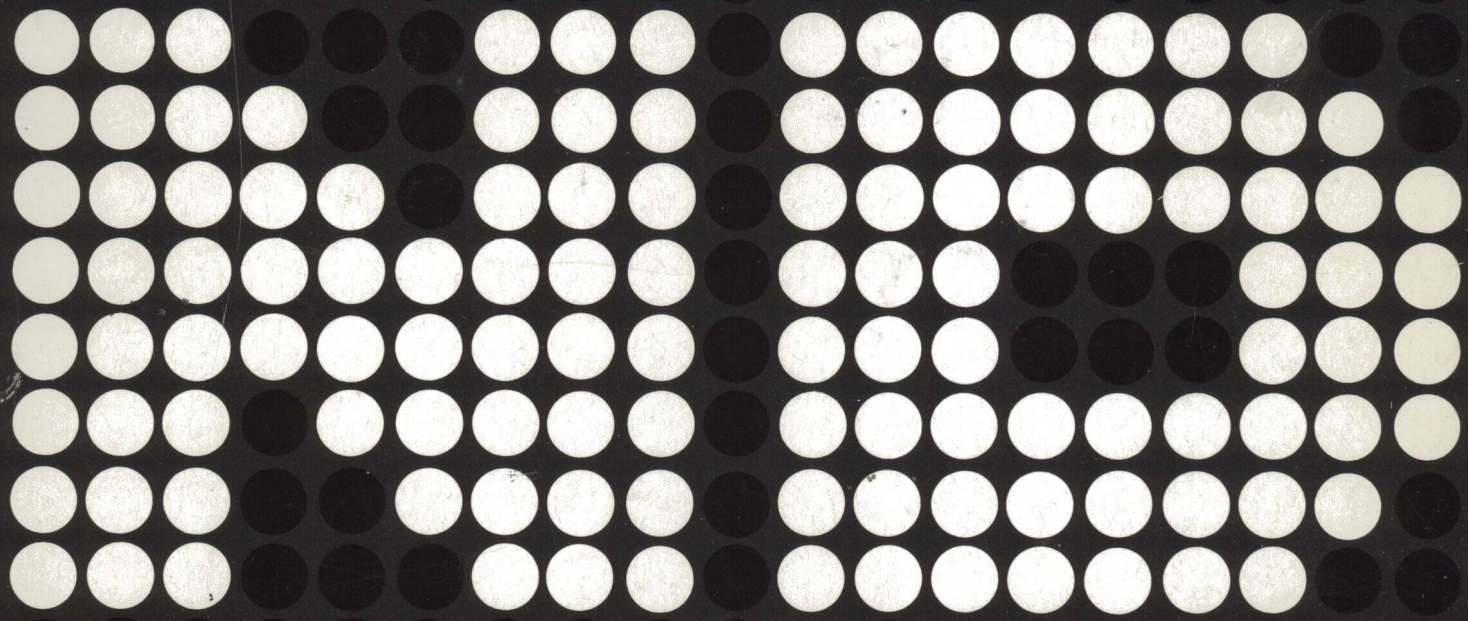
NORD-500
Reference Manual

NORSK DATA A.S



NORD-500
Reference Manual

NOTICE

The information in this document is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this document. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

The information described in this document is protected by copyright. It may not be photocopied, reproduced or translated without the prior consent of Norsk Data A.S.

Copyright © 1981 by Norsk Data A.S.

PRINTING RECORD

Printing	Notes
10/80	ORIGINAL PRINTING – Version 01

NORD–500 Reference Manual
Publication No. ND–05.009.01

NORSK DATA A.S
P.O. Box 4, Lindeberg gård
Oslo 10, Norway

Manuals can be updated in two ways, new versions and revisions. New versions consist of a complete new manual which replaces the old manual. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the ND Bulletin and can be ordered as described below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

These forms, together with all types of inquiry and requests for documentation should be sent to the local ND office or (in Norway) to:

Documentation Department
Norsk Data A.S
P.O. Box 4, Lindeberg gård
Oslo 10

PREFACE

THE PRODUCT

This manual describes the general design and the instruction set of the NORD-500 central processing unit.

THE READER

The NORD-500 CPU reference manual is intended for users of the NORD-500 system who would like to know about the general design of the NORD-500, programmers using the NORD-500 assembler, and system programmers needing to know the exact format of generated code.

PREREQUISITE KNOWLEDGE

No previous knowledge of the NORD-500 is required. However, general knowledge of computer architecture is desirable, and assembly programming experience is required for those using the manual to program in the NORD-500 assembler language. Programming the memory management system, the NORD-100/NORD-500 communication and the inner kernel of the operating system requires a more detailed description of both NORD-500 and NORD-100 hardware. This can be found in

NORD-500 Implementation manuals	- not yet available
NORD-100 Reference manual	- ND.06.014

Use of the NORD-500 assembler and how to link and load a NORD-500 program is described in the manuals

NORD-500 Assembler Reference manual	- ND.60.113
NORD-500 Monitor Loader	- ND.60.136

THE MANUAL

This manual is organized as a reference manual. It is intended for looking up the exact syntax of machine instructions and details of hardware which are relevant to the software. Most chapters are independent of the others and can be understood without reading previous chapters.

The chapters are organized as follows:

PART I General Design

Chapter 1: A general introduction to the NORD-500 system
Chapter 2: The register block
Chapter 3: Stack and heap management
Chapter 4: Memory management system
Chapter 5: Cache memory system
Chapter 6: The trap system
Chapter 7: Data types handled by the CPU
Chapter 8: Operand specifiers and addressing

PART II Instructions

Chapter 9: Instruction formats
Chapter 10: Data transfer, arithmetical and logical instructions
Chapter 11: Control instructions
Chapter 12: String instructions
Chapter 13: Miscellaneous instructions
Chapter 14: Special instructions
Chapter 15: NORD-500/NORD-100 communication

The appendices contain tables of address codes, instructions, figures and notational conventions.

Contents

1. INTRODUCTION 1

1.1. System configuration 1

1.2. Communication between the NORD-100 and NORD-500 CPUs 3

1.3. Domains, segments and processes 4

2. THE REGISTER BLOCK 5

3. STATIC DATA, STACK AND HEAP 7

3.1. Static allocation 7

3.2. Stack allocation 8

3.3. Heap allocation 10

4. MEMORY MANAGEMENT SYSTEM 12

4.1. Introduction 12

4.2. Memory management architecture 15
4.2.1. Address domain 15
4.2.2. Process 16
4.2.3. Process environment 17
3.1. Process registers 17
3.2. Capability tables 18
3.3. Domain information 19
4.2.4. Logical addressing 21
4.2.5. Domain communication 22
5.1. Alternative domain 22
5.2. Domain call 22
5.3. Trap handling 24

4.3. Physical implementation 25

4.4. Buffering 28

5. CACHE MEMORY SYSTEM 30

6. THE TRAP SYSTEM 32

6.1. Trap handler routines 32

6.2. The status register 37
6.2.1. Data status bits 37
6.2.2. Tracing status bits 39
6.2.3. Instruction and operand reference status bits 40
3.1. Ignorable trap conditions 40
3.2. Non-ignorable trap conditions 42
3.3. Fatal trap condition 42
6.2.4. Signalling, synchronization and miscellanous status bits 43
6.2.5. NORD-500 system error status bits 45
6.2.6. Addressing traps 46
6.2.7. Status bits survey 46

7. DATA TYPES 48

7.1. Introduction 48

7.2. Data types 48
7.2.1. Bit 48
7.2.2. Byte 49
7.2.3. Halfword 49
7.2.4. Word 49
7.2.5. Single precision floating point 50
7.2.6. Double precision floating point 50
7.2.7. Floating point rounding 51
7.2.8. Descriptor 52

7.3. Data formats in main memory 52

7.4. Data in registers 54

8. OPERAND SPECIFIERS AND ADDRESSING 56

8.1. Introduction 56

8.2. General and direct operands 57
8.2.1. Introduction 57
8.2.2. General operands 59
8.2.3. Post Index 61

8.3. Survey of addressing modes 62
8.3.1. Local addressing 65
8.3.2. Local, post indexed addressing 67
8.3.3. Local indirect addressing 69
8.3.4. Local indirect, post indexed addressing 71

8.3.5. Record addressing 73
8.3.6. Pre indexed addressing 75
8.3.7. Absolute addressing 77
8.3.8. Absolute, post indexed addressing 79
8.3.9. Constant operand addressing 81
8.3.10. Register addressing 83
8.3.11. Alternative addressing 84
8.3.12. Descriptor addressing 85

8.4. Direct operands 88
8.4.1. Introduction 88
8.4.2. Displacement addressing 88
8.4.3. Absolute program addressing 88
8.4.4. Absolute data addressing 88

9. THE NORD-500 INSTRUCTION SET 89

9.1. Introduction 89

10. DATA TRANSFER, ARITHMETICAL AND LOGICAL INSTRUCTIONS 94

10.1. Load 94

10.2. Load local base register 95

10.3. Load record register 96

10.4. Store 97

10.5. Store local base register 98

10.6. Store record register 99

10.7. Move 100

10.8. Swap 101

10.9. Compare 102

10.10. Compare two operands 103

10.11. Test against zero 104

10.12. Negate 105

10.13. Invert 106

10.14. Invert with carry add 107

10.15. Absolute value 108

10.16. Add 109

10.17. Subtract 110

10.18. Multiply 111

10.19. Divide 112

10.20. Add two operands 113

10.21. Subtract two operands 114

10.22. Multiply two operands 115

10.23. Divide two operands 116

10.24. Add three operands 117

10.25. Subtract three operands 118

10.26. Multiply three operands 119

10.27. Divide three operands 120

10.28. Multiply with overflow to register 121

10.29. Divide with remainder to register (modulo) 122

10.30. Unsigned multiply with overflow to register 123

10.31. Unsigned divide 124

10.32. Add with carry 125

10.33. Subtract with carry 126

10.34. Clear register 127

10.35. Store zero 128

10.36. Set to one 129

10.37. Increment 130

10.38. Decrement 131

10.39. And 132

10.40. Or 133

10.41. Exclusive or 134

10.42. Logical shift 135

10.43. Arithmetical shift 136

10.44. Rotational shift 137

10.45. Get bit 138

10.46. Put bit 139

10.47. Clear bit 140

10.48. Set bit 141

10.49. Get bit field 142

10.50. Put bit field 143

10.51. A to the I'th power 144

10.52. I to the J'th power 145

10.53. Square root 146

10.54. Polynomial 147

10.55. Floating point remainder 148

10.56. Integer part 149

10.57. Integer part with rounding 150

10.58. Multiply and add 151

10.59. Sum of products 152

10.60. Load index 153

10.61. Calculate index 154

11. CONTROL INSTRUCTIONS 156

11.1. Unconditional relative jump 156

11.2. Unconditional absolute jump 157

11.3. Conditional jump 158

11.4. Loop with increment 160

11.5. Loop with decrement 162

11.6. Loop general 164

11.7. Call subroutine general 166

11.8. Call subroutine absolute 168

11.9. Initialize stack 170

11.10. Subroutine entry points 171

11.11. Subroutine return 179

12. STRING INSTRUCTIONS 182

12.1. Introduction 182

12.2. String move 184

12.3. String move while 185

12.4. String move until 186

12.5. String move translated 187

12.6. String move translated until 188

12.7. String move n elements 189

12.8. String fill 190

12.9. String fill n elements 191

12.10. String compare 192

12.11. String compare translated 193

12.12. String compare with pad 194

12.13. String compare translated with pad 195

12.14. Skip elements 196

12.15. String locate elements 197

12.16. String scan 198

12.17. String span 199

12.18. String match 200

12.19. Set parity in string 201

12.20. Check parity in string 202

13. MISCELLANEOUS INSTRUCTIONS 203

13.1. Block move and Fill 203

13.2. Data type conversion 204

13.3. Data type conversion with rounding 206

13.4. Load address 207

13.5. Load address into record register 208

13.6. Load address into base register 209

13.7. Load address of multilevel link 210

13.8. No operation 211

13.9. Set flag 212

13.10. Clear flag 213

13.11. Get buddy element 214

13.12. Free buddy element 215

14. SPECIAL INSTRUCTIONS 216

14.1. Disable process switch 216

14.2. Enable process switch 217

14.3. Set bit in trap enable register 218

14.4. Clear bit in trap enable register 219

14.5. Break point 220

14.6. Test and set 221

14.7. Load special register 222

14.8. Store special registers 223

14.9. Integer float register communication 224

15. COMMUNICATION BETWEEN NORD-500 AND NORD-100 226

15.1. Hardware interconnection 226

APPENDIX A Address codes 231

APPENDIX B Address code table 233

APPENDIX C Symbols and abbreviations 235

APPENDIX D Figures 237

APPENDIX E Instruction table 238

APPENDIX F Alphabetical instruction table 251

APPENDIX G Instruction code table 255

APPENDIX H Instruction code cross reference table 258

APPENDIX I Setting of status bits 260

I N D E X 264

EXAMPLES USED IN THIS MANUAL

Due to the large number of instruction formats and address modes available, it is not possible to illustrate more than a small fraction of the legal combinations. An attempt has been made to show the use of each format and mode at least once.

Numeric quantities are presented in decimal, octal and/or hexadecimal format. Octal numbers are followed by a 'B', hexadecimal numbers by an 'H'. Hexadecimal numbers must always start with a decimal number to avoid confusion with identifiers (that is, FFH must be written as 0FFH). In this manual hexadecimal numbers are always preceeded by a zero.
Absence of a following letter indicates decimal number.

When reading examples containing word and halfword quantities displayed as octal bytes, the values in the upper bytes have to be shifted. Example:

Binary pattern:		00010000000010000100100101010010
Displayed as:	Four octal bytes:	020B 010B 111B 122B
	Two octal halfwords:	010010B 044522B
	Octal word:	02002044522B

Hexadecimal numbers require no shifting; the hexadecimal digits can be concatenated as they are, two digits per byte.

In the figures, address values increase downwards.

1. INTRODUCTION

1.1. System configuration

The NORD-500 central processing unit is a part of the NORD-500 computer system. This system is a combination of a NORD-100 CPU, a NORD-500 CPU and a shared memory, see figure 1.1.

NORD-100 CPU

- Supervises the NORD-500 CPU
- Runs the I/O system, file system, operating system and job scheduling
- Runs local NORD-100 jobs

NORD-500 CPU

- 32-bit logical address space
- User jobs up to 4 gigabytes in size
- Addressing system implemented twice by the memory management system to allow user programs of 4 gigabytes of instructions and 4 gigabytes of data
- CPU shared by many user programs through efficient use of the memory management system
- Operations on data units ranging from 1 to 64 bits
- Byte oriented instructions designed for execution efficiency of high level language programs
- Cache memory employing a forward fetch mechanism for main memory access
- Main memory access up to 16 bytes wide, eliminating the memory bandwidth bottleneck
- Two independent but identical cache systems, one for instructions and one for data
- The cache may be partitioned, each partition used either as cache memory or as high speed local memory
- The majority of machine level instructions requiring only one basic cycle

- Asynchronous floating point arithmetic for increased instruction execution speed
- Instruction and data pipelining techniques employed to optimize execution speeds
- Specialized high speed hardware for 32/64 bit floating point multiply and divide

MEMORY

- Multiport main memory with direct access for the NORD-500 CPU, NORD-100 CPU, and DMA transfer devices
- Physical main memory up to 32 Mbytes
- Virtual memory management system
- Memory fully or partially shared between NORD-100 and NORD-500

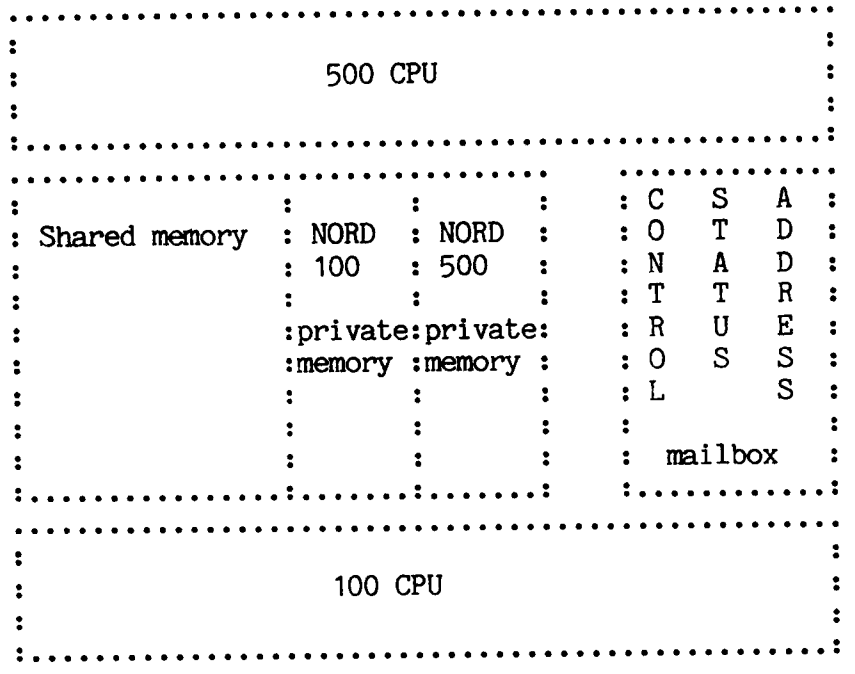


Fig. 1.1 The NORD-500 computer system

1.2. Communication between the NORD-100 and NORD-500 CPUs

All or part of the memory can be shared between the NORD-500 CPU, NORD-100 CPU and associated I/O devices. This allows for easy access and control by all components of the system.

The communication between the NORD-100 and NORD-500 is set up as a mailbox and DMA transfer system. The mailbox contains 3 registers:

Control register: For NORD-100 to give NORD-500 a command

Status register : For NORD-500 to give NORD-100 status

Address register: A pointer to where in the NORD-100 memory chains of instruction or data will be found, or where the NORD-500 can store extended status information

The status information returned to NORD-100 reports that a job is finished, the reason for NORD-500 termination and type of possible NORD-500 malfunctions.

The NORD-500 microprogram initiates and controls the DMA access channel to NORD-100 memory. The communication channel is also used extensively for diagnostic and test program information. The NORD-100 is used as a diagnostic vehicle for the NORD-500.

1.3. Domains, segments and processes

In the NORD-500 memory is logically structured into DOMAINS. A domain is one 32 bit address area (4 gigabytes) for executable code (the program domain), another for data (the data domain).

Each domain is divided into SEGMENTS, up to 31 per domain. A segment can be up to 128 Mbytes, equivalent to 27 address bits. The smallest unit for access protection (write and parameter access protection) is a segment.

Data may be accessed from the entire domain, but if the segment number (the upper five address bits) is zero, data will be taken from the data segment with the same segment number as the current program segment, thus making data references independent of the actual segment number.

Two (or more) domains may have segments in common, in order to share data. The segment number(s) does not have to be the same in the two domains.

A sequence of operations requiring no parallel execution is called a PROCESS. A process is carried out sequentially in the CPU, but several processes started at different times may run concurrently.

A process may refer to up to 256 domains of data and instructions. These are connected in a tree stucture called a domain tree, specified by the process description kept by the memory management system. The links between the domains are determined at the creation of each domain. The domain closest above (that is, closer to the root) a domain D is the mother of D, D is the child. D may itself be the mother of other child domains.

Control can be switched from one domain to another by calling a routine in the other domain, or by causing an error situation (trap condition) not taken care of by a routine in the current domain. A routine may access data in the domain from which it was called, through an address prefix (ALT).

Within a segment routines are called directly, by address. Routines on other segments are called through their routine number on the segment, not by address.

Communication between processes is possible through monitor calls or through a shared data segment.

