| 20: mwt     | 4b: mod     | 76: iowbwc    |
| 21: iowc    | 4c: and     | 77: iowbw     |
| 22: iors    | 4d: or      | 78: mcist     |
| 23: iorw    | 4e: xor     | 79: pcald     |
| 24: ioww    | 4f: crcl6   | 7a: pcals 1 4 |
| 25: iogo    | 50: neg     | 7b: pexit 2   |
| 26: iogi    | 51: abs     | 7c: lpush     |
| 27: ionci   | 52: compl   | 7d: lpop      |
| 28: mwit    | 53: shc     | 7e: lrese     |
| 29: iocci   | 54: shc8    | 7f: llock 2   |
| 2a: iocda   | 55: except  | 80: svsb0     |

|     |        |     |     |        |       |     |         |   |   |
|-----|--------|-----|-----|--------|-------|-----|---------|---|---|
| 81: | rvsb0  |     | ac: | moveb  |       | d7: | revlf   | 2 | 1 |
| 82: | svsb2  |     | ad: | rvswl2 |       | d8: | stvsf   | 2 | 1 |
| 83: | rvsb2  |     | ae: | moveg  |       | d9: | revsf   | 2 | 1 |
| 84: | svsb4  |     | af: | revpw  |       | da: | stvaf   | 4 | 1 |
| 85: | rvsb4  |     | b0: | readw  |       | db: | revaf   | 4 | 1 |
| 86: | svsb6  |     | b1: | crput  |       | dc: | svsf28  | 1 |   |
| 87: | rvsb6  |     | b2: | stvgw  | 2     | dd: | svsf29  | 1 |   |
| 88: | mwil   |     | b3: | revgw  | 2     | de: | svsf30  | 1 |   |
| 89: | revgbs | 1   | b4: | stviw  | 1 2   | df: | svsf31  | 1 |   |
| 8a: | stvlbs | 1   | b5: | reviw  | 1 2   | e0: | reagd   | 2 |   |
| 8b: | revlbs | 1   | b6: | stvlw  | 2     | e1: | rvsd0   |   |   |
| 8c: | stnhb  | 2   | b7: | revlw  | 2     | e2: | readid  | 1 | 2 |
| 8d: | rvsbl2 |     | b8: | stvsw  | 2     | e3: | rvsd2   |   |   |
| 8e: | renpb  |     | b9: | revsw  | 2     | e4: | reald   | 2 |   |
| 8f: | renhb  | 2   | ba: | stvaw  | 4     | e5: | rvsd4   |   |   |
| 90: | readb  |     | bb: | revaw  | 4     | e6: | uadhw   | 2 |   |
| 91: | crget  |     | bc: | svsw28 |       | e6: | reasd   | 2 |   |
| 92: | stvgb  | 2   | bd: | svsw29 |       | e7: | rvsd6   |   |   |
| 93: | revgb  | 2   | be: | svsw30 |       | e8: | reaad   | 4 |   |
| 94: | stvib  | 1 2 | bf: | svsw31 |       | e8: | rechd   | 2 | 2 |
| 95: | revib  | 1 2 | c0: | svsf0  | 1     | e9: | revgds  | 1 |   |
| 96: | stvlb  | 2   | c1: | rvsf0  | 1     | ea: | stvlids | 1 |   |
| 97: | revlb  | 2   | c2: | svsf2  | 1     | eb: | revlds  | 1 |   |
| 98: | stvsb  | 2   | c3: | rvsf2  | 1     | ec: | setst   |   |   |
| 99: | revsb  | 2   | c4: | svsf4  | 1     | ed: | rvsdl2  |   |   |
| 9a: | stvab  | 4   | c5: | rvsf4  | 1     | ee: | stcea   |   |   |
| 9b: | revab  | 4   | c6: | svsf6  | 1     | ef: | revpd   |   |   |
| 9c: | svsb28 |     | c7: | rvsf6  | 1     | f0: | cwram   |   |   |
| 9d: | svsb29 |     | c8: | rechws | 1     | f1: | mbset   | 2 |   |
| 9e: | svsb30 |     | c9: | revgfs | 1 1   | f2: | stvgd   | 2 |   |
| 9f: | svsb31 |     | ca: | stvlfs | 1 1   | f3: | revgd   | 2 |   |
| a0: | svsw0  |     | cb: | revlfs | 1 1   | f4: | stvid   | 1 | 2 |
| a1: | rvsw0  |     | cc: | revsm  |       | f5: | revid   | 1 | 2 |
| a2: | svsw2  |     | cd: | rvsfl2 | 1     | f6: | stvld   | 2 |   |
| a3: | rvsw2  |     | ce: | reagds | 1     | f7: | revld   | 2 |   |
| a4: | svsw4  |     | cf: | realds | 1     | f8: | stvsd   | 2 |   |
| a5: | rvsw4  |     | d0: | crram  |       | f9: | revsd   | 2 |   |
| a6: | svsw6  |     | d1: | mbtes  | 2     | fa: | stvad   | 4 |   |
| a7: | rvsw6  |     | d2: | stvgf  | 2 1   | fb: | revad   | 4 |   |
| a8: | rechw  | 2   | d3: | revgf  | 2 1   | fc: | reard   | 2 |   |
| a9: | revgws | 1   | d4: | stvif  | 1 2 1 | fd: | reaxd   |   |   |
| aa: | stvlws | 1   | d5: | revif  | 1 2 1 | fe: | cexch   |   |   |
| ab: | revlws | 1   | d6: | stvlf  | 2 1   | ff: | except  |   |   |