2. THE REGISTER BLOCK

The NORD-500 CPU has four registers for program and data addressing. These are the program counter P, the L (link) register containing the subroutine return address, the local variable base register B, and the record base register R.

The four 32-bit general registers, I1, I2, I3, and I4, may be used as integer accumulators or as index registers. They are used for both word and partial word operations (halfword, byte, bit and bit field).

The A1, A2, A3, and A4 registers are 32-bit floating point accumulators used for real number arithmetic. Each floating point accumulator may be extended with a 32-bit Extension register (E1, E2, E3 and E4), making four 64-bit floating point accumulators for double precision arithmetic.

The NORD-500 also has several special purpose registers. These are the 64-bit registers:

STatus register	- ST
Own Trap Enable register	- OTE
Child Trap Enable register	- CTE
Mother Trap Enable register	- MTE
Trap Enable Modification Mask	- TEMM

and the 32-bit registers:

Top Of Stack register	- TOS
Low Limit trap register	- LL
High Limit trap register	- HL
Trap Handler Address register	- THA

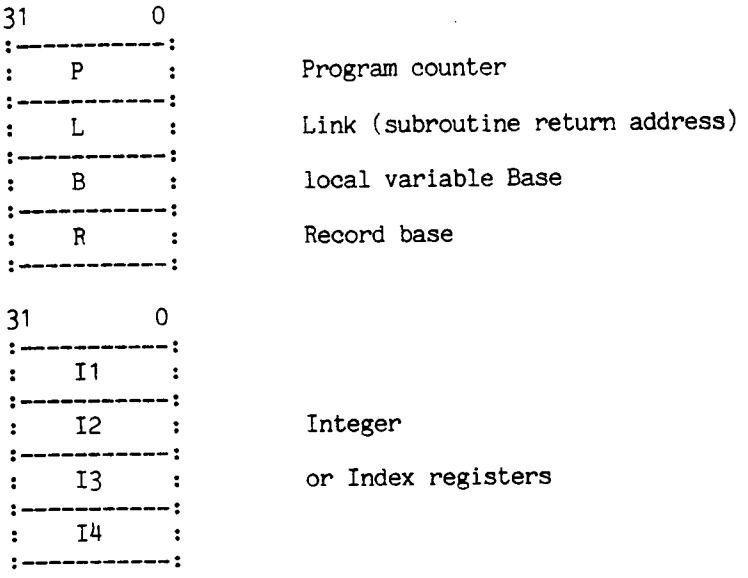
The ST, OTE, CTE, MTE and TEMM registers are treated as two 32-bit registers when referenced in instructions. The least significant parts (bits 0-31) are called ST1, OTE1, CTE1, MTE1 and TEMM1. The most significant parts (bits 32-63) are called ST2, OTE2, CTE2, MTE2 and TEMM2.

The memory management system utilizes a number of registers accessible only to the microprogram. These include

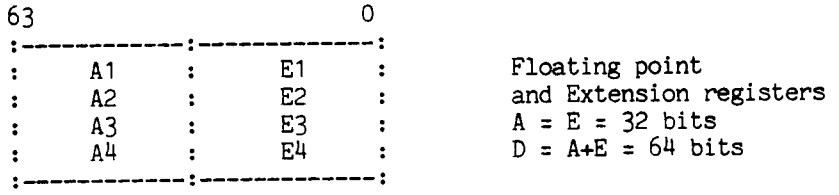
Current Executing Segment register	- CES
Current Executing Domain register	- CED
Current Alternative Domain register	- CAD
Process Segment register	- PS
Physical Segment Table Pointer	- PSTP

The CES, CED, CAD and PS registers exist as one copy per process in the system, while the PSTP is unique.

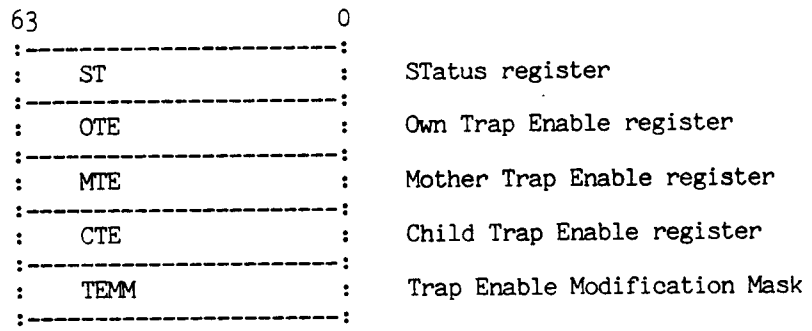
THE REGISTER BLOCK



The In accumulators are named BIn, BYn, Hn, and Wn when used for BIt, BYte, Halfword, or Word operations. (n=1,2,3,4).



The An accumulators are named Fn when used as single precision floating point registers. The (An, En) double registers are named Dn when used as double precision floating point registers.



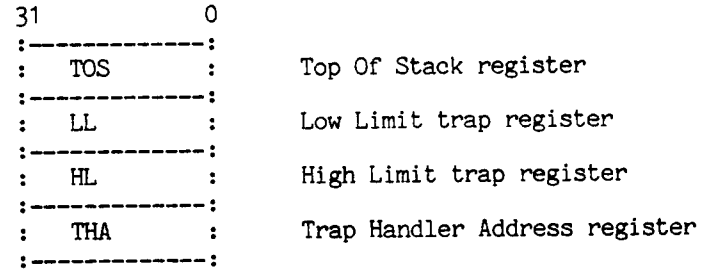


Fig. 2.1 THE REGISTER BLOCK

3. STATIC DATA, STACK AND HEAP

Space for data objects may be allocated

i) in a fixed location in memory, referenced relative to the B register or by absolute address (static allocation)

ii) on a stack growing from low to high memory, referenced relative to the B register

iii) in a block unlinked from a freelist, anywhere in otherwise unused memory, referenced relative to the B register.

Static or dynamic allocation of local data area is determined by the kind of entry point instruction, and a program system may contain a mixture of procedures with statically and dynamically allocated data areas. In most cases the calling procedure need not be concerned with the allocation strategy used.

3.1. Static allocation

Data allocated in fixed locations may be addressed by a full 32-bit address referencing any segment within the domain, or it may have a segment number of zero, indicating that the segment number of the currently executing program segment is to be used. The latter case allows a segment with its own fixed data area to be part of more than one domain, having different segment numbers in different domains, with no need to relocate the addresses. Statically allocated data may not be released for other use, and local variables in procedures keep their values from one call to the next.

Procedures with static data areas are entered through an ENTF or ENTFN instruction. Such routines are by definition non-reentrant, but in other respects behave as other routines. The fixed local data area is initialized as shown in figure 3.1. The B register is updated to point to the local data area and data references may be addressed relative to the B register, as with stack routines, and may also be addressed directly.

Trap handlers always have a fixed local data area which has a special layout discussed in chapter 6.

3.2. Stack allocation

A stack is initialized through the INIT or ENTM instruction, either one declaring the lowest stack address and its maximum extent. When a stack is initialized, the TOS register is loaded with the address of the first free location above the stack's maximum extent. TOS serves as a guard trapping eg. a "wild" recursive routine and as a pointer to the variables describing the heap.

A new data block on the stack is allocated by executing an ENTS or ENTSN instruction. On routine entry the data block is always initialized by the system as follows:

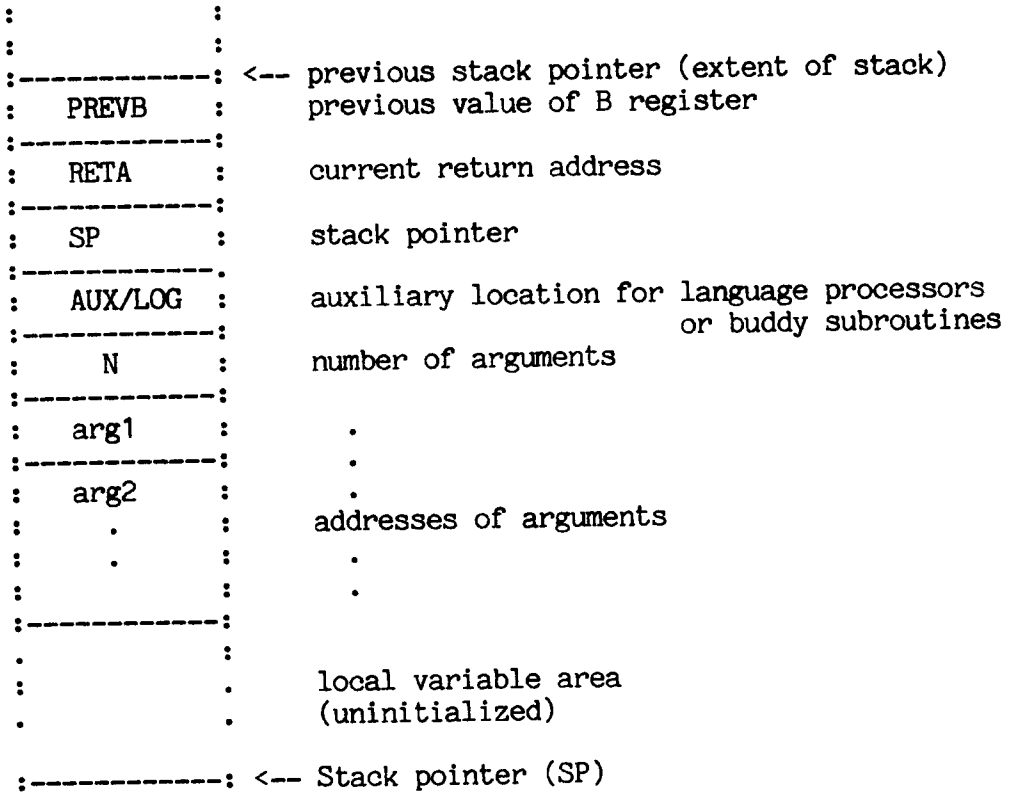


Fig. 3.1 Local data area layout

If the number of arguments supplied exceeds the maximum allowed by the ENTSN entry point instruction, the N location will contain the number of arguments actually supplied, but only the maximum allowed number of argument addresses will be put on the stack. (This also applies to the ENTFN instruction.)

The INIT instruction initializes the stack in a similar way, but the PREVB and RETA will be zeroed, so that an attempt to link downwards beyond the lower stack address will cause an Address Zero trap.

The ENTM instruction initializes a new stack starting from a specified address, giving the TOS register a new value. The old TOS value is

saved on the current top of the old stack, pointed to by B.SP. Otherwise, the initialization is as for a routine entry, with the base address of the previous stack block saved in PREVB. The ENTM is typically used for initializing a stack for the routines on a segment, being called from other segments in the same domain. Executing the same ENTM instruction twice will overwrite the old initial values, possibly destroying return address and other information.

Stack space is released through the RET or RETK instructions. The B register is loaded from the PREVB location. On exit from a module (a subroutine entered through ENTM) the TOS register is not updated; this must be done explicitly.

Stack displacements (relative to the B register) are always positive, the displacement being the number of bytes to add to the B register. PREVB, RETA, SP, AUX and N are predefined as 0, 4, 8, 12 and 16, respectively.

3.3. Heap allocation

When running several routines concurrently, stack allocation of local data areas will cause problems if the routine finishing first is not the one with its data area on top of the stack.

Complex data structures like trees, lists and networks, may grow and shrink dynamically and elements acquired during the execution of a procedure should not be released upon exit.

For both these uses data elements may be allocated from a pool of unreserved space called the heap. The heap is described by an array of list heads to linked lists of free elements, one list per block size. The block size is always a power of two and is indicated by the logarithm to the base two (the "log size") of the number of words. The first word of an element contains the address of the next element in the list, zero indicating the end of the list.

The first location above the stack, pointed to by the TOS register, contains the maximum size of elements to be allocated. The next two locations are reserved for the lower and upper address limits of the pool, respectively. Above these two locations is the array of head pointers.

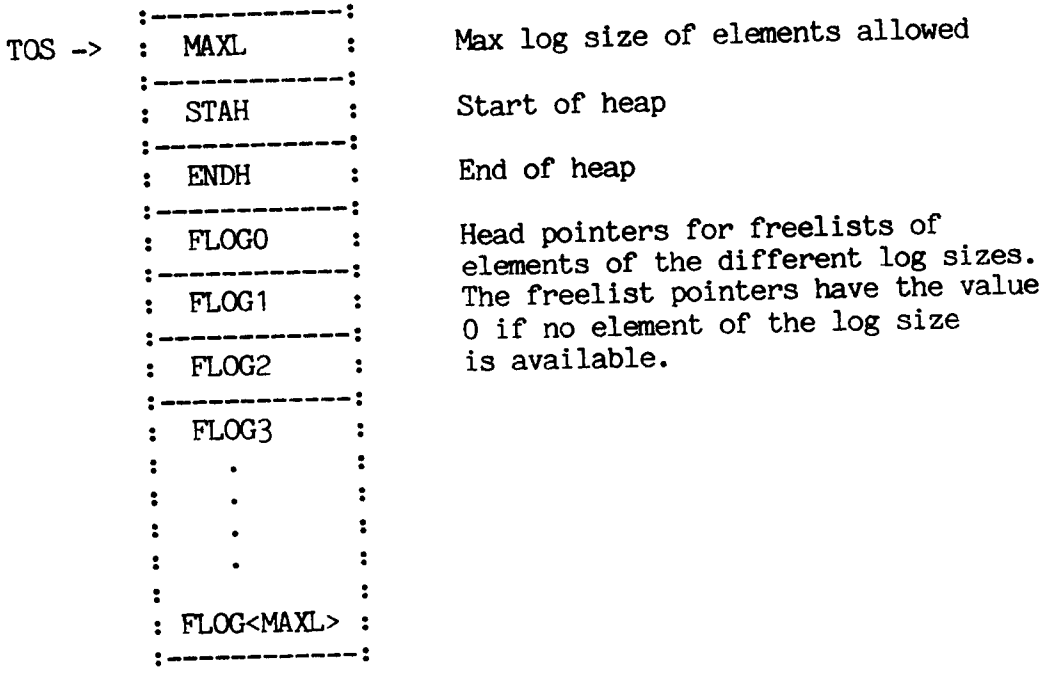


Fig. 3.2 Layout of heap variables

The heap variables must be initialized by the user program and the user is responsible for building the lists. The STAH and ENDH variables are not used by the heap instructions, but are available for a heap administration routine implemented as a trap handler for the stack overflow trap.

STATIC DATA, STACK AND HEAP

A local area for use by a subroutine may be allocated by executing the ENTB instruction. This contains an indication of the required block size. On routine entry, the address of the allocated block is loaded into the B register, and the block size is stored in the AUX/LOG location. In all other respects the local data area is initialized as for a stack routine.

A data element is allocated by the GETB instruction, specifying the size of the desired element. The address of the element is loaded into the specified register.

If a block of the requested size is available, it is unlinked from the list. If the list head is zero, indicating that the list is empty, lists representing larger blocks are examined. If a larger block is available, it is split in halves and one half left in the appropriate freelist. The block may have to be split several times before an element of the requested size can be granted to the program. If no larger element is available, or if the requested size is larger than the MAXL value, a stack overflow trap condition occurs.

A routine entered through ENTB may release its local data area by returning through the RETB or RETBK instruction. An element aquired by the GETB may be released by the FREEB instruction.

A released element will be linked to the appropriate freelist according to the size of the element. Elements are not combined; this may be done by the trap handler for the stack overflow condition.

Be aware that initializing a new stack by INIT or ENTM will change TOS, thus another set of heap variables will be used by the buddy instructions. The new heap variables may be initialized to the values of the old ones or to new values.

No assumptions should be made about initial values of locations of stack or heap elements not explicitly mentioned here.

4. MEMORY MANAGEMENT SYSTEM

4.1. Introduction

A process is a sequential computation in NORD-500 that may refer to up to 256 domains. Each domain is a full 32 bit address area for program instructions and another for data. A process may easily access two such data domains, the socalled Current Executing Domain (CED) and the Current Alternative Domain (CAD). Instructions will always be fetched from CED, but data will be taken from CAD when the address code prefix ALT is used. If ALT is omitted data accesses will be done in CED.

Each domain is divided into 32 logical segments each of 27 address bits. 31 of these are used, segment number 0 is never used. A 27 bit logical segment address is translated by the memory management system so that it addresses a location in a socalled physical segment. Physical segments contain the data and programs in the NORD-500. A physical segment is divided into blocks of 2k bytes, and may have any size from 2**11 to 2**27 bytes. The blocks of 2k bytes are called pages, and they can be moved (swapped) between main memory and secondary storage as the need arises.

All physical segments in a NORD-500 system is described in the Physical Segment Table (PST). The PST is always resident in main memory and it is used by the translation mechanism to find the physical segment. If a physical segment consists of more than one page, an indexing mechanism is used to address the segment. Each physical segment is described by a 16 bit entry in PST.

By following this scheme each process in NORD-500 may use up to 256*31 physical segments of program, and an equal number of physical segments of data. The structure and properties of the domains and segments of a process are kept on a special physical segment generated and maintained by supervising mechanisms. This physical segment is called the Process Segment (PS). There is one PS for each process in the NORD-500. The size of a PS will depend on the number of domains the process can use.

The PS of a process cannot be accessed directly by the process itself. It is used by supervising mechanisms which may be other processes, other domains or NORD-100. Each domain used by a process has one entry in the PS of a process.

One entry in the process segment is called the domain information table. A domain information table contains 31 pointers for data (the data capability table) and 31 pointers for program (the program capability table), one pointer for each logical segment of the domain. The pointers indicate the PST entry describing the physical segment to be addressed by the logical address of the domain. Information on legal access modes for each logical segment is also kept in the domain information table, together with the pointers. One PST pointer with the corresponding legal access mode indicators is called a capability. Also located in the domain information table is the neccessary

information for the trap and domain call system.

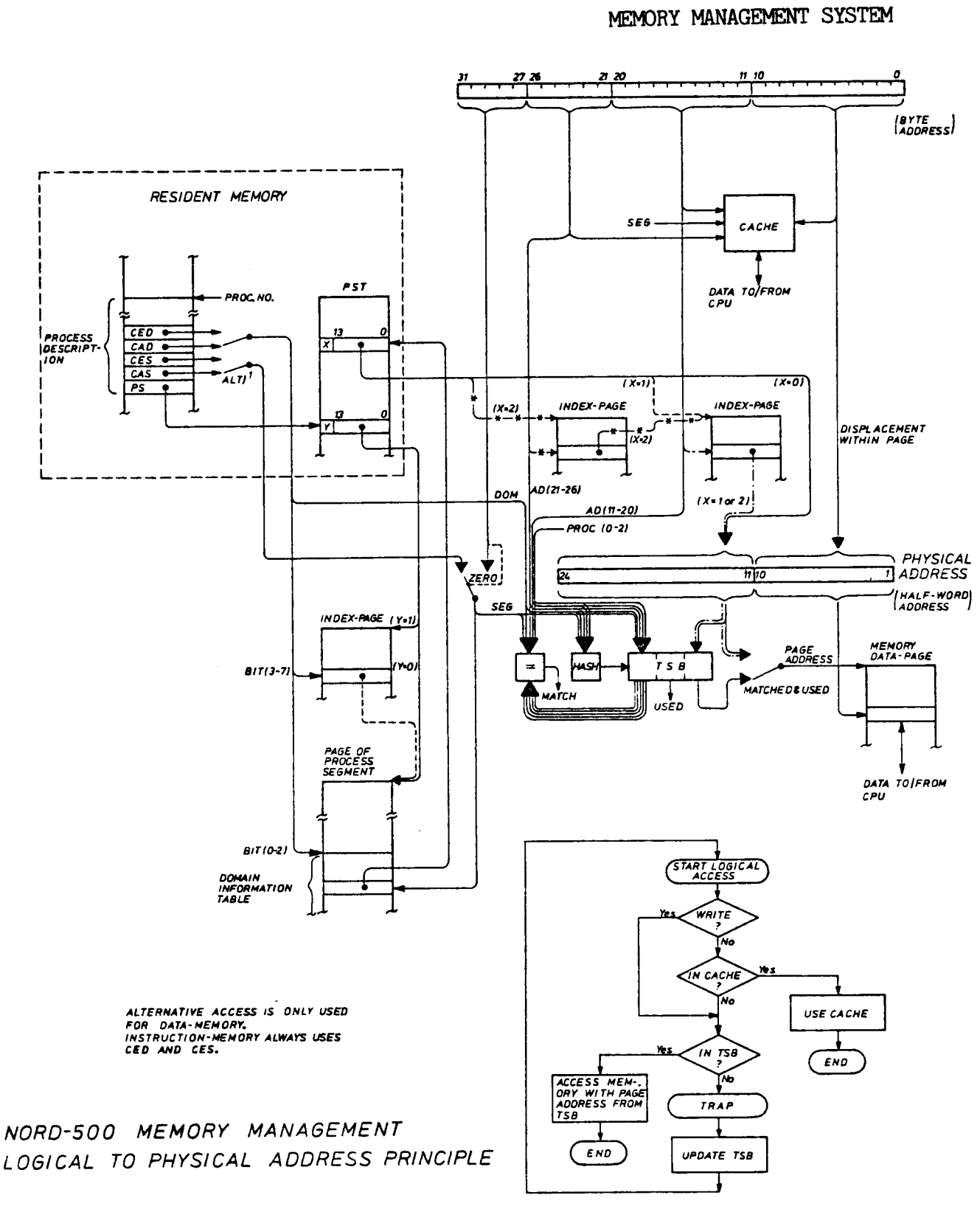
The PS of a process will be referenced frequently when the process executes. Since the PS is an ordinary physical segment, it will be addressed through the PST entry that describes it. A pointer to the PST entry describing the PS of the executing process is kept in the PS register and is updated when a new NORD-500 process starts execution. The PS register is part of the process description of a process, together with the contents of the register block and some other information.

A schematic exposition of the translation mechanisms is found in the drawing next page.

This scheme for the translation from logical to physical addressing makes it easy for different domains or processes to share data or programs. Sharing is done by having the capabilities in the different domain information tables point to the same PST entry. Thereby the same physical segment will be addressed.

If the translation mechanism were to perform all the outlined table lookups on each memory access, the result would be unacceptably slow. A speed-up mechanism is therefore introduced. Whenever an access is completed, the number of the referenced page is stored in a cache-like Translation Speedup Buffer (TSB). The physical page number is stored together with the corresponding logical page number, the domain number and process number. Next time an access to the same logical page is done by the same domain, the physical page number is found in TSB without any need to perform other lookups. The index in the TSB is found by using a hashing algorithm that takes into account the logical address including the segment number, the domain number and the process number.

The detailed description that follows is divided into the Memory Management Architecture and its Physical Implementation. The architecture section involves the transformation from logical to physical segment numbers, and includes descriptions of the capability tables and the process segment. The implementation section covers the mechanisms by which physical segments are placed and accessed in main memory. The present architecture is implemented with a paging mechanism, but no inhereht property of the architecture prohibits other implementation strategies.



NORD-500 MEMORY MANAGEMENT
LOGICAL TO PHYSICAL ADDRESS PRINCIPLE

Fig. 4.1 Logical addressing scheme

4.2. Memory management architecture

4.2.1. Address domain

A NORD-500 address has 32 bits, ie. an address is in the range from 0 to (2**32)-1. Instruction fetches and data references goes to different addressing areas; if it is an instruction fetch, the address value range is called a program domain, if it is a data reference, it is called a data domain.

A logical address domain is divided into 32 segments. The 5 upper bits of an address are segment numbers and the 27 lower bits are the address within the segment.

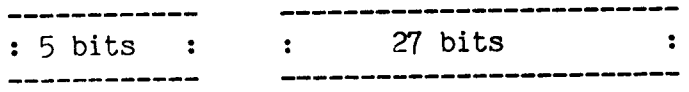


Fig. 4.2 Logical address

If the program or data domain is not explicitly stated, domain is understood to be both the program domain and its corresponding data domain.

4.2.2. Process

The operations of a computation must be carried out in a certain order to ensure a meaningful result. The simplest possible rule is the execution of operations one at a time in strict sequential order. This type of computation is called a process.

Information about a process is kept in the process description. The term process will hereafter mean a sequential computation described by a process description.

A NORD-500 process may have up to 256 different logical domains, each comprising an address space up to 2**32 bytes for each of program and data.

From a domain it is possible to create and call new program domains, ie. a NORD-500 process may have a hierarchy of domains. The hierarchical sturcture is reflected in the process description.

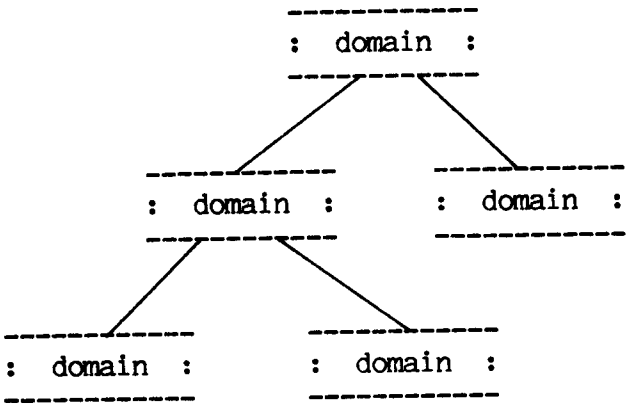


Fig. 4.3 Hierarchy of program domains

Transfer of control between domains may takes place by routine calls (domain calls) or enabled traps. Parameter transfer between different domains is performed by the alternative address mode. (See section about addressing modes.) When a routine in domain A calls a routine in domain B, domain A is set as alternative domain to B and operands accessed via alternative address mode are accessed in domain A.

More extensive data exchanges and exchanges between arbitrary domains are done by letting the domains have one or more data segments in common.

4.2.3. Process environment

The memory management system needs information about existing processes. This information resides on a physical segment, the Process Segment. This segment is not directly accessible to the process, but is used by supervising mechanisms, which may be other processes, other domains or NORD-100, and by microcode routines. There is one process segment for each process; the number of this segment is held in the Process Segment register (PS). For each domain owned by the process the process segment contains one domain information table which consists of

- the program capability table
- the data capability table
- domain call information
- trap handling information

4.2.3.1. Process registers

CED	Current Executing Domain
CAD	Current Alternative Domain
CES	Current Executing Segment
CAS	Current Segment on Alternative domain
PS	Process Segment

Fig. 4.4 Memory management registers

Some information about a process is used so frequently by the memory management system that it must be kept in hardware registers while the process is executing. The five registers CED, CAD, CES, CAS and PS are a part of the process description of the running process. Ie. the registers' contents are exchanged when exchanging process.

The Current Executing Domain register holds the program domain number of the current executing process. When a domain call is performed, or when a trap condition is not own but mother enabled, the domain number of the calling domain is stored in the Current Alternative Domain register. The Current Executing Segment register holds the segment number within the current executing domain, and is copied into CAS when changing domain. CAD is used with the alternative addressing mode, CAS is used when the segment number is zero in alternative addressing.

4.2.3.2. Capability tables

Each domain has two capability tables, one for instructions and one for data. Each table has 31 elements, one for each segment in the domain. Each element consists of 16 bits, numbered from 0 to 15. Such an element is called a capability, and it specifies the physical segment number and its access rights. A program capability has a layout different from a data capability.

In a program capability, bit 15 indicates whether the segment is in the current domain or not. If the bit is zero the segment is in the current domain. A segment not in the current domain, called an indirect segment, has bit 14 set if the physical segment resides in another machine, otherwise it is reset. The capability of an indirect segment contains the logical domain and segment numbers of another segment, and the physical segment number is found in the capability of that segment.

In a data capability bit 15 indicates write permission. If this bit is reset the segment is a read-only segment. Bit 14 indicates whether routines in other domains may refer to this segment through the ALT prefix. Violation of the protection set by these two bits causes a protect violation trap. Bit 13 is set if the physical segment is shared between different domains or different processes. If a segment is shared, data will always be read from main memory rather than from cache, to ensure that different processes are aware of each others' updating of a data item.

Direct program segments and data segments contain the physical segment number in the lower 12 bits.

Program segment capability:

a) Direct segment

: 1 bit :	: 3 bits :	: 12 bits :
direct (=0)	unused	physical segment number

b) Indirect segment

: 1 bit :	: 1 bit :	: 1 bit :	: 8 bits :	: 5 bits :
indirect (=1)	other machine	unused	domain	segment

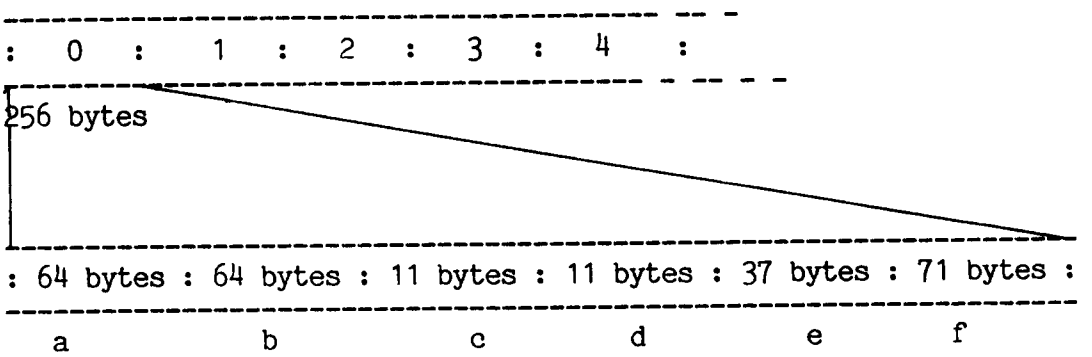
Data segment capability:

: 1 bit :	: 1 bit :	: 1 bit :	: 1 bit :	: 12 bits :
write permitted	parameter access	shared segment	unused	physical segment number

Fig. 4.5 Capability layout

4.2.3.3. Domain information

When performing domain calls and trap handling, some extra table space is needed for each domain. The first eleven bytes of a domain information table are used as a save area at domain calls. The next eleven bytes are used as a save area in trap handling. The last 37 bytes are the domain characteristics. This information and two capability tables constitute each domain information table of 256 bytes. This is shown in the following figure.



The capital letters used in the table below have the following meaning:

M - set by hardware at domain call
T - set by hardware at trap handling
O - set by operating system and read by hardware

Domain information table layout:

		bytes	
a. Program capability table	domain 1		
b. Data capability table	domain 1		
c. Domain call information	domain 1		
Calling domain		1	M
Alternative of calling domain		1	M
Current segment on alternative domain		1	M
P of calling domain		4	M
B of calling domain		4	M
d. Trap handling information	domain 1		
Trapped domain		1	T
Alternative of trapped domain		1	T
Current segment on alternative domain		1	M
Status register save area		8	T
e. Domain characteristics	domain 1		
Own trap enable		8	O
Child trap enable		8	O
Mother trap enable		8	O
Trap enable modification mask		8	O
Trap handler address		4	O
Mother domain		1	O
f. Unused			

Fig. 4.6 Domain information table

4.2.4. Logical addressing

A logical address consists of the segment number and the segment relative address. The memory management system will transform the logical segment number to a physical segment number. The segment relative address is relative to the start of the physical segment.

If the segment number in a program or data reference is zero, the number in the Current Executing Segment register is taken as the logical segment number. Where addresses are transferred or loaded the current segment will be inserted in the address automatically; this applies to the instructions CALL, CALLG, LADDR, BLADDR, RLADDR, INIT and ENTM instructions. If the data segment number is not zero it will be used without modification.

If the segment number is zero when alternative addressing is used, the segment number in the CAS register is inserted.

The logical segment number is used as an index in the capability table. The addressed element in this table gives the physical segment number.

A jump to another program segment is only performed legally by a CALL or CALLG instruction and implies that the new segment number is loaded into the Current Executing Segment.

When a legal call to another segment is performed, the segment relative address is taken as an index in a start address vector first on the new segment. The first word on a segment is the length of the start address vector. The index is compared against this word. If the index is greater it is an illegal call and causes an instruction sequence error trap condition.

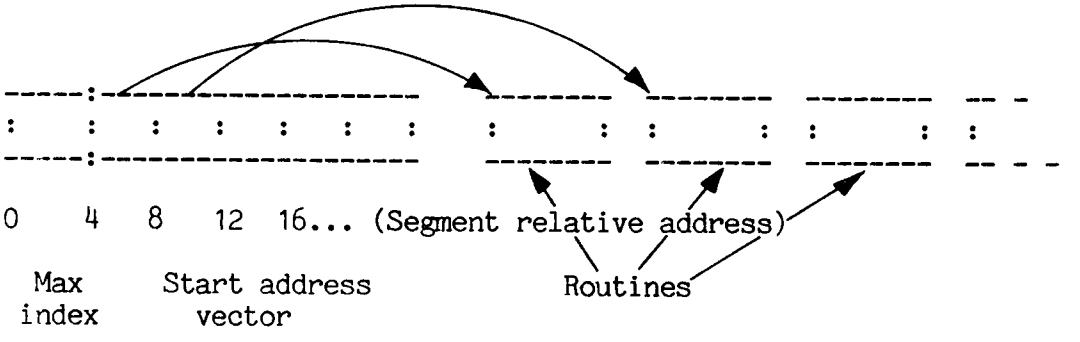


Fig. 4.7 Program segment layout

4.2.5. Domain communication

Within the domain hierarchy of a NORD-500 process program control may change from one domain to another. Data may be accessed in either the called or the calling domain. In this section change of control and communication between different domains are described.

4.2.5.1. Alternative domain

The alternative domain is used when accessing and returning parameters from or to a calling domain. The calling domain is set as the alternative to the called domain by loading its number into the CAD register. This is done by hardware at a domain call. Access to operands in the alternative domain is by the alternative address code prefix, ALT(<operand>). When using the ALT address code prefix only the last data access goes to the alternative domain; indirect addresses and descriptors are taken from the current domain. (See the chapter on operand specifiers and addressing modes for further explanation.) The calling domain may protect its data from illegal access from other domains by resetting the parameter access bit of its capability. This is done through monitor calls.

4.2.5.2. Domain call

From one domain, a routine on any other domain may be called through the CALL and CALLG instructions if an indirect capability to that domain is set up, indicated by bit 15 set in the capability of the segment. An indirect capability is set up through monitor calls. An indirect segment resides in another than the current one. A call to a routine on such a segment implies change of domain, and is denoted a domain call. Domain calls to supervising domain routines performing specific functions are called monitor calls.

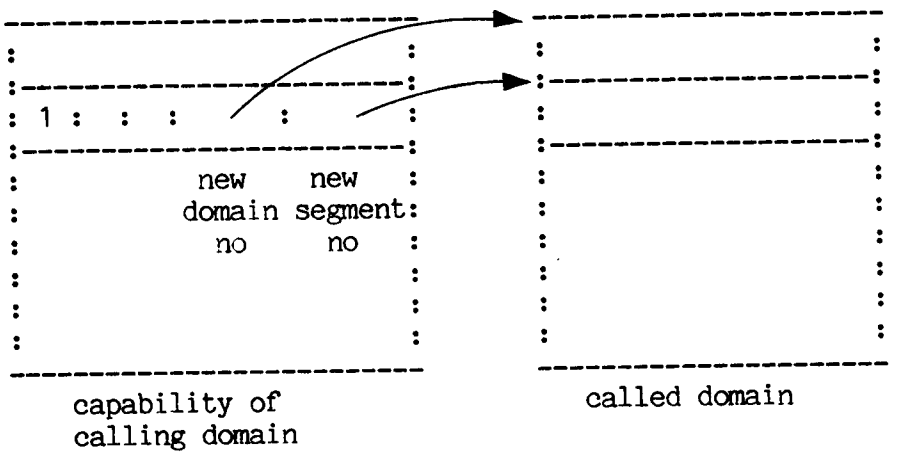


Fig. 4.8 Indirect segment

The new domain and segment number are taken from the capability of the calling segment. The program counter, base register, domain number, alternative domain number and current segment on the alternative domain of the calling one are saved in the domain information table of the called domain. When a subroutine is called, certain initializations of the local data field are made. (See the CALL, CALLG and entry point instructions.) The return address and old base register field of the local data field of the new routine are filled with zero.

The new domain and segment number are loaded into the Current Executing Domain and Current Executing Segment registers. The number of the calling domain is loaded into the alternative domain register, and the current segment on the alternative domain is loaded into the Current Segment Alternative domain register. This segment number is used whenever an access to an alternative domain is done with the segment number equal to zero.

The segment relative part of the new program address is used as the index in a start address vector in the same way as when calling a routine on another segment within a domain.

On jump to another domain, a new stack has to be set up in the called domain. Therefore, the subroutine address must be the address of an ENTM, ENTF or ENTFN instruction. If a routine with a fixed data area calls routines using stack space, the stack must be initialized prior to the call. A routine being called from another domain must not be entered through an ENTS, ENTSN, ENTB or ENTD entry point, as the stack would then not be properly initialized.

The control is changed back to the calling domain when the return address, the old base register or both is zero when a return instruction is executed. On return from a domain call the CED, CAD, CAS, P and B are loaded from the domain information table.

Note that return information is not stacked, ie. calling the same domain twice without return in between will cause an instruction sequence error trap condition. The memory management system will zero fill the return address and B register value at a domain call return to indicate that a call to the domain may be done. If the return information is non-zero a domain call is in progress and another domain call to the same domain will be trapped as illegal (Instruction Sequence Error).

A return instruction with 0 in PREVB or RETA will only change domain if there is a domain to return to. If CAD is unequal to CED and non-zero, return is to the domain saved in the domain information table. Otherwise the return will be performed to address 0 on the current domain, causing a stack underflow trap condition.

4.2.5.3. Trap handling

When a trap condition occurs, the procedure described in chapter 6 on traps will determine if a trap handler routine is to be called, and in that case which domain has a handler for the offending trap. If the trap is handled by a mother domain, the new domain number is loaded into the CED register. The old CED, CAD and CAS are saved into the domain information table of the mother domain. CAD is loaded with CED of the trapping domain.

The status register is saved into the domain information table of the trapped domain, and upon return the non-ignorable bits are reloaded.

When the system trap handler returns, the new trap enable register contents are taken from the domain information table of the trapped domain.

Trap handler startup and stack initializations take place in the same way as when invoking a local trap handler. See chapter 6 for further explanation. The new trap enable register contents are taken from the domain information table of the mother domain, except that OTE is cleared by hardware at the ENTT instruction and restored when a RETT is executed.

4.3. Physical implementation

The physical NORD-500 memory is divided into pages of size 2048 bytes. The physical main memory size may be up to 2**25 bytes. The page size of 2048=2**11 implies 2**14 pages, or a 14 bits page number.

The memory management system has a bit map with two bits per physical page, set if the page is used and if the page has been written to, respectively. If the page has been written to it must be copied back to mass storage before it is replaced with another one. The table size is 2*(2**14) bits, and it is accessible to microcode only.

The memory management system maintains a Physical Segment Table Pointer (PSTP) pointing to the start of the Physical Segment Table. This table contains a two byte entry for each physical segment, giving the page number of a data page or an index page.