A.2 Instructions Listed by Name

A.2

|        |    |        |    |        |    |
|--------|----|--------|----|--------|----|
| abs    | 51 | iorbw  | 75 | mwst   | 30 |
| add    | 44 | iorbwc | 74 | mwt    | 20 |
| and    | 4c | iors   | 22 | mwtac  | 17 |
| cexch  | fe | iorw   | 23 | mxept  | 2e |
| cgreg  | 1f | iowbb  | 73 | ne     | 33 |
| cllst  | 13 | iowbbc | 72 | neg    | 50 |
| compl  | 52 | iowbw  | 77 | not    | 3d |
| crcl6  | 4f | iowbwc | 76 | or     | 4d |
| crele  | 12 | iowc   | 21 | pcald  | 79 |
| crget  | 91 | ioww   | 24 | pcals  | 7a |
| crput  | b1 | jmcht  | 6b | pexit  | 7b |
| crram  | d0 | jmphe  | 69 | read   | e8 |
| csell  | 1a | jmppe  | 6a | readb  | 90 |
| csens  | 15 | jmprr  | 67 | readw  | b0 |
| csign  | 11 | jmzeq  | 61 | reagd  | e0 |
| cskip  | 14 | jmzge  | 66 | reagds | ce |
| cslev  | 1e | jmzgt  | 64 | read   | e2 |
| cstdr  | 1c | jmzle  | 65 | reald  | e4 |
| cstop  | 1b | jmzlt  | 63 | realds | cf |
| cwait  | 10 | jmzne  | 62 | reard  | fc |
| cwram  | f0 | le     | 36 | reasd  | e6 |
| cwtac  | 16 | llock  | 7f | reaxd  | fd |
| div    | 4a | lpop   | 7d | rec0   | 60 |
| eq     | 32 | lpush  | 7c | recl   | 01 |
| except | 00 | lrese  | 7e | rec10  | 0a |
| except | 3e | lt     | 34 | rec11  | 0b |
| except | 3f | madd   | 40 | rec12  | 0c |
| except | 55 | mbset  | f1 | rec13  | 0d |
| except | ff | mbtes  | d1 | rec14  | 0e |
| ge     | 37 | mcis   | 58 | rec15  | 0f |
| gt     | 35 | mcist  | 78 | rec2   | 02 |
| index  | 6d | mcit   | 68 | rec3   | 03 |
| inprs  | 6e | mnoop  | 2f | rec4   | 04 |
| inpss  | 6f | mod    | 4b | rec5   | 05 |
| intrs  | 6c | moveb  | ac | rec6   | 06 |
| iocci  | 29 | moveg  | ae | rec7   | 07 |
| iocda  | 2a | msub   | 41 | rec8   | 08 |
| iogi   | 26 | mtime  | 19 | rec9   | 09 |
| iogo   | 25 | mul    | 49 | rechd  | e8 |
| ioibx  | 2b | mw     | 88 | rechw  | a8 |
| ionci  | 27 | mwis   | 18 | rechws | c8 |
| iorbb  | 71 | mwist  | 38 | renhb  | 8f |
| iorbbc | 70 | mwit   | 28 | renpb  | 8e |

|        |    |        |    |        |    |
|--------|----|--------|----|--------|----|
| revab  | 9b | rvsf2  | c3 | stvsb  | 98 |
| revad  | fb | rvsf4  | c5 | stvsd  | f8 |
| revaf  | db | rvsf6  | c7 | stvsf  | d8 |
| revaw  | bb | rvsw0  | a1 | stvsw  | b8 |
| revgb  | 93 | rvsw12 | ad | sub    | 45 |
| revgbs | 89 | rvsw2  | a3 | svsb0  | 80 |
| revgd  | f3 | rvsw4  | a5 | svsb2  | 82 |
| revgds | e9 | rvsw6  | a7 | svsb28 | 9c |
| revgf  | d3 | sched  | ld | svsb29 | 9d |
| revgfs | c9 | setad  | 5f | svsb30 | 9e |
| revgw  | b3 | setcr  | 56 | svsb31 | 9f |
| revgws | a9 | setdi  | 5e | svsb4  | 84 |
| revib  | 95 | seteq  | 59 | svsb6  | 86 |
| revid  | f5 | setin  | 5d | svsf0  | c0 |
| revif  | d5 | setsb  | 5a | svsf2  | c2 |
| reviw  | b5 | setsp  | 5b | svsf28 | dc |
| revlb  | 97 | setst  | ec | svsf29 | dd |
| revlbs | 8b | settm  | 57 | svsf30 | de |
| revld  | f7 | setun  | 5c | svsf31 | df |
| revlds | eb | shc    | 53 | svsf4  | c4 |
| revlf  | d7 | shc8   | 54 | svsf6  | c6 |
| revlfs | cb | stcea  | ee | svsw0  | a0 |
| revlw  | b7 | stnhb  | 8c | svsw2  | a2 |
| revlws | ab | stvab  | 9a | svsw28 | bc |
| revpd  | ef | stvad  | fa | svsw29 | bd |
| revpw  | af | stvaf  | da | svsw30 | be |
| revsb  | 99 | stvaw  | ba | svsw31 | bf |
| revsd  | f9 | stvgb  | 92 | svsw4  | a4 |
| revsf  | d9 | stvgd  | f2 | svsw6  | a6 |
| revsm  | cc | stvgf  | d2 | teqad  | 3c |
| revsw  | b9 | stvgw  | b2 | tlock  | 3b |
| rvsb0  | 81 | stvib  | 94 | tnill  | 39 |
| rvsb12 | 8d | stvid  | f4 | topen  | 3a |
| rvsb2  | 83 | stvif  | d4 | trape  | 2c |
| rvsb4  | 85 | stviw  | b4 | trapr  | 2d |
| rvsb6  | 87 | stvlb  | 96 | uadd   | 42 |
| rvsd0  | e1 | stvlbs | 8a | uadhv  | e6 |
| rvsd12 | ed | stvld  | f6 | udiv   | 47 |
| rvsd2  | e3 | stvlds | ea | ult    | 31 |
| rvsd4  | e5 | stvlf  | d6 | umod   | 48 |
| rvsd6  | e7 | stvlfs | ca | umul   | 46 |
| rvsf0  | c1 | stvlw  | b6 | usub   | 43 |
| rvsf12 | cd | stvlws | aa | xor    | 4e |