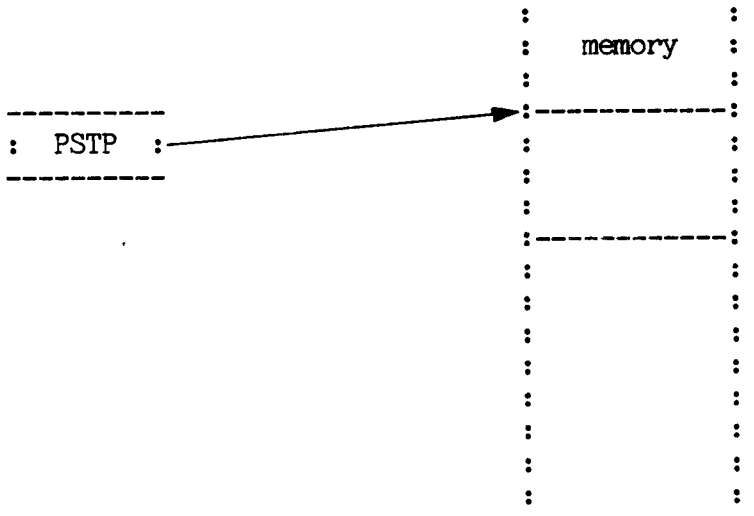


Fig. 4.9 Physical segment table

The access method, directly by physical page number, or indexed once or twice, depends on the size of the segment. Bit 14-15 of an element in the physical segment table holds information about access method.

Direct access restricts the segment size to 2 k bytes. Single indexing allows 1 k pages, or 2 megabytes maximum size. Larger segments use double indexing, the maximum size of which (2**31 bytes) exceeds the maximum segment size.

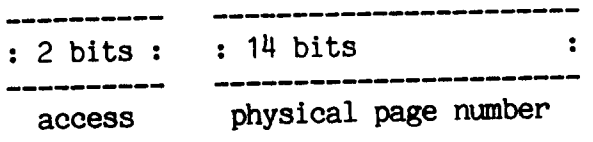


Fig. 4.10 Physical segment table entry

The two access bits have the following meaning

0 - direct, physical page number is data page
1 - single indexing, physical page number is the address of an index page
2 - double indexing
3 - unused

An index page entry has a layout similar to a PST entry, with the access bits reflecting the current indexing level: An index page to a single indexed segment has access bits equal to 0; the upper level index page to a double indexed segment has access bits equal to 1, the lower level index pages to 0. The physical address is calculated from the physical segment number and segment relative address as shown in the following figure.

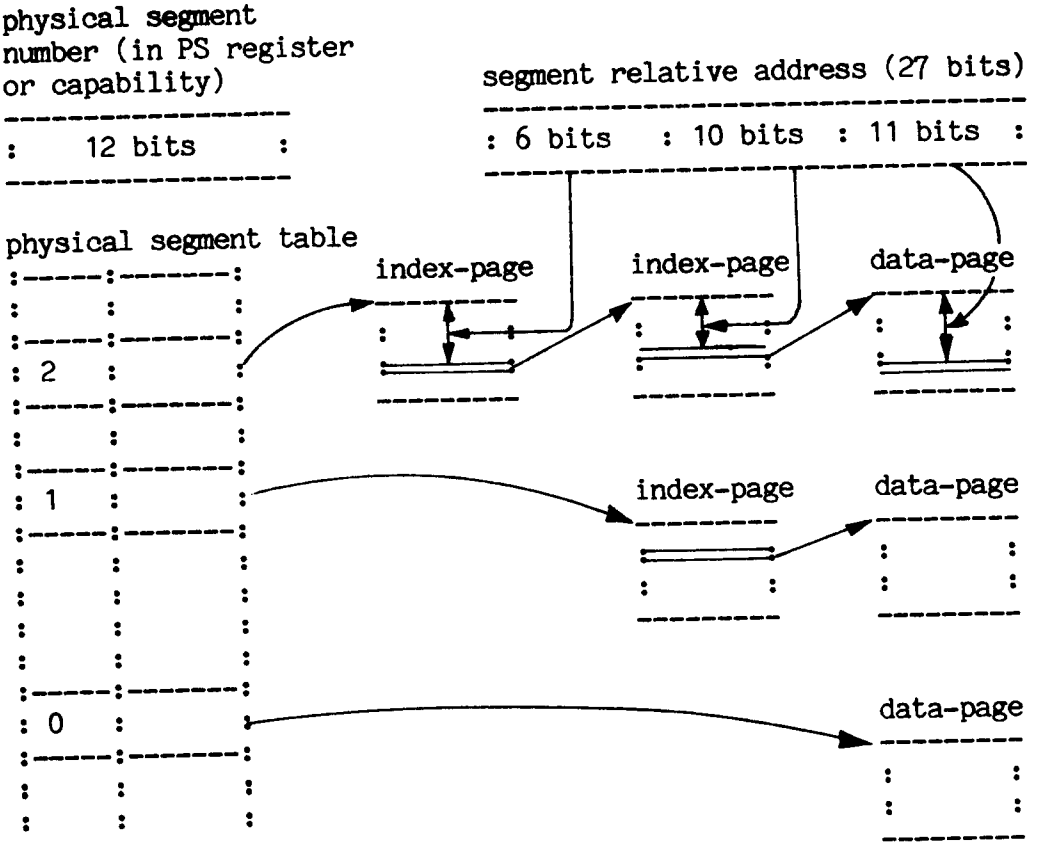


Fig. 4.11 Physical memory

MEMORY MANAGEMENT SYSTEM

The capability table holds the physical segment numbers of all logical segments in a domain. The capabilities are found on the segment specified by the process segment register (PS) of the process. On this segment, the currently executing domain register (CED) selects a 256 byte domain information table which includes the capability tables. The currently executing segment register (CES) selects an entry in the capability table, containing the physical segment number of the referenced segment.

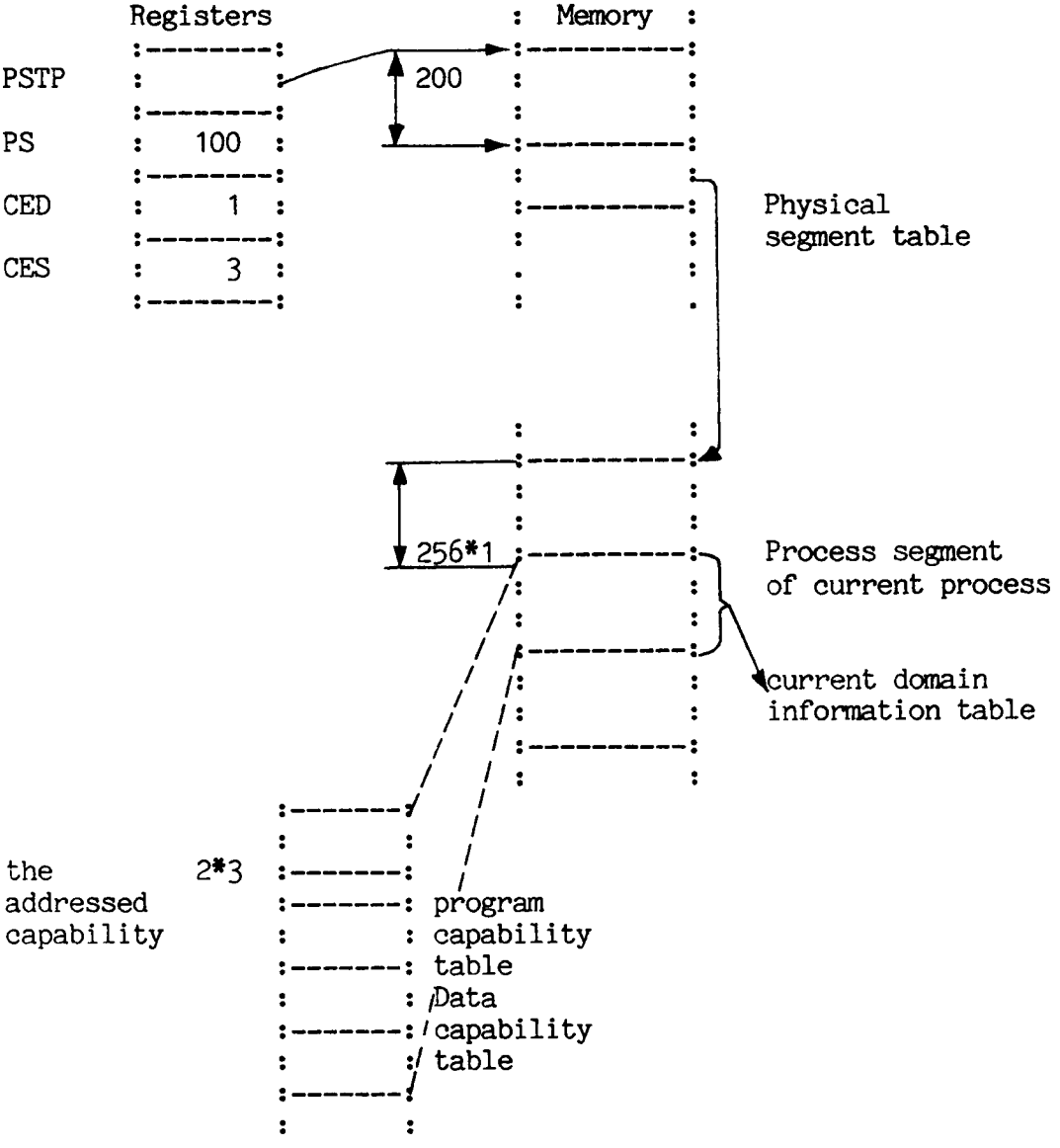


Fig. 4.12 Addressing a program capability

4.4. Buffering

Translation from logical to physical address is complicated and requires several memory accesses. To reduce the number of accesses the last used logical page number, domain number and a part of the process number is saved together with the corresponding physical page number and the permit bits of the corresponding capability. Later references to the same page may then avoid referencing the capability table, the physical segment table and the index pages.

The table used to hold this information is the Translation Speedup Buffer. The domain and process numbers are also stored. Therefore it is not neccessary to clear the buffer when changing domain or process.

The index in TSB is selected by using a hashing algorithm on the process number, the domain number and the logical page number. The buffer has a capacity of 256 elements plus an overflow of another 256 used for multi-operand instructions (such as POLY). (This capacity may be doubled by the microprogram, actually two banks of 512 elements exist.)

When access to memory is performed, the actual process number, domain number and logical page number are compared to the TSB counterparts pointed at by the index. If they are equal no further table lookup is neccessary and the physical page number in the translation speedup buffer is used. If they are not equal the memory management system will update the TSB once the neccessary information has been found.

MEMORY MANAGEMENT SYSTEM

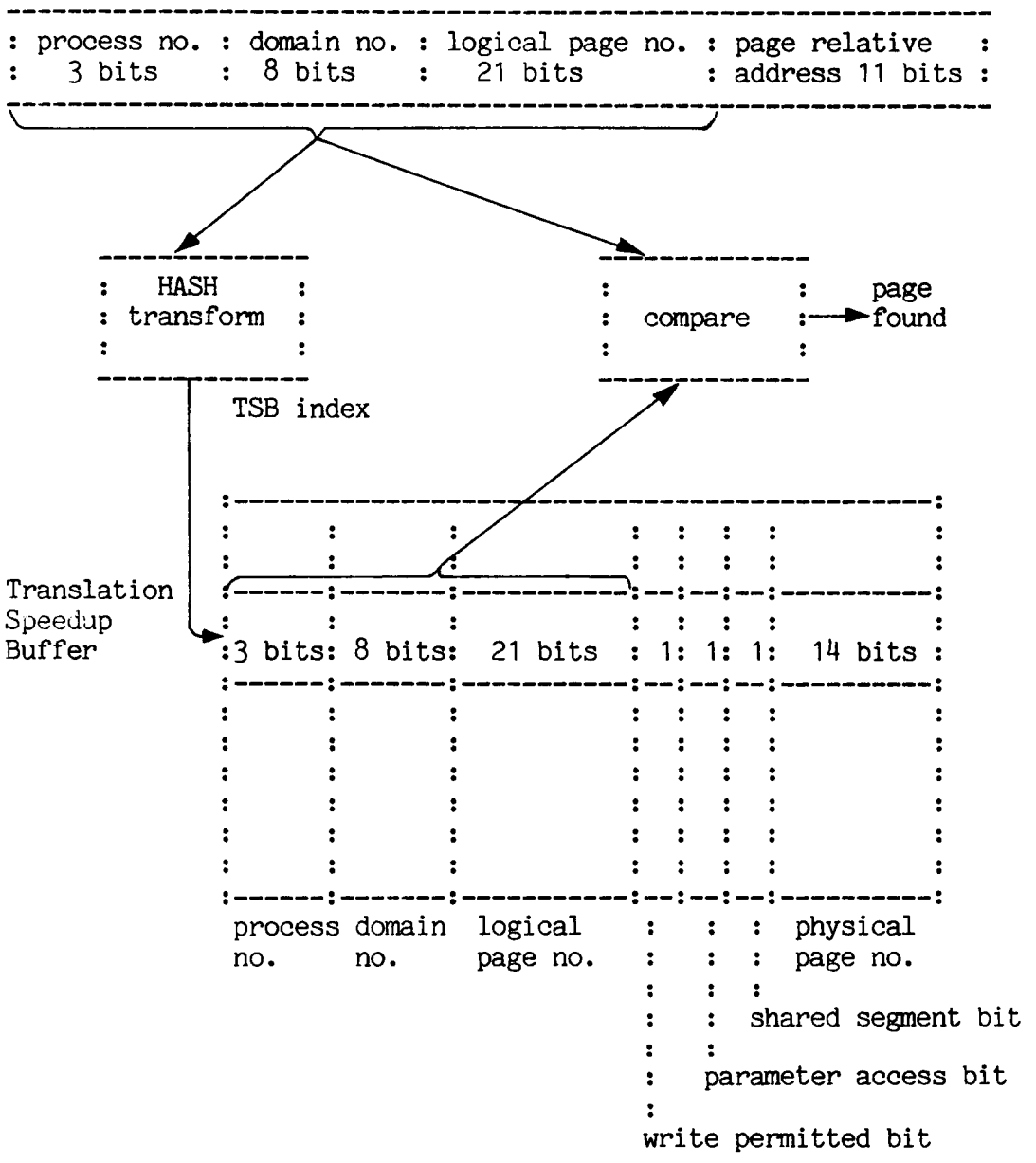


Fig. 4.13 Translation speedup buffer

5. CACHE MEMORY SYSTEM

The speed of the CPU is considerable higher than the speed of primary memory; if several memory accesses are required to complete an instruction, the CPU may be spending most of its time waiting for data to be loaded into the registers. To reduce the time spent waiting, the most recently used data are kept in high speed buffer memory, where data are available to the CPU in a fraction of the time required for a main memory access. This buffer is called cache. For economical reasons the cache is comparatively small, and sophisticated circuitry is employed to determine which data elements should be alotted space in the cache.

The effective memory access time as seen from the CPU is a fuction of several factors: The size and speed of the cache, main memory access time and the average percentage of data accesses where the requested data is available in the cache without further delay ("hit rate").

In NORD-500 the maximum cache size is 128 kilobytes. To prevent that instructions and data located at the same cache address constantly displace each other when a loop is executed, instructions and data have separate cache systems.
An access to main memory is up to 16 bytes wide. Most instructions are executed sequentially and data accesses are often made to variables located closely together, therefore the probability of finding the required data on the next request increases significantly.
Forward fetch mechanisms decode the operand specifiers in the instruction concurrently with the decoding of the instruction itself. "Data not in cache" can thus be detected and main memory access initiated at an early stage.

The two cache systems, for instructions and for data, are separate but identical. The cache word may be 32, 64, or 128 bits wide and the cache is always 4 K words deep. The width of the cache word is equal to the width of the channel to main memory.

The maximum cache size is 64 K bytes for instructions and 64 K bytes for data.

In addition to the bits of the data words, there are certain control bits in each word used by the cache system to identify the information stored. Parity on each byte is used for error detection.

The cache can be partitioned into 1, 2, or 4 parts of 4 K, 2 K, or 1 K cache words. Each partition can be assigned to one or more programs, or to library subroutines.

The cache is addressed by the logical address from the CPU and is byte addressable.

In a 128 bit wide and 4 K words deep system, the 4 least significant bits of the logical address are used to select the byte of the 16 byte word. The next 12 bits of the address are used to select one of the 4

CACHE MEMORY SYSTEM

K words. The remaining 16 bits of the 32 bit address are compared with 16 bits of word identification stored in cache. If they match, the requested data is present in cache and sent to the prefetch processor.

In a 64 bit wide cache, the 3 least significant bytes select the byte in the word, the next 12 bits select the cache word and the upper 17 bits are compared with the logical adress. In a a 32 bit wide system, 2 bits select the cache word byte, 12 bits the cache word and the upper 18 bits are matched with the logical address.

The cache and the memory management system is addressed in parallell. If the data is contained in the cache the CPU will fetch the data directly from the cache. Otherwise a request to main memory is generated after the logical address is converted to a physical address by the memory management system.

A "write through" algorithm is implemented, generating a write access to main memory in parallell with a write access to cache. Main memory will therefore always contain an updated copy of the contents of the cache.

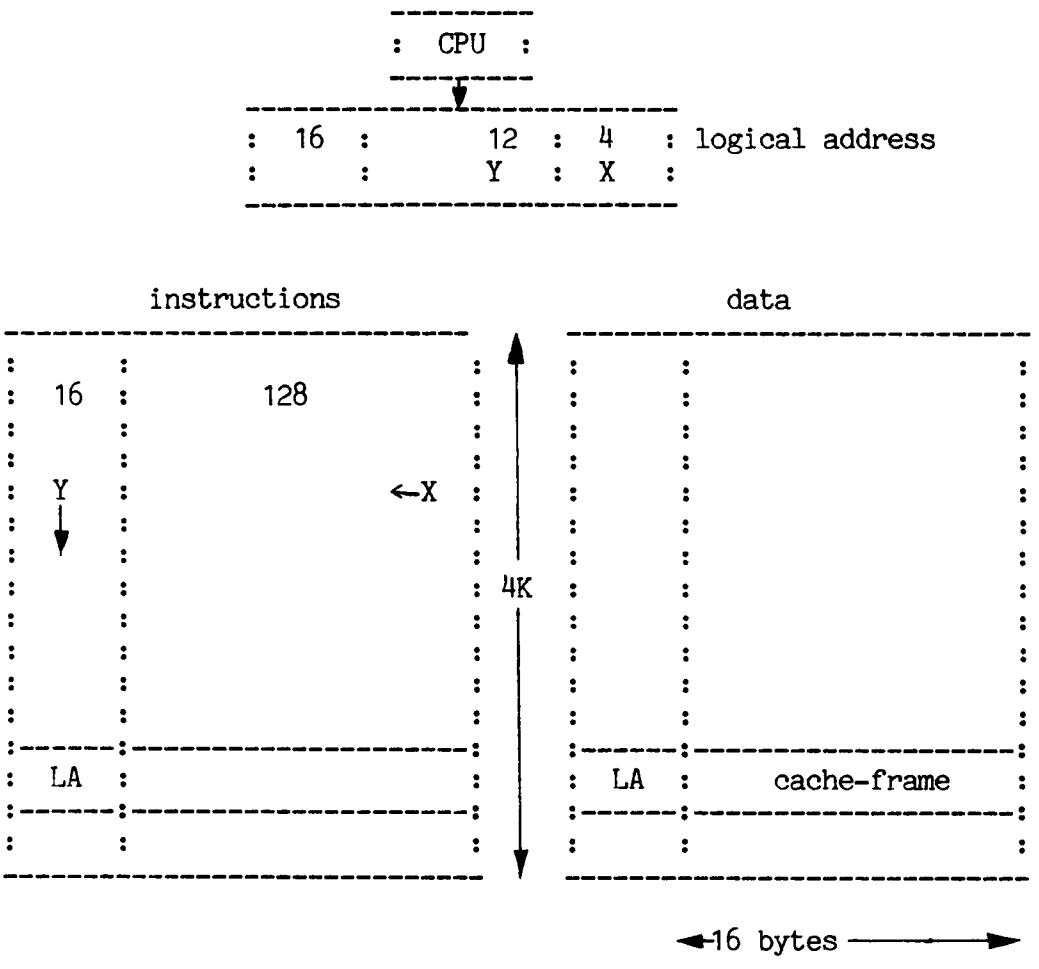


Fig. 5.1 The CACHE system, 128 K byte cache

6. THE TRAP SYSTEM

It is an advantage to be able to detect special situations arising during program execution, such as attempts to divide numbers by zero in a program performing many arithmetic divisions. Such checks may be made by software, but will require explicit programming.