## A.3 Instructions, Section Reference

A.3

|        |         |        |         |        |        |
|--------|---------|--------|---------|--------|--------|
| abs    | 9.1.3   | iowbb  | 10.11.1 | ne     | 9.2.11 |
| add    | 9.2.1   | iowbbc | 10.12.1 | neg    | 9.1.1  |
| and    | 9.2.6   | iowbw  | 10.13.1 | not    | 9.1.2  |
| cexch  | 6.3.2   | iowbwc | 10.14.1 | or     | 9.2.7  |
| cgreg  | 11.10.5 | iowc   | 10.1.1  | pcald  | 7.1.2  |
| cllst  | 11.3.1  | ioww   | 10.2.1  | pcals  | 7.1.1  |
| compl  | 9.1.4   | jmcht  | 8.2.1   | pexit  | 7.2.1  |
| crcl6  | 9.2.24  | jmphe  | 8.1.1   | read   | 5.2.2  |
| crele  | 11.1.2  | jmppe  | 8.1.2   | readb  | 5.4.23 |
| crget  | 11.10.1 | jmpw   | 8.1.3   | readw  | 5.4.24 |
| crput  | 11.10.2 | jmzeq  | 8.3.1   | reagd  | 5.3.4  |
| crsam  | 11.10.3 | jmzge  | 8.3.6   | reagds | 18     |
| csell  | 11.4.1  | jmzgt  | 8.3.4   | read   | 5.3.5  |
| csens  | 11.2.1  | jmzle  | 8.3.5   | reald  | 5.3.3  |
| csign  | 11.1.1  | jmzlt  | 8.3.3   | realds | 18     |
| cskip  | 11.3.2  | jmzne  | 8.3.2   | reard  | 5.3.2  |
| cslev  | 11.9.2  | le     | 9.2.14  | reasd  | 5.3.6  |
| cstdr  | 11.9.1  | llock  | 11.12.1 | reaxd  | 5.3.1  |
| cstop  | 11.5.1  | lpop   | 11.11.2 | rec0   | 18     |
| cwait  | 11.2.2  | lpush  | 11.11.1 | rec1   | 18     |
| cwram  | 11.10.4 | lrese  | 11.12.2 | rec10  | 18     |
| cwtac  | 11.2.3  | lt     | 9.2.12  | rec11  | 18     |
| div    | 9.2.4   | madd   | 9.2.22  | rec12  | 18     |
| eq     | 9.2.10  | mbset  | 11.10.7 | rec13  | 18     |
| except | 13.2    | mbtes  | 11.10.6 | rec14  | 18     |
| ge     | 9.2.15  | mcis   | 11.6.1  | rec15  | 18     |
| gt     | 9.2.13  | mcist  | 11.6.3  | rec2   | 18     |
| index  | 12.2.1  | mcit   | 11.6.2  | rec3   | 18     |
| inprs  | 12.3.1  | mnoop  | 13.1    | rec4   | 18     |
| inpss  | 12.4.1  | mod    | 9.2.5   | rec5   | 18     |
| intrs  | 12.1.1  | moveb  | 6.3.1.2 | rec6   | 18     |
| iocci  | 10.7.1  | moveg  | 6.3.1.1 | rec7   | 18     |
| iocda  | 10.9.1  | msub   | 9.2.23  | rec8   | 18     |
| iogi   | 10.6.1  | mtime  | 11.7.1  | rec9   | 18     |
| iogo   | 10.3.1  | mul    | 9.2.3   | rechd  | 5.2.2  |
| ioibx  | 10.10.1 | mwi    | 11.6.4  | rechw  | 5.2.1  |
| ionci  | 10.8.1  | mwis   | 11.6.6  | rechws | 18     |
| iorbb  | 10.15.1 | mwist  | 11.6.9  | renhb  | 5.1.2  |
| iorbbc | 10.16.1 | mwit   | 11.6.7  | renpb  | 5.1.1  |
| iorbw  | 10.17.1 | mwst   | 11.6.8  | revab  | 5.4.3  |
| iorbwc | 10.18.1 | mwt    | 11.6.5  | revad  | 5.4.5  |
| iors   | 10.4.1  | mwtac  | 11.6.10 | revaf  | 5.4.6  |
| iorw   | 10.5.1  | mxept  | 13.2    | revaw  | 5.4.4  |

KAY, T. A. / VISITING

...and the ... ..

Longman, Ed.