NORD-500 CPU performs a number of checks automatically on every arithmetic operation, showing errors that would otherwise go unnoticed. Errors caught this way are said to be trapped. Situations leading to a possible trap are called trap conditions. A trap condition may or may not lead to a trap, depending on whether the trap is enabled or not. The above case is called a divide by zero trap condition.

Other examples of trap conditions are floating point overflow, illegal index and stack overflow.

For most trap conditions, it is possible to choose whether the trap is to be acted upon (ie. enabled) or not. If a trap is to be acted upon, a trap handler routine will be entered.

Trap conditions are divided into three categories depending on the way they are treated by hardware.

1. Ignorable trap conditions, which may be ignored
2. Non-ignorable trap conditions, which require treatment
3. Fatal trap conditions, fatal to the NORD-500 CPU

The NORD-500 CPU status register has one bit for each possible trap condition. When a trap condition occurs, this bit is set. The same bit is reset when a trap handler routine is invoked.

6.1. Trap handler routines

Most traps may be handled by a routine in the NORD-500. Every domain can have its own routines for the trap conditions allowed by its mother domain. If it does not take care of the trap itself, control may be transferred to the mother domain.

The mother may handle the situation, or hand it over to her mother. At the top of the domain tree is the operating system, and the NORD-100 is the "great grandmother" of all domains, ensuring there will always be at least one domain responsible for taking care of a trap propagated from lower levels. Eg. a trap condition encountered during the running of a user program may be handled in the user domain, in one of the mother domains between the user domain and the root of the tree, in the operating system domain, or in the NORD-100.

THE TRAP SYSTEM

After a trap situation has been taken care of, control will normally return to the instruction following that which caused the trap; for some trap conditions, the trapped instruction will be repeated or resumed. Note that the call sequence prior to the trap situation may be totally unrelated to the mother/child links.

Three registers in the NORD-500 are used for trap enabling: The Own (OTE), the Mother (MTE) and the Child (CTE) trap enable registers. Each domain has its own copy of these registers.

If a bit in OTE is set, the domain has a trap handler routine for the corresponding trap conditions occuring within the domain, and this routine will be called when a trap occurs. If the MTE bit is set, the mother (or grandmother etc.) domain of the trapping domain has a trap handler routine for this trap condition. If the corresponding bit in OTE is reset, this routine will be called.

A bit set in the CTE indicates that this domain has a trap handler routine to be used when the corresponding trap condition occurs in child domains, unless taken care of locally within the child domain. The CTE bit set will cause the MTE bit to be set in all child domains.

When a domain is created, it is given a trap enable modification mask (TEMM) from its mother. This mask specifies which bits in OTE the domain is allowed to change. An attempt to change a bit in OTE that is reset in TEMM will be ignored, while a change in an OTE bit that is set in the TEMM will have the desired effect.

MTE and CTE are not program modifiable. The system sets a bit in a domain's MTE if any of the mother domains in the tree structure have the corresponding bit set in their CTE register. The NORD-100 CPU will always be the mother of the upper NORD-500 domain. Trap conditions are always enabled in the NORD-100 CPU. Non-ignorable trap conditions may be enabled in the NORD-500 and handled by some program in the NORD-500. If they are not, they will be reported to NORD-100. Fatal trap conditions are always reported directly to NORD-100.

Status bits representing non-ignorable and fatal trap conditions will always yield a zero result (bit reset) if explicitly tested. It is not meaningful to perform a conditional jump on these bits, as the condition is always false.

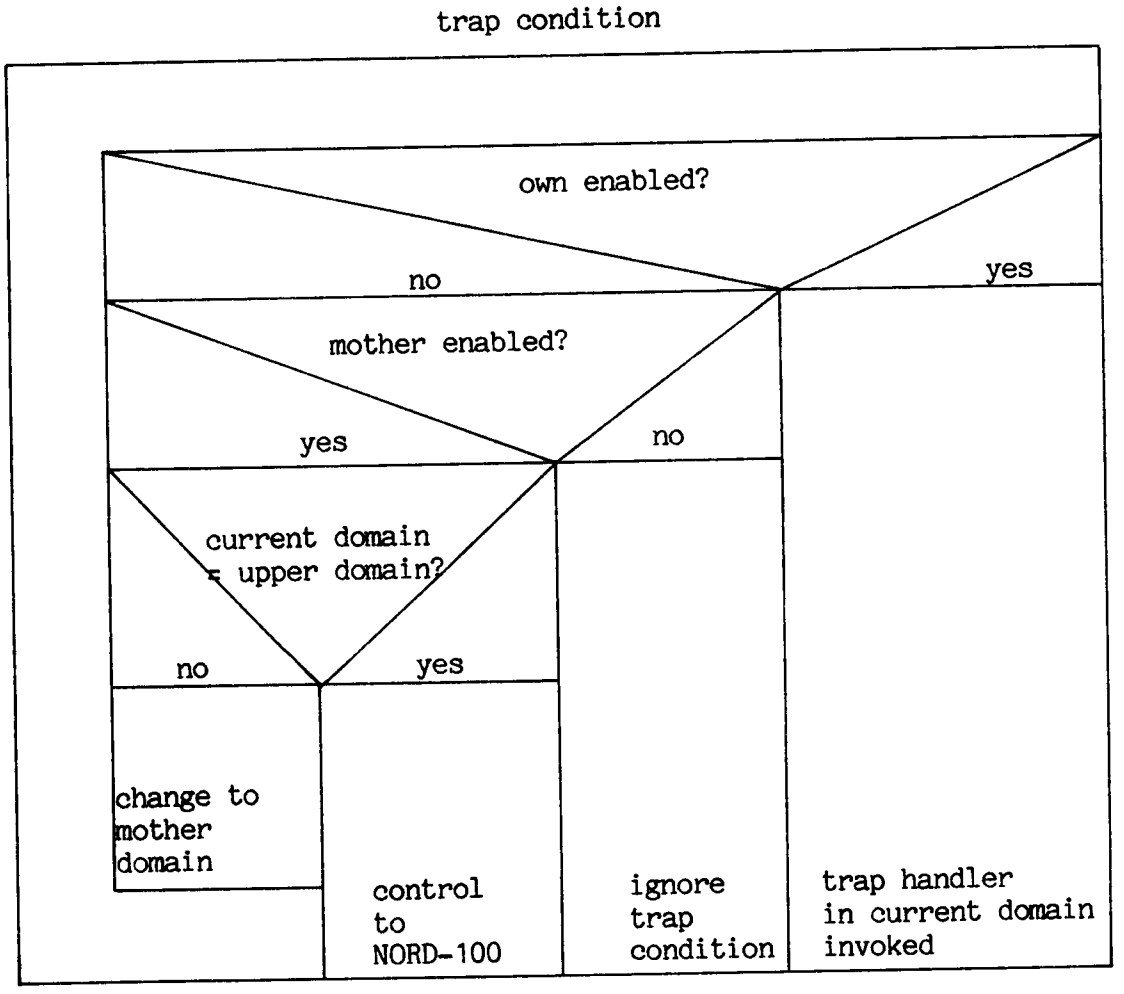


Fig. 6.1 Treatment of non-fatal trap conditions

THE TRAP SYSTEM

The Trap Handler Address register, THA, points to the base of an array in data memory, containing the start addresses of the trap handler routines in program memory. The n'th element of this array must hold the start address of the routine to handle the n'th trap condition. The area after the start address vector is used as a local data field for the invoked trap handler routine. This data field is filled by the ENTT instruction (see chapter 11.10).

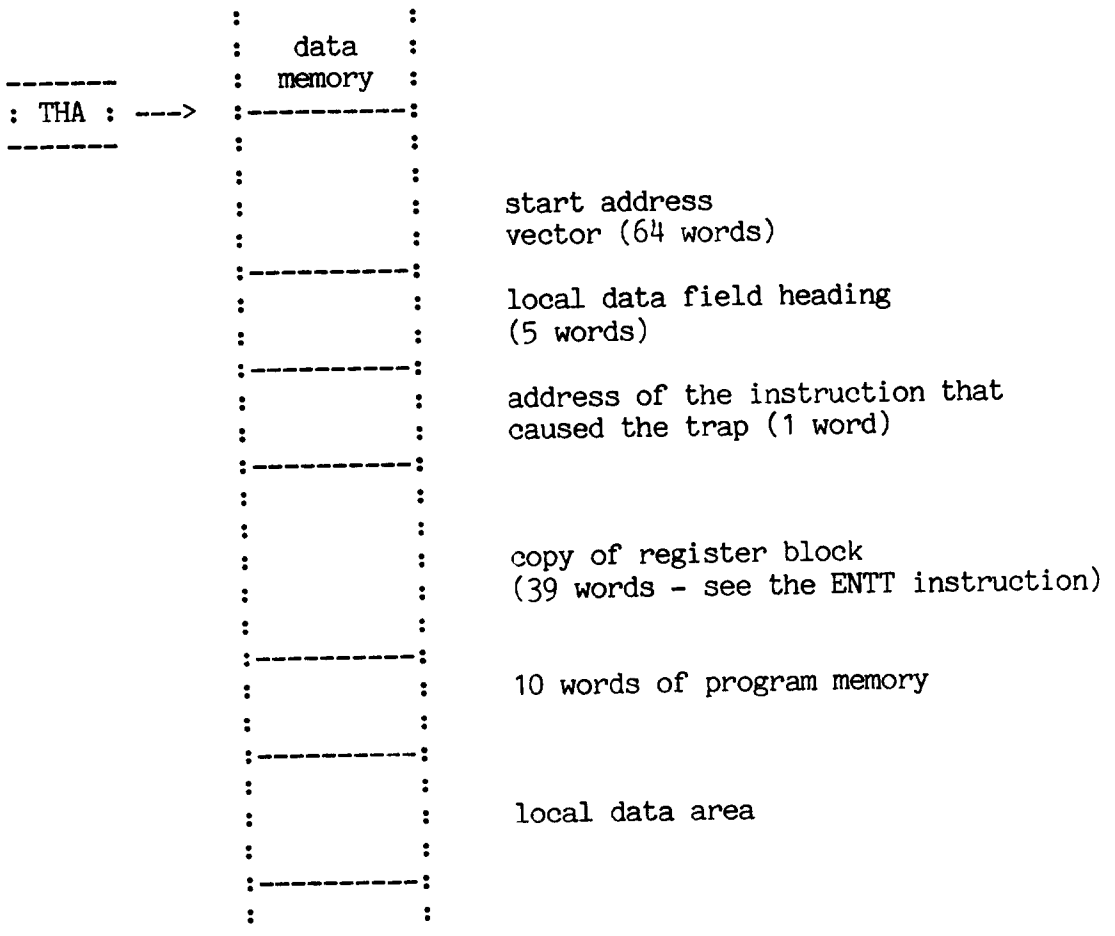


Fig. 6.2 Trap handler start address and local data field

When a trap handler is invoked, the address of the instruction that caused the trap condition, the register block, and information about the trap is saved in the local data area of the trap handler. The saved program counter holds the address of the instruction to be executed when the trap condition has been taken care of.

Trap handlers are not reentrant, due to the fixed location of the data area. The Own Trap Enable register (OTE) is therefore cleared, forcing propagation to the mother domain of any trap condition occuring during trap handler execution. The OTE register is reloaded from the domain information table on return from the trap handler.

When a trap handler is invoked, the status register (ST) is saved in the domain information table of the domain where the trap occurred. The layout and use of this table is described in more detail in the Memory Management section. If the trap condition is not handled by a local trap handler routine, an identification of the domain where the trap condition occurred is also saved in this table. Before the trap handler is entered the status bit causing the trap is cleared.

Status register bits representing ignorable trap conditions may be modified during running of the trap handler routine. Status bits representing non-ignorable and fatal trap conditions may not be modified. Setting a trap bit will cause a new trap immediately on return to the trapped routine. If several trap bits are set, several trap handlers will be called in sequence according to their bit numbers in the status register (highest numbered ones first).

Status bits are modified during running of a trap handler routine by modifying the status word in the saved register block. Upon trap handler return, this status word is "merged" with the saved status word in the domain information table and loaded into the status register. Unmodifiable status bits will contain their original values when the process continues.

Every enabled trap condition detected during the execution of an instruction will be reported to a trap handler routine in order of priority. The highest numbered traps are handled first.

6.2. The status register

There are 64 bits in the status register. 42 of these bits are currently defined. The status bits are grouped as follows:

Data status bits

Tracing status bits

Instruction and operand reference status bits

Signalling, synchronization and miscellaneous status bits

NORD-500 system error status bits

6.2.1. Data status bits

Code	Name	Bit no.
Z	zero	5
C	carry	6
S	sign	7
O	overflow	9
IVO	invalid operation	11
DZ	divide by zero	12
FU	floating underflow	13
FO	floating overflow	14
BO	BCD overflow	15

The data status bits hold information about the operand or result of the last executed operation on data. The majority of control and special instructions, including conditional jump instructions, leave the data status bits unaffected.

In the description of the instruction set, the effect on the data status bits is listed with every instruction. Bits that are set, reset or left unaffected are mentioned explicitly. All data status bits not mentioned are reset.

The Z, C, and S status bits have no corresponding trap conditions. They are used only for conditional jumps. All other data status bits are ignorable trap conditions. If trapping is not enabled, these bits may be tested with conditional jump instructions.

Z : The Zero bit is set if the operand/result of the last instruction was exactly zero. Otherwise it is cleared.

S : The Sign bit of the status register holds the sign bit of the last operand/result.

C : The Carry bit may be set only when performing integer arithmetic; otherwise it is cleared. The C bit is set if a carry out of or borrow into the most significant bit occurs. The contents of the carry bit is also used by the ADDC, SUBC and INVC instructions.

O : Integer Overflow may be set only when performing integer arithmetic; otherwise it is cleared. The O bit is set if the result of the operation is too large to be represented in the destination or register. It will occur in an integer addition when the sign bits of the two addends are equal, and the sign bit of the result is different from those of the addends. Note that subtraction is an addition of the two's complement of the subtrahend. In multiplication, integer overflow occurs when the destination is not large enough to hold the product. In case of overflow, the S and Z bits are set according to the actual result of the operation, rather than to the theoretical value. The least significant 32 bits of the extended result will be stored in the destination operand.

IVO : InValid Operation. Eg. executing a square root instruction with a negative argument will cause an invalid operation trap condition.

DZ : Divide by Zero trap. A division with zero will leave the largest possible value in the destination with the sign of the dividend, unless the dividend is also zero. Zero divided by zero gives a result of zero.

FU : Floating Underflow will occur if a negative exponent requires more than 9 bits to be represented. A value of zero will be stored in the destination. Other data status bits are set according to final result.

FO : Floating Overflow will occur in floating arithmetic if the result of an operation is too large to be represented in the NORD-500 floating point format, ie. a signed exponent requiring more than 9 bits. The largest possible floating point value will be stored in the destination, with the correct sign. Other data status bits are set according to final result.

BO : BCD Overflow. Overflow of the Binary Coded Decimal format. (BCD aritmetic is not yet implemented.)

6.2.2. Tracing status bits

Code	Name	Bit no.
SIT	single instruction trap	17
BT	branch trap	18
CT	call trap	19
BPT	breakpoint instruction trap	20

All the tracing status bits are ignorable trap conditions. They are valuable tools for debugging programs and performance evaluation.

SIT : Single Instruction Trap. This trap condition is caused when the execution of an instruction has terminated. With this trap condition, it is possible to step through a NORD-500 program one instruction at a time.

BT : Branch Trap condition occurs when the next instruction to be executed may be another than the one immediately following the last executed instruction; ie. after a GO, JUMPG, LOOP or conditional jump instruction. The trap condition occurs even if the test in the conditional jump is false and no jump is made.

CT : Call Trap condition occurs immediately after execution of a call subroutine instruction.

BPT : BreakPoint instruction Trap condition occurs when a breakpoint instruction is executed.

If several enabled trace trap conditions occur, the CPU handles the one with the highest priority first. Trace traps are listed from high to low priority in the following order:

Break Point Trap
Call Trap
Branch Trap
Single Instruction Trap

The tracing status bits are always reset when execution of the next instruction starts, even if they are not trap enabled. This means these bits are used for trapping purposes only, since they will always yield a zero result if explicitly tested.

6.2.3. Instruction and operand reference status bits

Code	Name	Bit no.
IOV	illegal operand value	16
ATF	address trap fetch	21
ATR	address trap read	22
ATW	address trap write	23
AZ	address zero access	24
DR	descriptor range	25
IX	illegal index	26
STO	stack overflow	27
STU	stack underflow	28
XSE	index scaling error	32
IIC	illegal instruction code	33
IOS	illegal operand specifier	34
ISE	instruction sequence error	35
PV	protect violation	36
PGF	page fault	38

These status bits are all trap conditions. Most are ignorable, but STO, STU, XSE, IIC, IOS, ISE and PV are considered so serious that they are defined as non-ignorable. PGF is defined as fatal; it may arise for all instructions. All trap conditions result from the decoding and accessing of instructions and operands.

Non-ignorable and fatal trap condition status bits are always zero when tested from a program, consequently they can be used only for trapping purposes. Ignorable trap condition status bits may be used either for trapping purposes or for explicit program testing (conditional jumps).

6.2.3.1. Ignorable trap conditions

IOV : Illegal Operand Value. Operand values exceeding the legal range, eg. in the bit field and call subroutine instructions, may cause an Illegal Operand Value trap condition. This status bit is set/reset in all instructions where there is given a limit for the operand values.

The NORD-500 has Low Limit (LL) and High Limit (HL) 32-bit registers for protecting program and data. These two registers are compared to the logical program and data address for each memory reference. If the actual logical address referenced is within the area limited below by the LL register and above by the HL register, a trap condition occurs whose type is determined by the current memory reference. (Memory reference type may be fetch, read, or write access.)

ATF : If the current memory reference is a program reference, a reference within the program memory area guarded by the LL and

HL registers will cause an Address Trap Fetch condition. The ATF status bit is set/reset at the end of each instruction.

ATR : If the current memory reference is a read reference to the data area guarded by the LL and HL registers, an Address Trap Read trap condition will arise. The ATR bit is set/reset at the end of each instruction.

ATW : If the current memory reference is a write reference to the area guarded by the LL and HL registers, it will cause an Address Trap Write trap condition. The ATW bit is set/reset at the end of each instruction.

AZ : Address bits 0-26 equal to zero will cause an Address Zero trap condition. INIT will set PREVB (see chapter on stack management) to zero, causing an AZ trap condition if attempts are made to link to a data block below the bottom of the stack.
A jump to segment address zero will also cause an AZ trap condition. Location zero will normally contain the number of routines on a segment, consequently it represents no meaningful instruction.
The AZ bit is set/reset for each instruction with memory access.

DR : Addressing via a descriptor may cause a Descriptor Range trap condition. This occurs if the contents of the index register is negative or greater than the maximum number of elements (length) field of the descriptor. The DR bit is set/reset at the end of all instructions with descriptor addressing. (See section chapter 8.3.12)

IX : The LIND and CIND instructions allow loading and calculating an array index and check that it does not exceed the array dimensions. If it does, it causes an Illegal indeX trap condition. The IX bit is set/reset by the LIND and CIND instructions.

6.2.3.2. Non-ignorable trap conditions

STO : When the contents of a new stack pointer in a stack subroutine call is greater than the contents of the TOS (top of stack register), a STack Overflow trap condition occurs. Stack overflow may also occur on execution of the GETB or ENTB instructions if there are no free data blocks of the requested size or larger. The STO status bit is set/reset for each enter stack subroutine and buddy allocation instruction.

STU : Performing a subroutine return instruction with RETA, PREVB or both equal to zero leads to a STack Underflow trap condition if there is no alternative domain (CAD zero or equal to CED) This status bit is set/reset at each return from a stack subroutine. This trap condition is also used to return control to the operating system when a program terminates (unless it is taken care of locally within the domain where the trap occured).

XSE : IndeX Scaling Error. The index exceeds 32 bits after post index scaling.

IIC : Illegal Instruction Code. Undefined code, or privileged instruction with the PIA status bit reset.

IOS : Illegal Operand Specifier. Constant operands as destination, prefixed argument to call instruction, type conflict between instruction and operands or non-constant number of arguments to call and polynomial instructions.

ISE : Instruction Sequence Error. Illegal subroutine entry point, illegal jump to another segment, illegal domain call nesting or an entry point instruction encountered not by executing a subroutine call.

PV : Protect Violation. This trap occurs when the segment access code in the capability table (see chapter 4.2.3) is violated.

6.2.3.3. Fatal trap condition

PGF : PaGe Fault. This trap may be caused by all instructions, and is a signal to the NORD-100 that another page has to be swapped in from backing storage. It will normally cause a process switch, but has no other effect on the program. If a page fault arises with the process switch disabled, it will cause a disable process switch error trap.

6.2.4. Signalling, synchronization and miscellanous status bits

Code	Name	Bit no.
K	flag	8
PRT	programmed trap	29
PIA	privileged instructions allowed	1
PD	part done	2
IR	instruction reference	3
PSD	process switch disabled	4
DT	disable process switch timeout	30
DE	disable process switch error	31

K : Flag. The flag bit is used for signalling purposes. There are special instructions for setting, resetting and testing this condition. The K flag is also used by instructions using descriptor addressing (see chapter 8.3.12) to indicate that the last element in the array is accessed and in string instructions to indicate termination conditions. All other instructions leave the K bit unchanged.

PRT : PRogrammed Trap. A process in the NORD-500 may interrupt another process by setting the second process' programmed trap status bit, which acts as a trap condition for this purpose. If the PRT trap is enabled the trapped process will immediately be interrupted and its trap handler invoked. If the process is not in the active state, as soon as it becomes active the trap will occur. If the process swith is disabled in the machine where the trapped process resides, the trap will occur as soon as the process switch is enabled.
The PRT bit is set through monitor calls. A process may trap itself by setting the PRT bit in the status register.

PIA : Privileged Instructions Allowed. Privileged instructions can only be executed when this bit is set; other attempts to execute privileged instructions will cause an illegal instruction code trap condition. This bit may not be changed by instructions, and is defined in the domain information table. Currently there are no privileged instructions.

PD : Part Done. This bit is used by the microprogram in long interruptable instructions to indicate if the instruction is to be restarted, eg. after page fault in string instructions.

IR : Instruction Reference. This is used by the paging system microprogram to indicate if there was a page fault on an instruction or on a data reference.

The NORD-500 is protected against bad synchronization procedures. Synchronization procedures can execute with the process switch disable status bit set. If this bit is set for more 256 microcycles, a process switch timeout trap condition occurs. Most simple instructions, like load, store, and simple arithmetic executes in one microcycle per operand specifier. When executing with process switch disable set, non ignorable traps (such as page fault) that require process switching must not occur. If they do occur, they cause a disable process switch error trap condition.

PSD : Process Switch Disabled. The process switch disable bit is only modifiable by the SOLO and TUTTI instructions.

DT : Disable process switch Timeout. Occurs if the process switch has been disabled for more than 256 microcycles.

DE : Disable process switch Error. Occurs if a non-ignorable process switch (like page fault) occurs while the process switch is disabled.

6.2.5. NORD-500 system error status bits

Code	Name	Bit no.
MOR	memory address out of range	37
PWF	power failure	39
PRF	processor fault	40
MSE	memory system error	41
CPE	cache parity error	42
MME	memory management system error	43

The system error status bits are all fatal CPU traps. On detection they are reported directly to the NORD-100 CPU.

MOR : Memory address Out of Range occurs when an address of non-existent physical memory is presented to the memory management system.

PWF : PoWer Failure.

PRF : PRocessor Fault. This is a fatal hardware error, caused by failure of hardware or microprogram.

MSE : Memory System Error. This means that there is a data error that cannot be corrected by the verification bits of primary memory.

CPE : Cache Parity Error.

MME : Memory Management system Error.

6.2.6. Addressing traps

In the instruction descriptions, the term addressing traps is used as a common name for all traps that may occur during operand fetching or instruction addressing. Most instructions may cause these traps, which include:

Address Trap Fetch	Descriptor Range trap
Address Trap Read	Illegal indeX
Address Trap Write	IndeX Scaling Error
Address Zero trap	Illegal Operand Specifier
Protect Violation	

6.2.7. Status bits survey

The first column indicates the trap type using the following abbreviations:

S - status bit, no corresponding trap condition
I - ignorable trap
N - non ignorable trap
F - fatal CPU error

The second column indicates whether the status bit is modifiable by software.

The third column indicates whether the trap is handled before, during, or after the current executing instruction:

Before : The instruction has not stored any results before the trap occurs. If the execution of the program may be resumed after handling the trap, the instruction will have to be executed once more. The P register and the "address of the instruction causing the trap" location in the trap handler local data area are of equal value.

During : This is the same as "Before" except for some instructions partially executed before the trap occurs and which may continue after being restarted. (String, block move and fill, call, enter, and return instructions) Instructions with one destination operand will not have stored a result, but destinations in multiple destination operand instructions have unpredictable values. If the instruction is to be restarted the trap handler should not modify any of the arithmetic registers.

After : The instruction causing the trap is completed and results stored before the trap occurs. If the execution of the program is resumed after the trap the next instruction is executed. The P register contains the address of the next instruction; the "address of the instruction causing the trap" location in the trap handler local data area contains the address of the instruction causing the trap.

THE TRAP SYSTEM

Trap type.....x
Modifiable(M).....x
Trap handled before(B),during(D),or after(A).....x

Bit no.	Name	Code			
0	not used				
1	privileged instruction allowed	PIA	S		
2	part done	PD	S		
3	instruction reference	IR	S		
4	process switch disable	PSD	S		
5	zero	Z	S	M	
6	carry	C	S	M	
7	sign	S	S	M	
8	flag	K	S	M	
9	overflow	O	I	M	A
10	not used				
11	invalid operation	IVO	I	M	A
12	divide by zero	DZ	I	M	A
13	floating underflow	FU	I	M	A
14	floating overflow	FO	I	M	A
15	BCD overflow	BO	I	M	A
16	illegal operand value	IOV	I	M	A
17	single instruction trap	SIT	I	M	A
18	branch trap	BT	I	M	A
19	call trap	CT	I	M	A
20	breakpoint instruction trap	BPT	I	M	B
21	address trap fetch	ATF	I	M	A
22	address trap read	ATR	I	M	A
23	address trap write	ATW	I	M	A
24	address zero access	AZ	I	M	A
25	descriptor range	DR	I	M	D
26	illegal index	IX	I	M	A
27	stack overflow	STO	N	M	D
28	stack underflow	STU	N	M	D
29	programmed trap	PRT	I	M	B
30	disable process switch timeout	DT	N		A
31	disable process switch error	DE	N		A
32	index scaling error	XSE	N		D
33	illegal instruction code	IIC	N		D
34	illegal operand specifier	IOS	N		D
35	instruction sequence error	ISE	N		D
36	protect violation	PV	N		D
37	memory out of range	MOR	F		D
38	page fault	PGF	F		D
39	power fail	PWF	F		A
40	processor fault	PRF	F		D
41	memory system error	MSE	F		D
42	cache parity error	CPE	F		D
43	memory management system error	MME	F		D
44-63	not used				

7. DATA TYPES

7.1. Introduction

In the NORD-500 programs and data are always stored in separate logical address spaces, referred to as the program memory and the data memory. Instructions are always stored in the program memory and operands usually in the data memory. Because the program memory functions as a read-only memory during program execution, instructions are protected from alteration.

Most instructions perform operations on operands. There are three categories of operands:

Variable operands residing in data memory
Constants residing in program memory,
as a part of the instruction using them
Register operands

7.2. Data types

The NORD-500 instruction set handles several basic data types: Bit, byte, halfword, word, float, doublefloat and binary coded decimal, abbreviated as BI, BY, H, W, F, D and BCD respectively. (BCD is not yet implemented.) Operations may also be performed on bit fields of varying lengths. In addition there are instructions allowing operations on arrays of BI, BY, H, W, F and D data. A large number of string instructions allow easy manipulation of character strings (byte arrays).

7.2.1. Bit

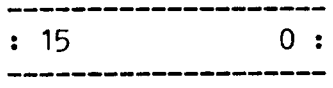
As the NORD-500 is a byte addressable machine, a bit is specified by its byte address. The specified bit is the rightmost bit (bit 0, the least significant bit) in the addressed byte. By post indexing or special instructions, it is possible to address bits other than bit zero.

An operand of type bit is a single bit, which is always treated as unsigned. The GETBF (get bit field) and PUTBF (put bit field) instructions operate on variable length (1 to 32 bits) bit fields. Note that these instructions treat the bit fields as signed quantities, even if they are only one bit long.

7.2.2. Byte

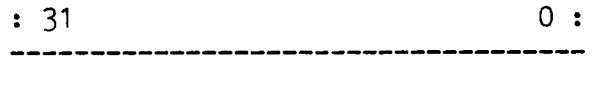
A byte is 8 contiguous bits starting at any byte boundary. The bits are numbered from the right, 0 to 7. Bit 0 is the least significant. A byte may be interpreted both as a signed and an unsigned integer. Signed byte values are in the range -128 to +127, represented in two's complement form. Unsigned byte values are in the range 0 to 255. Unsigned values may be interpreted as characters in any 8 bit (or less) character set, and instructions are available to set, check or clear the parity bit (bit 7) of a byte.

7.2.3. Halfword



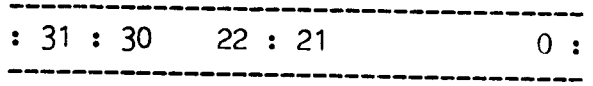
A halfword is 2 contiguous bytes, 16 bits, starting at any byte boundary. The bits are numbered from the right, 0 to 15. Bit 0 is the least significant. Like a byte, a halfword may be interpreted either as a signed or unsigned integer, in the range -32768 (-(2**15)) to +32767 ((2**15)-1) in two's complement form, or 0 to 65535 ((2**16)-1) respectively.

7.2.4. Word



A word is 32 bits, or 4 contiguous bytes, starting at any byte boundary. It may be used as an unsigned integer in the range 0 to 4294967295 ((2**32)-1) or as a two's complement integer in the range -2147483648 (-(2**31)) to +2147483647 ((2**31)-1).

7.2.5. Single precision floating point



sign : exponent : mantissa

A single precision floating point number is represented by a mantissa of 22+1 bits, a binary exponent of 9 bits with a bias of 256 and a sign bit. The range is 10**(-76) to 10**76 with an accuracy of approximately 7 decimal digits. Zero is represented as all bits zero. Minus zero (all but bit 31 zero) is interpreted as zero; minus zero will, however, never be returned as the result of a floating point operation.

7.2.6. Double precision floating point

sign : exponent : mantissa

A double precision floating point number is represented by a mantissa of 54+1 bits, a binary exponent of 9 bits with a bias of 256 and a sign bit. The range is 10**(-76) to 10**76, with an accuracy of approximately 16 digits. Zero is represented as all bits zero. Minus zero (all but bit 63 zero) is interpreted as zero; minus zero will, however, never be returned as the result of a floating point operation.

Floating point numbers are always normalized, - ie. the most significant bit in the mantissa is always one. It is therefore unneccessary to represent this bit explicitly. For single and double floating point numbers there is always one hidden bit in the mantissa, called the implicit bit. This is always assumed to be one, unless all bits in the exponent are zero. It is used in the arithmetic and removed from the result, thereby giving one more bit of precision. This is the reason why the length of the mantissa is expressed in terms of "+1".

The value of a floating point number is

```
S * 2**e * M        if e >< -256
0                   if e =  -256  (exponent bits all zero)
```

where S is the sign, with the value -1 if the sign bit is set and 1 if the sign bit is reset. e is the value of the 9 bit exponent (taken as an unsigned number) minus 256. Thus the range of e is -255 <= e <= 255. M is the mantissa interpreted as a binary fraction with the decimal point to the left of the implicit bit, giving a range of M of 0.5 <= M < 1.

Examples:

```
                        1 (implicit bit)
                        v
-1.0  = 1 100000001 00000000000000000000 = -1*2**(257-256)*0.5

12.75 = 0 100000100 10011000000000000000 =  1*2**(260-256)*0.796875

0.5   = 0 100000000 00000000000000000000 =  1*2**(256-256)*0.5

0.375 = 0 011111111 10000000000000000000 =  1*2**(255-256)*0.75

-5.0  = 1 100000011 01000000000000000000 = -1*2**(259-256)*0.625

0.0   = 0 000000000 00000000000000000000     (special case)
```

7.2.7. Floating point rounding

After a floating point operation, the result is normalized and the full mantissa is checked for rounding. Rounding up is done by adding one to the least significant bit of the mantissa, rounding down is done by ignoring bits beyond the least significant bit. The bits affecting the rounding are labeled

L - least significant bit of that part of the full mantissa which goes into a float or double float mantissa
G - the bit nearest L to the right
S - the result of an OR operation of all bits to the right of G

```
- -- ----------------
          : L : G : S :
- -- -- --------:--------
    Mantissa   :
```

```
if G=1 and (S=1 or L=1) then
   add one to the least significant bit of mantissa
endif
```

Fig. 7.1 Floating point rounding

The effective result is equivalent to rounding up when the last decimal digit is larger than 5, rounding down if it is less than 5. If the last decimal digit is equal to 5, the rounding up or down is determined by the L bit, causing round off errors to take both positive and negative values in order to partially self-compensate in long computations.

7.2.8. Descriptor

A descriptor is used for addressing arrays and strings (byte arrays) through the DESC prefix. The descriptor consists of 8 bytes, the first four containing the length of the array, the last four containing the address of element number zero.

bytes 0 to 3	bytes 4 to 7
Number of elements	Address of element 0

The hardware will compare the first half of the descriptor against the value of the index register used. Illegal indexing will be trapped as a Descriptor Range error (DR). Indexing is assumed to range from zero upwards; thus, index values below zero or larger or equal to the number of elements are illegal.

7.3. Data formats in main memory

Data are stored in memory in various ways depending on their type. The basic unit in the NORD-500 memory is byte. In data types which consist of more than one byte, the bytes are numbered left to right. The bits in a single element of a data type are numbered right to left. The leftmost bit is the most significant bit.

Please note that post indexing always counts the elements from the left, even if the data type is bit.

byte0	byte1	byte2	byte3

When addressing with byte, halfword, or word displacement part, the calculated address is the address of the leftmost (lowest numbered or most significant) byte. Addressing with short address codes is either B or R relative and has word as the displacement unit. The memory must then be looked on as if the basic unit is word, and the data object must be located on a word boundary. The calculated address is the leftmost byte of the word. When addressing with short word displacement, the byte displacement is 4 * word displacement. (This is taken care of by the assembler and will be of little concern to the programmer.)

An array is addressed by its zeroth element, a multi-dimensional array by the element having all indexes zero. This may be a "virtual" element, in case the range of valid index values does not include zero, or the array may actually start at a lower address if negative indexes are allowed.

DATA TYPES

Most multi-operand instructions require operands to be of the same type. The operands will be addressed as such, which may cause unexpected results. If, for example, a byte is addressed as a word, the intended byte and the following three bytes in memory will be used as if it were a word sized data item.

BIT:	The rightmost bit of a byte, specified by the byte address.
BYTE:	8 contiguous bits, starting at any byte boundary.
HALFWORD:	16 contiguous bits (2 bytes), starting at any byte boundary and addressed by the leftmost byte.
WORD:	32 contiguous bits (4 bytes), starting at any byte boundary and addressed by the leftmost byte.
FLOAT:	32 contiguous bits (4 bytes), starting at any byte boundary and addressed by the leftmost byte.
DOUBLE FLOAT:	64 contiguous bits (8 bytes), starting at any byte boundary and addressed by the leftmost byte.
DESCRIPTOR:	64 contiguous bits (8 bytes), starting at any byte boundary and addressed by the leftmost byte.

Fig. 7.2 Data formats in main memory

7.4. Data in registers

Data may be stored temporarily in the registers in the NORD-500 CPU register block. Integer data types, ie. BI, BY, H and W data, may be stored in the four Integer registers (In, n=1,2,3,4). Floating point data types, ie. F and D data, may be stored in the four floating point Accumulators (An, n=1,2,3,4). The floating point accumulators may be extended with the Extension registers (En, n=1,2,3,4) for double precision floating point data. Data is stored in the registers as shown in the figure below.

The In accumulators are named BIn, BYn, Hn and Wn when used for BIt, BYte, Halfword, or Word operations. (n=1,2,3,4)

The An accumulators are named Fn when used as single precision registers. The (An,En) double registers are named Dn when used as double precision floating point registers.

A common name for BIn, BYn, Hn, Wn, Fn and Dn is Rn. Rn may be used when referencing a register where the type is determined by the context.