STATE OF NEW YORK

|        |        |        |         |        |         |
|--------|--------|--------|---------|--------|---------|
| revgb  | 5.4.11 | rvswl2 | 18      | sub    | 18      |
| revgbs | 18     | rvsw2  | 18      | svsb0  | 18      |
| revgd  | 5.4.13 | rvsw4  | 18      | svsb2  | 18      |
| revgds | 18     | rvsw6  | 18      | svsb28 | 18      |
| revgf  | 5.4.14 | sched  | 11.8.1  | svsb29 | 18      |
| revgfs | 18     | setad  | 9.3.1.2 | svsb30 | 18      |
| revgw  | 5.4.12 | setcr  | 9.3.1.1 | svsb31 | 18      |
| revgws | 18     | setdi  | 9.3.2.3 | svsb4  | 18      |
| revib  | 5.4.15 | seteq  | 9.3.3.1 | svsb6  | 18      |
| revid  | 5.4.17 | setin  | 9.3.2.2 | svsf0  | 18      |
| revif  | 5.4.18 | setsb  | 9.3.3.2 | svsf2  | 18      |
| reviw  | 5.4.16 | setsp  | 9.3.3.3 | svsf28 | 18      |
| revlb  | 5.4.7  | setst  | 6.3.5   | svsf29 | 18      |
| revlbs | 18     | settm  | 9.3.4.1 | svsf30 | 18      |
| revld  | 5.4.9  | setun  | 9.3.2.1 | svsf31 | 18      |
| revlds | 18     | shc    | 9.2.8   | svsf4  | 18      |
| revlf  | 5.4.10 | shc8   | 18      | svsf6  | 18      |
| revlfs | 18     | stcea  | 6.3.3   | svsw0  | 18      |
| revlw  | 5.4.8  | stnhb  | 6.1.1   | svsw2  | 18      |
| revlws | 18     | stvab  | 6.2.1   | svsw28 | 18      |
| revpd  | 5.4.2  | stvad  | 6.2.3   | svsw29 | 18      |
| revpw  | 5.4.1  | stvaf  | 6.2.4   | svsw30 | 18      |
| revsb  | 5.4.19 | stvaw  | 6.2.2   | svsw31 | 18      |
| revsd  | 5.4.21 | stvgb  | 6.2.9   | svsw4  | 18      |
| revsf  | 5.4.22 | stvgd  | 6.2.11  | svsw6  | 18      |
| revsm  | 6.3.4  | stvgf  | 6.2.12  | teqad  | 9.3.3.4 |
| revsw  | 5.4.20 | stvgw  | 6.2.10  | tlock  | 9.1.7   |
| rvsb0  | 18     | stvib  | 6.2.13  | tnill  | 9.1.5   |
| rvsbl2 | 18     | stvid  | 6.2.15  | topen  | 9.1.6   |
| rvsb2  | 18     | stvif  | 6.2.16  | trape  | 13.3.1  |
| rvsb4  | 18     | stviw  | 6.2.14  | trapr  | 13.3.2  |
| rvsb6  | 18     | stvlb  | 6.2.5   | uadd   | 9.2.17  |
| rvsd0  | 18     | stvlbs | 18      | uadhw  | 5.3.6   |
| rvsd12 | 18     | stvld  | 6.2.7   | udiv   | 9.2.20  |
| rvsd2  | 18     | stvlds | 18      | ult    | 9.2.9   |
| rvsd4  | 18     | stvlf  | 6.2.8   | umod   | 9.2.21  |
| rvsd6  | 18     | stvlfs | 18      | umul   | 9.2.19  |
| rvsf0  | 18     | stvlw  | 6.2.6   | usub   | 9.2.18  |
| rvsf12 | 18     | stvlws | 18      | xor    | 9.2.16  |
| rvsf2  | 18     | stvsb  | 6.2.17  |        |         |
| rvsf4  | 18     | stvsvd | 6.2.19  |        |         |
| rvsf6  | 18     | stvsvf | 6.2.20  |        |         |
| rvsw0  | 18     | stvsw  | 9.2.18  |        |         |

## RETURN LETTER

Title: RC3502 Reference Manual

RCSL No.: 52-AA972

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

 **REGNECENTRALEN**  
af 1979

Information Department  
Lautrupbjerg 1  
DK-2750 Ballerup  
Denmark