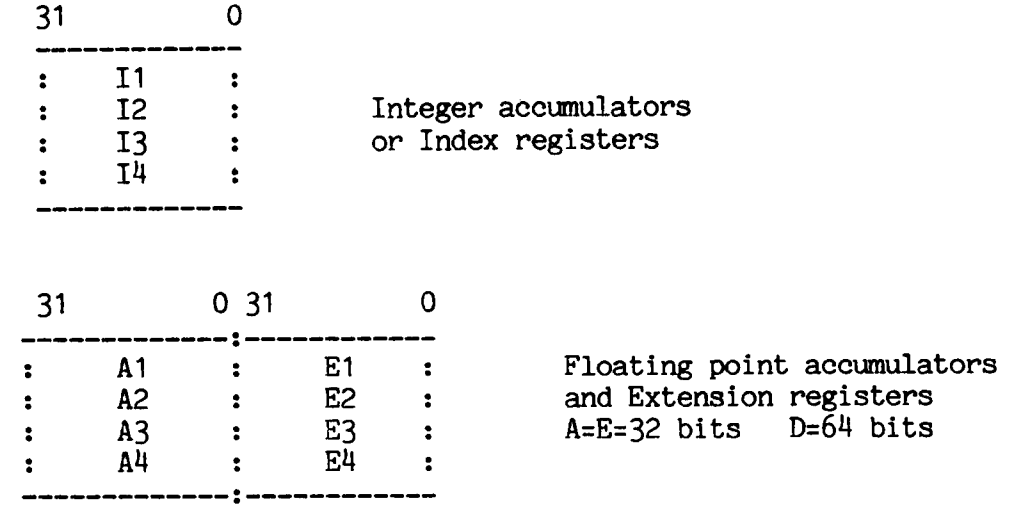


Fig. 7.3 Arithmetic registers

DATA TYPES

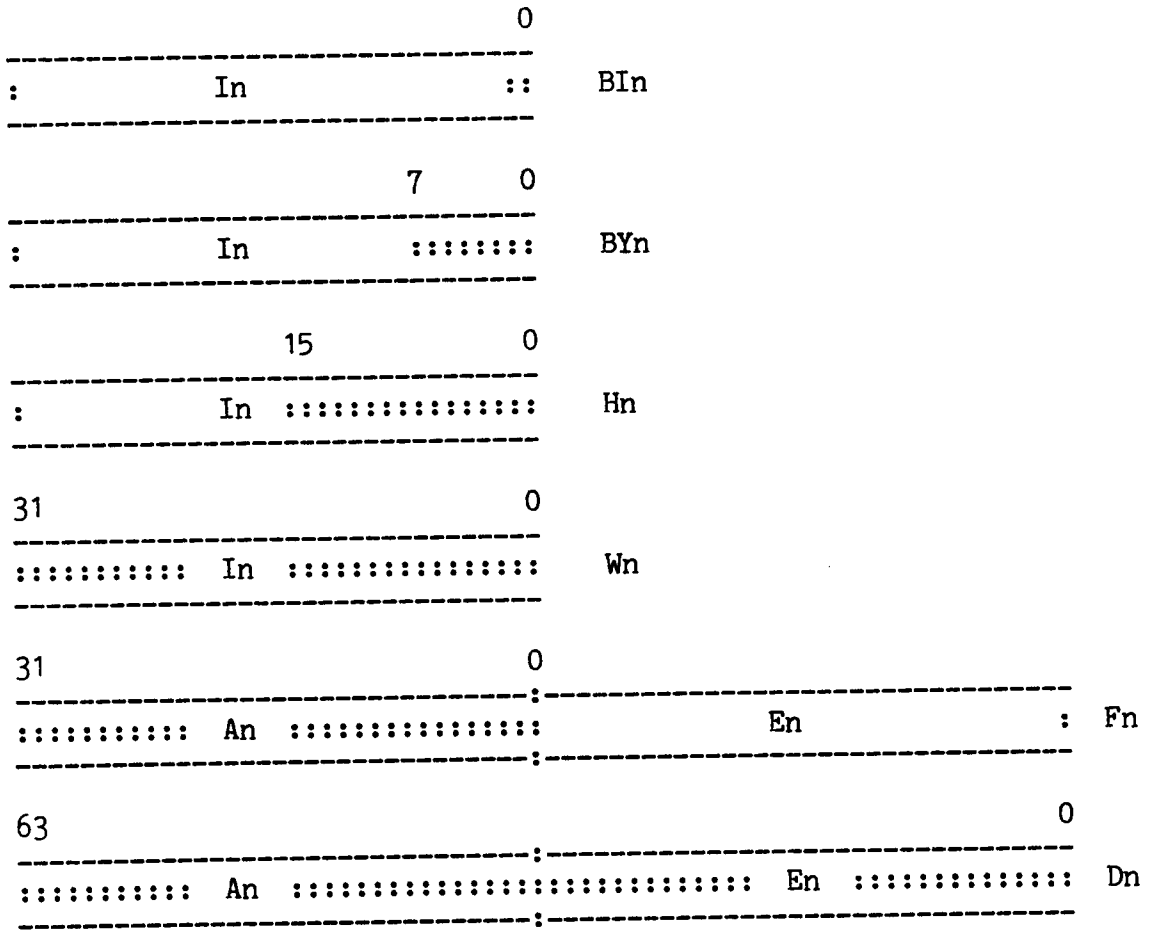


Fig. 7.4 Data in registers

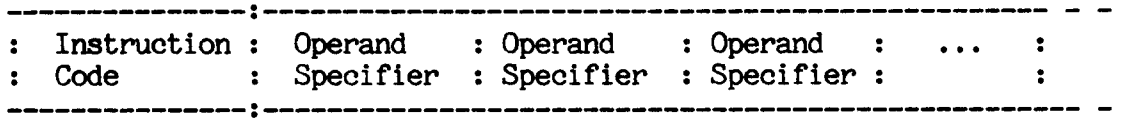
When using the integer registers for BIt, BYte and Halfword, the unused upper part of the register is always zero-filled rather than sign-extended when data is loaded into the register.

When single float data is stored in one of the Fn registers, ie. An, the corresponding En register remains unchanged.

8. OPERAND SPECIFIERS AND ADDRESSING

8.1. Introduction

An instruction consists of an instruction code and zero or more operand specifiers. The general instruction format is shown in the figure below:



1 or 2 bytes Zero or more operand specifiers, each 1 to 9 bytes

Fig. 8.1 Instruction format

The instruction code specifies the operation to be performed and the operand data types. The operand specifier names the data to be worked on. This chapter describes the different formats of the operand specifier. The next chapter gives details of the instruction code.

In many NORD-500 instructions one of the integer registers or one of the floating point registers are used as argument or result. The two lower bits of the instruction code then specify the register number, a floating point or double precision floating point register (Fn or Dn) when the data type is floating or double floating and an integer register (In) when the data type is integer.

8.2. General and direct operands

8.2.1. Introduction

An operand specifier designates the data for an instruction to work on. If an instruction requires several operands, a corresponding number of operand specifiers follow the instruction code.

prefix(es)	address code	data part

Fig. 8.2 Operand specifier format

The length of an operand specifier may be one to nine bytes.

Operand specifiers are divided into general operand specifiers and direct operand specifiers. The interpretation of a general operand is determined by an address code, data part and optional prefix(es). The interpretation of a direct operand depends on the instruction; the operand may only have a data part, no prefix or address code.

The instruction determines whether a general or a direct operand should be used. Instructions using direct operands are mentioned in 8.4; all others use general operands. Direct operands are used most places where the operand value has to be a constant of a specific type, and the operand value can be determined unambigously as the contents of the succeeding bytes.

The notational conventions used in this manual to indicate general and direct operands are explained in Appendix C. Operand names are chosen to give more information about the specific operand in use, eg. <source>.

The following table describes the structure of operand specifiers in relation to general and direct operands. The blank part of the table indicates that there are no prefixes or addressing codes for direct operands and no prefixes for constant and register general operands. All general operands have an address code.

Operand specifier

	prefix	address code	data part
General operands:			
1) Constant		--------	constant
2) Register		--------	
3) Data memory	----	--------	absolute address or displacement
Direct operands:			
1) Absolute address (program/data memory)			absolute address
2) Displacement (program relative)			displacement
	1 or 2 bytes	2 bits or 1 byte	6 bits, 1,2,4 or 8 bytes

Fig. 8.3 Operand specifier structures

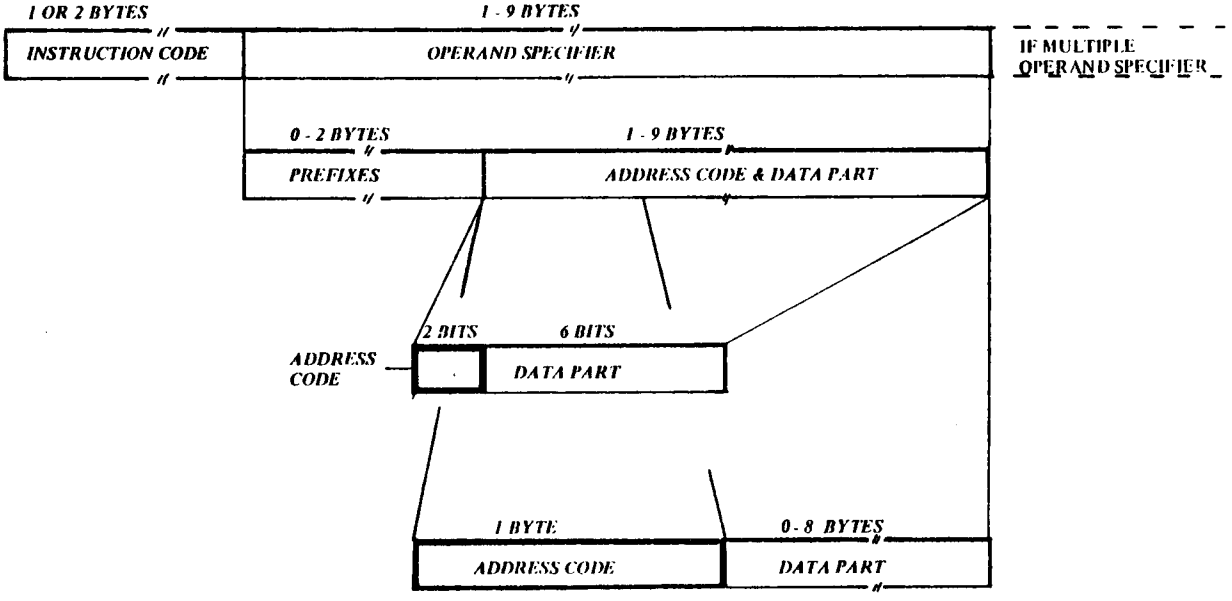


Fig. 8.4 Operand specifier layout

8.2.2. General operands

A general operand consists of the address code, the data part and possibly a prefix.

The Address Code

The address code is either 2 bits or 1 byte long. It indicates both the address mode, of which there are 10 types, and the length of the data part, of which there are 6. Combinations of address modes and data part lengths give 28 different address codes.

The table below lists the data part length specifiers (in the NORD-500 assembler notation), names and sizes. Note that :W and :F are different assembly notations for the same operand specifier format.

:S	-	short	6 bits
:B	-	byte	1 byte
:H	-	halfword	2 bytes
:W	-	word	4 bytes
:F	-	floating	4 bytes
:D	-	double float	8 bytes

Fig. 8.5 Data part length specifiers

Normally the NORD-500 assembler will select the optimal displacement size. It is possible, however, to force a particular (larger) size of displacement by following the operand specifier by either :S, :B, :H, :W, :F or :D. (The last two applies to constants only.) In examples shown, a data part length specifier is used only when forcing a non-default data part length.

The following table shows the 10 address modes and the 6 data part length specifiers. Legal combinations are marked with +. Note that post index is abbreviated as P.I.

Address mode	Data part length specifier						No data part
	:S	:B	:H	:W	:F	:D	
1. LOCAL	+	+	+	+			
2. LOCAL P.I.		+	+	+			
3. LOCAL INDIRECT		+	+	+			
4. LOCAL INDIRECT P.I.		+	+	+			
5. RECORD	+	+	+	+			
6. PRE INDEXED		+	+	+			
7. ABSOLUTE				+			
8. ABSOLUTE P.I.				+			
9. CONSTANT	+	+	+	+	+	+	
10.REGISTER							+
Operand specifier prefix:							
DESCRIPTOR							+
ALTERNATIVE							+

Fig. 8.6 NORD-500 address modes

Most address codes contain '11' in the leftmost two bits. The remaining six bits in the byte then specify the code.

However, in 3 special cases the leftmost two bits are '00', '01' or '10'. These are the short address codes (:S in the table) and the two bits alone indicate both length and mode. The remaining six bits are then taken as the data part, so that the complete operand specifier occupies only one byte.

The Data Part

The last part of the operand specifier, the data part, may be from six bits (for short data parts) to 8 bytes (for double word data parts). The data part contains an address, a displacement or a constant. The register address mode has no data part since the register number is contained in the address code.

OPERAND SPECIFIERS AND ADDRESSING

Addresses are always word sized. Short, byte and halfword displacements are always treated as unsigned values.

The displacement unit is always bytes, except for short displacements, where the unit is words. The range for short displacement is consequently 0..63 word from the record or base registers, and the addressed data object must be located an integral number of words from the register referred.

Prefixes

All address codes except constant and register may include prefixes as the first 1 or 2 bytes. These are used in two special cases where the operand specifier does not point to the operand itself. Such an operand specifier may point to an array descriptor or to an operand on an alternative domain. The prefixes are followed by the operand specifiers.

The only allowed two-prefix combination is when an operand points to an array descriptor referring to an alternative domain, written as ALT(DESC(<operand>)(Rn)). Only the last data access then goes to the alternative domain; the descriptor itself is accessed in the current domain.

8.2.3. Post Index

Post index is used in the local post indexed, the local indirect post indexed, absolute post indexed and the descriptor addressing modes.

Post indexed addressing means that the index register holds the address of the operand element relative to the start of the addressed structure. In NORD-500 the index is always a logical index giving the element number in the array, regardless of the element size. Accessing the next element in the structure is done by incrementing the index register by one.

Hardware will multiply the logical index with a data type dependent factor, the post index scaling factor. The result gives the physical index. The post index scaling factor is the number of bytes used to represent the data type in question. The post index scaling factor is 1/8 (BI), 1 (BY), 2 (H), 4 (W), 4 (F), 8 (D) and 8 (descriptor). The physical index is added to the base address of the structure in order to get the address of the operand. Note that the index is signed.

8.3. Survey of addressing modes

The first column lists the different groups of addressing modes in the assembler notation for displacements and the name of the displacement. The second column lists the algorithm used for determining the effective address (ea) of the operand or the operand itself. The third column lists the address code. (Abbreviations are explained in Appendix C.)

		Hex code	Octal code
LOCAL			
B. <displ> :S short displacement	ea=(B)+d*4	040H+xx	100B+xx
B. <displ> :B byte displacement	ea=(B)+d	0C1H	301B
B. <displ> :H halfword displacement		0C2H	302B
B. <displ> :W word displacement		0C3H	303B
LOCAL, POST INDEXED			
B. <displ> (Rn) :B byte displacement	ea=(B)+d+p*(Rn)	0D4H+y	324B+y
B. <displ> (Rn) :H halfword displacement		0D8H+y	330B+y
B. <displ> (Rn) :W word displacement		0DCH+y	334B+y
LOCAL INDIRECT			
IND (B. <displ> :B) byte displacement	ea=((B)+d)	0C5H	305B
IND (B. <displ> :H) halfword displacement		0C6H	306B
IND (B. <displ> :W) word displacement		0C7H	307B

OPERAND SPECIFIERS AND ADDRESSING

LOCAL INDIRECT, POST INDEXED			
IND (B.<displ> :B) (Rn) byte displacement	ea=((B)+d)+p*(Rn)	0E4H+y	344B+y
IND (B.<displ> :H) (Rn) halfword displacement		0E8H+y	350B+y
IND (B.<displ> :W) (Rn) word displacement		0ECH+y	354B+y
RECORD			
R. <displ> :S short displacement	ea=(R)+d*4	080H+xx	200B+xx
R. <displ> :B byte displacement	ea=(R)+d	0C9H	311B
R. <displ> :H halfword displacement		0CAH	312B
R. <displ> :W word displacement		0CBH	313B
PRE-INDEXED			
Rn. <displ> :B byte displacement	ea=(Rn)+d	0F4H+y	364B+y
Rn. <displ> :H halfword displacement		0F8H+y	370B+y
Rn. <displ> :W word displacement		0FCH+y	374B+y
ABSOLUTE			
<address>	ea=a	0C4H	304B
ABSOLUTE, POST INDEXED			
<address> (Rn)	ea=a+(Rn)*p	0E0H+y	340B+y

CONSTANT

<constant> :S short constant	op=c	000H+xx	000B+xx
<constant> :B byte constant		0CDH	315B
<constant> :H halfword constant		0CEH	316B
<constant> :W , <constant> :F word constant, floating point constant		0CFH	317B
<constant> :D double floating point constant		0CCH	314B

REGISTER

Rn	op=(Rn)	0D0H+y	320B+y

DESCRIPTOR

DESC (<descriptor>) (Rn)	ea=A+p*(Rn)	0F0H+y	360B+y

```
(Rn)+1=:Rn
if (Rn) > descriptor.length then
   descriptor range trap condition
endif
if (Rn) >= descriptor.length then
   1=:status.flag
endif
```

ALTERNATIVE

ALT (<operand>)		0C8H	310B

The address (ea) is referenced on the alternative domain. Parameter access is required on the referenced segment in the alternative domain.

8.3.1. Local addressing

Assembly notation	Name	Hex code	Octal code
B.<displ>	local		
B.<displ>:S	local, short displacement	040H+xx	100B+xx
B.<displ>:B	local, byte displacement	0C1H	301B
B.<displ>:H	local, halfword displacement	0C2H	302B
B.<displ>:W	local, word displacement	0C3H	303B

ea = (B)+d

The local addressing mode is addressing relative to the base register B. This register is meant to hold the address of the beginning of the local variables of a routine, hence the name local addressing.

The effective address is calculated by adding the value of the displacement to the content of the base register.

Short displacement part with a displacement unit of word is legal, in addition to byte, halfword and word displacement parts with the displacement stored in 1, 2, or 4 byte(s) after the address code, displacement unit byte. Displacement values are treated as unsigned.

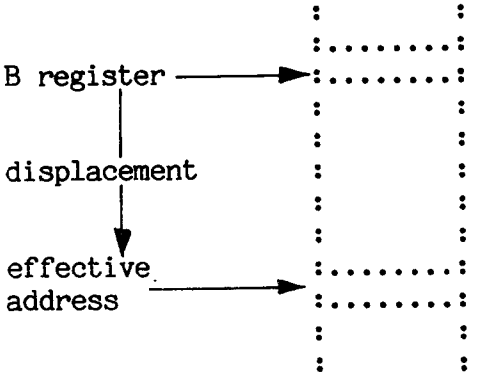
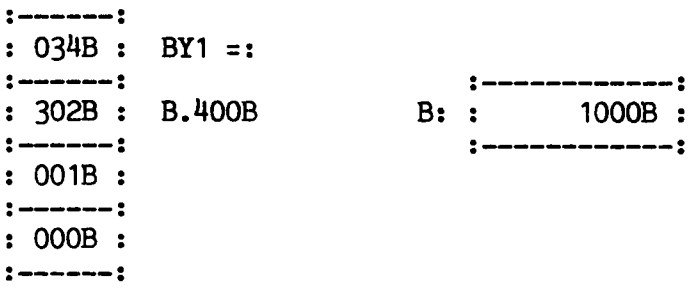


Fig. 8.7 Local addressing

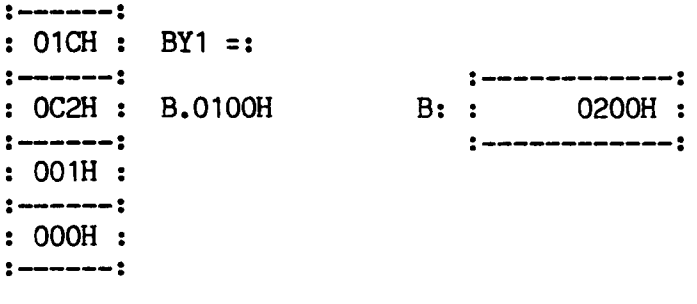
Example:



ea = (B)+d = 1000B+400B = 1400B

Octal

Hexadecimal



ea = (B)+d = 0200H+0100H = 0300H

8.3.2. Local, post indexed addressing

Assembly notation	Name	Hex code	Octal code
B.<displ>(Rn)	local, post indexed		
B.<displ>(Rn):B	local, post indexed, byte displacement	0D4H+y	324B+y
B.<displ>(Rn):H	local, post indexed, halfword displacement	0D8H+y	330B+y
B.<displ>(Rn):W	local, post indexed, word displacement	0DCH+y	334B+y

ea = (B)+d+p*(Rn)

A local post indexed address is calculated by adding the displacement, the content of the B register and the content of the index register multiplied by the post index scaling factor. See the section on post indexing.

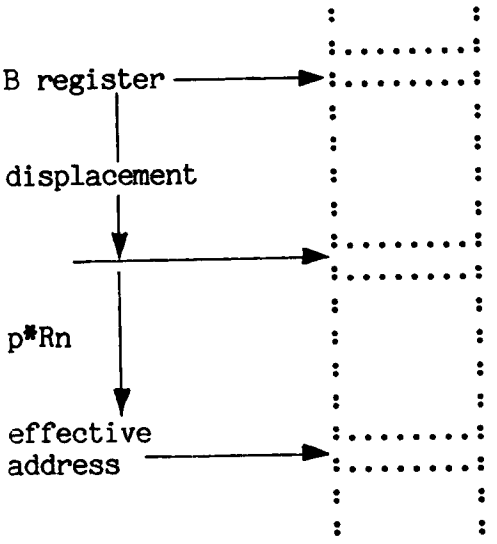
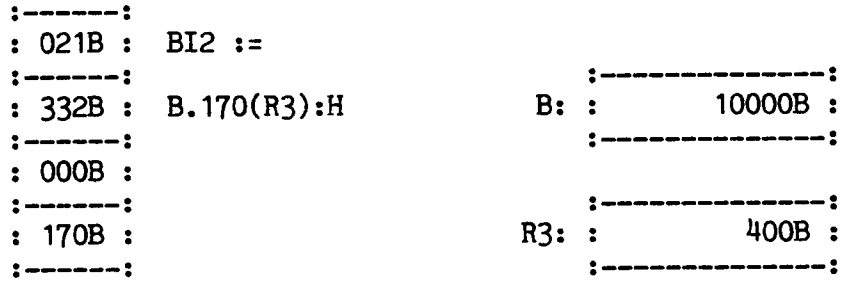


Fig. 8.8 Local, post indexed addressing

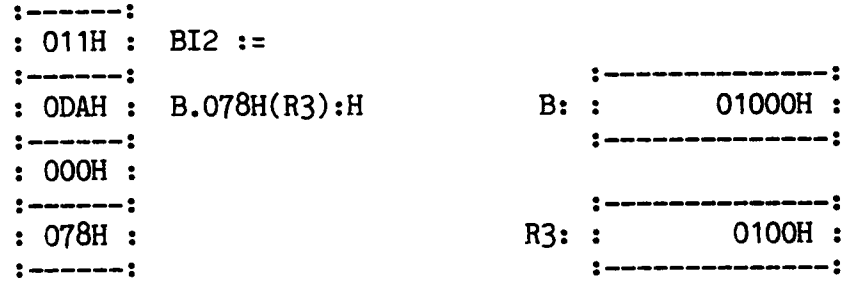
Example:



ea = (B)+d+p*(Rn) = 10000B+170B+400B/10B = 10230B

Octal

Hexadecimal



ea = (B)+d+p*(Rn) = 01000H+078H+0100H/08H = 01098H

8.3.3. Local indirect addressing

Assembly notation	Name	Hex code	Octal code
IND(B.<displ>)	indirect		
IND(B.<displ>:B)	indirect, byte displacement	0C5H	305B
IND(B.<displ>:H)	indirect, halfword displacement	0C6H	306B
IND(B.<displ>:W)	indirect, word displacement	0C7H	307B

ea = ((B)+d)

The value of the unsigned displacement is added to the local base register and this sum forms the address of a word which holds the address of the operand. Subroutine arguments are usually accessed by local indirect addressing.

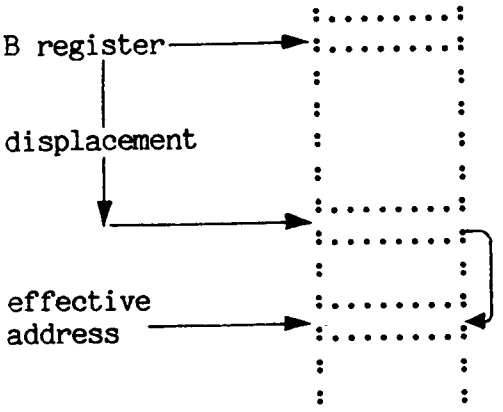
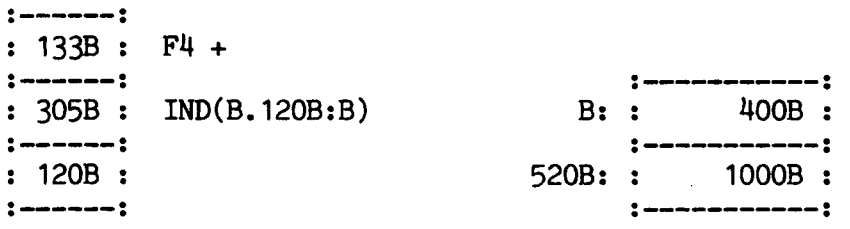


Fig. 8.9 Local indirect addressing

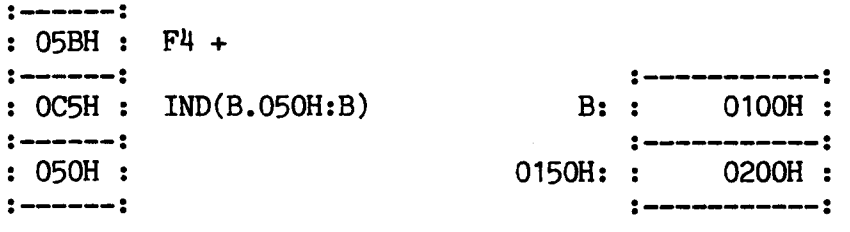
Example:



ea = ((B)+d) = (400B+120B) = 1000B

Octal

Hexadecimal



ea = ((B)+d) = (0100H+050H) = 0200H

8.3.4. Local indirect, post indexed addressing

Assembly notation	Name	Hex code	Octal code
IND(B.<displ>)(Rn)	indirect, post indexed		
IND(B.<displ>:B)(Rn)	indirect, post indexed, byte displacement	0E4H+y	344B+y
IND(B.<displ>:H)(Rn)	indirect, post indexed, halfword displacement	0E8H+y	350B+y
IND(B.<displ>:W)(Rn)	indirect, post indexed, word displacement	0ECH+y	354B+y

ea = ((B)+d) + p*(Rn)

The address is calculated by adding the unsigned displacement of the address code to the content of the base register. This sum is interpreted as an address. The content of the word with this address is added to the content of the specified register multiplied by the post index scaling factor. This sum is the address of the operand. Subroutine array arguments are usually accessed with local indirect, post indexed addressing.

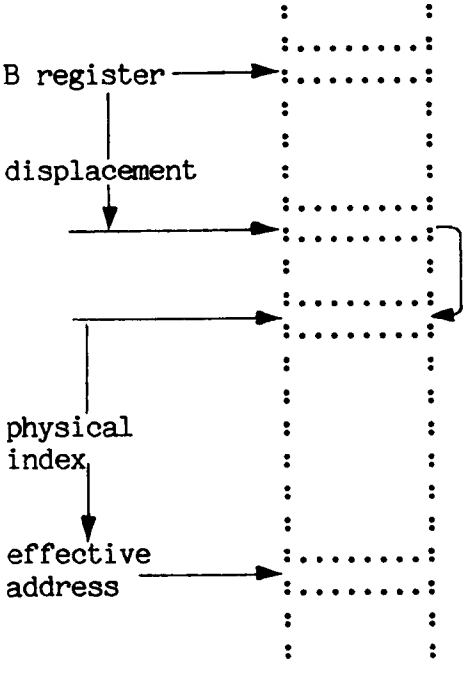


Fig. 8.10 Local indirect, post indexed addressing

Example:

```
:------:                                  :----------:
: 013B :  H4 :=                       B:  :      600B :
:------:                                  :----------:
: 347B :  IND(B.60B)(R4)           660B:  :     2000B :
:------:                                  :----------:
: 060B :                              R4:  :      150B :
:------:                                  :----------:
```

ea = ((B)+d)+p*(Rn) = (660B)+2*150B = 2000B+320B = 2320B

Octal

Hexadecimal

```
:------:                                  :----------:
: 00BH :  H4 :=                       B:  :     0180H :
:------:                                  :----------:
: 0E7H :  IND(B.030H)(R4)         0160H:  :     0400H :
:------:                                  :----------:
: 030H :                              R4:  :      068H :
:------:                                  :----------:
```

ea = ((B)+d)+p*(Rn) = (0160H)+2*068H = 0400H+0D0H = 04D0H

8.3.5. Record addressing

Assembly notation	Name	Hex code	Octal code
R.<displ>	record		
R.<displ>:S	record, short displacement	080H+xx	200B+xx
R.<displ>:B	record, byte displacement	0C9H	311B
R.<displ>:H	record, halfword displacement	0CAH	312B
R.<displ>:W	record, word displacement	0CBH	313B

ea = (R)+d

The address of the operand is calculated by adding the displacement to the content of the record register (R).

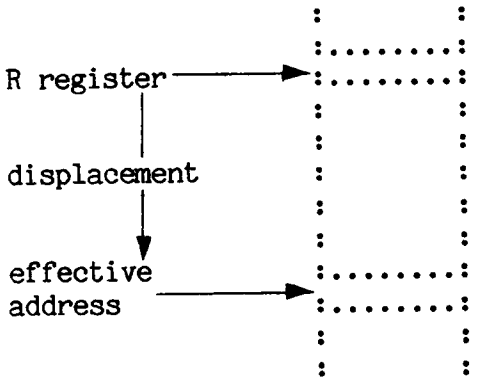
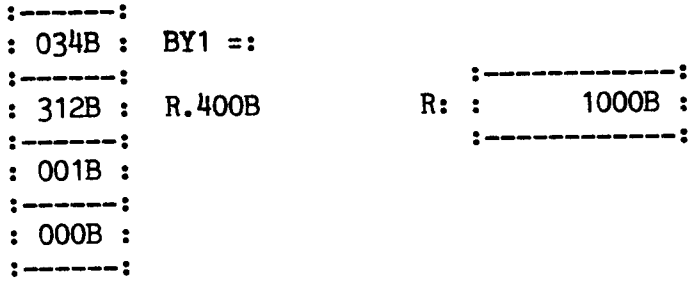


Fig. 8.11 Record addressing

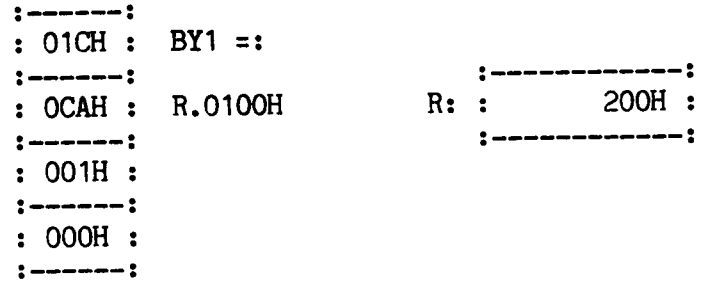
Example:



ea = (B)+d = 1000B+400B = 1400B

Octal

Hexadecimal



ea = (B)+d = 200H+100H = 300H

8.3.6. Pre indexed addressing

Assembly notation	Name	Hex code	Octal code
Rn.<displ>	pre indexed		
Rn.<displ>:B	pre indexed, byte displacement	0F4H+y	364B+y
Rn.<displ>:H	pre indexed, halfword displacement	0F8H+y	370B+y
Rn.<displ>:W	pre indexed, word displacement	0FCH+y	374B+y

ea = (Rn)+d

The contents of the index register specified in the address code is added to the unsigned displacement of the address code. This sum is taken as the address of the operand.

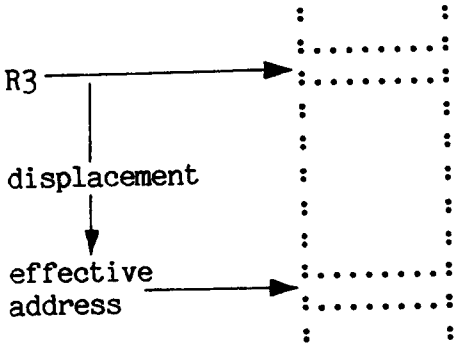
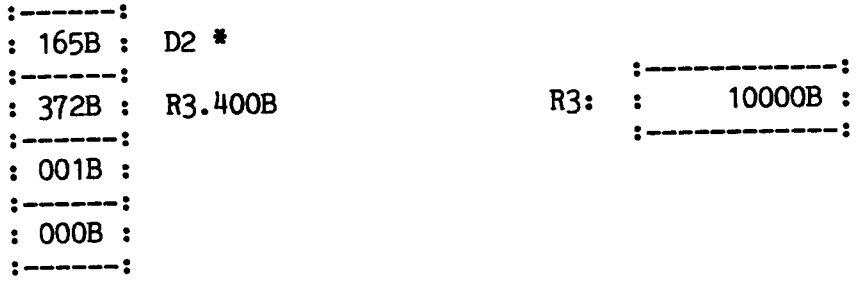


Fig. 8.12 Pre indexed addressing

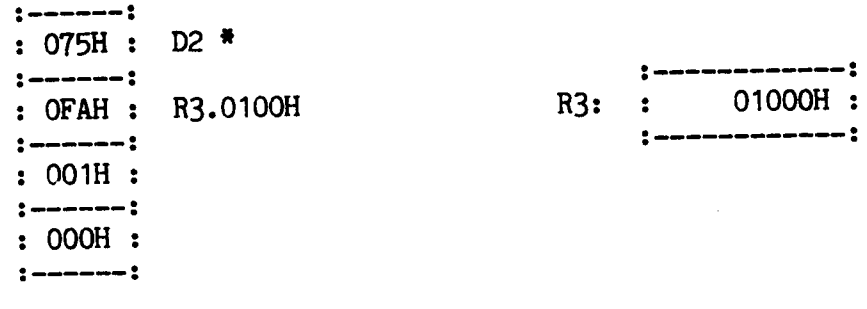
Example:



ea = (Rn)+d = 10000B+400B = 10400B

Octal

Hexadecimal



ea = (Rn)+d = 01000H+0100H = 01100H

8.3.7. Absolute addressing

Assembly notation	Name	Hex code	Octal code
<label>	absolute addressing	0C4H	304B

ea = a

When the address code is equal to 304B, 0C4H, the four bytes following the address code are taken as the address of the operand.

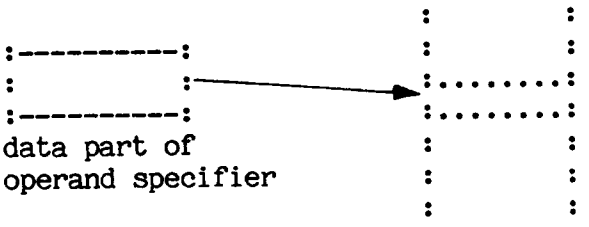


Fig. 8.13 Absolute addressing

Example:

```
:------:
: 165B :  D2 *
:------:
: 304B :  2002044522B
:------:
: 020B :
:------:
: 010B :
:------:
: 111B :
:------:
: 122B :
:------:
```

ea = 2002044522B

Octal

Hexadecimal

```
:------:
: 075H :  D2 *
:------:
: 0C4H :  010084952H
:------:
: 010H :
:------:
: 008H :
:------:
: 049H :
:------:
: 052H :
:------:
```

ea = 010084952H

8.3.8. Absolute, post indexed addressing

Assembly notation	Name	Hex code	Octal code
<label>(Rn)	absolute, post indexed	0E0H+y	340B+y

ea = a+p*(Rn)

The four bytes following the address code are taken as the base address. An absolute, post indexed address is then the content of the index register multiplied by the post index scaling factor and added to the word integer following the address code giving the effective address.

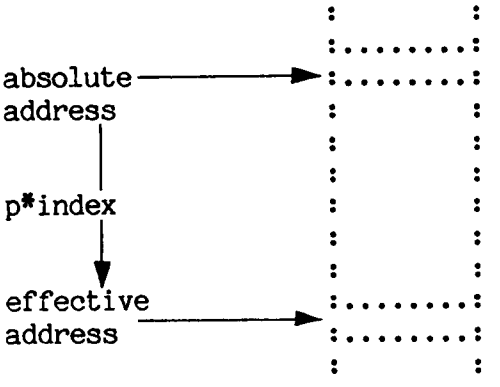
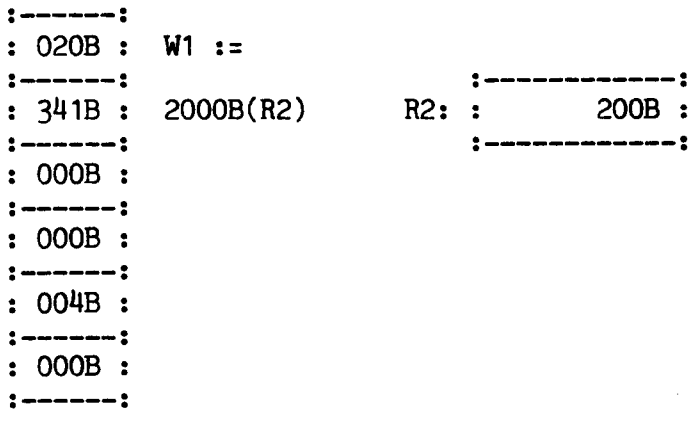


Fig. 8.14 Absolute, post indexed addressing

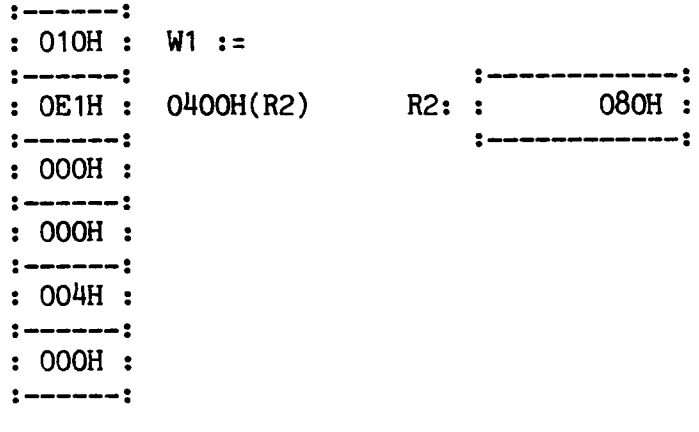
Example:



ea = a+p*(Rn) = 2000B+4*200B = 3000B

Octal

Hexadecimal



ea = a+p*(Rn) = 0400H+4*080H = 0600H

8.3.9. Constant operand addressing

Assembly notation	Name	Hex code	Octal code
<constant>	general constant		
<constant>:S	short constant	000H+xx	000B+xx
<constant>:B	byte constant	0CDH	315B
<constant>:H	halfword constant	0CEH	316B
<constant>:W	word constant	0CFH	317B
<constant>:F	floating point constant	0CFH	317B
<constant>:D	double floating point constant	0CCH	314B

op = data part of operand specifier

The data to be operated on is a part of the operand specifier. It resides in the program memory and can not be modified by any instruction. The value of the operand may have a length of six bits, one, two, four or eight bytes.

Constant operands are illegal for all write instructions, eg. store, swap, or shift instructions, and as the destination operand(s) for multi-operand instructions. They are also illegal as subroutine arguments, as they have no address in data memory.

Note that word and floating point constants have the same address code.

Assembly notation		byte0	byte1	byte2	byte3	byte4
150B:B	Octal:	315B	150B			
	Hex:	0CDH	068H			
1200000:W	Octal:	317B	000B	022B	117B	200B
	Hex:	0CFH	000H	012H	04FH	080H
12B:S	Octal:	012B				
	Hex:	00AH				
6400H:H	Octal:	316B	144B	000B		
	Hex:	0CEH	064H	000H		

Fig. 8.15 Example of constants

The instruction code decides the interpretation of the operand addressed by the operand specifier. This may produce conflicts between the operand interpretation and the size of the data part of constant operands. These are solved by sign extension or data conversion if possible, done automatically by hardware. If no conversion is meaningful an illegal operand specifier trap condition occurs.

The the following abbreviations are used in the table.

IOS	- ILLEGAL OPERAND SPECIFIER TRAP CONDITION
BZ	- bit zero of constant is operand
SX	- sign extended (unless instruction calls for unsigned)
CF	- convert to float
CDF	- convert to double float
NC	- no conversion required
32LZ	- 32 least significant bits zero filled
<c>	- general operand with constant type

Instruction operand type	Constant operand type <c>:S	<c>:B	<c>:H	<c>:W	<c>:F	<c>:D
BI	BZ	IOS	IOS	IOS	IOS	IOS
BY	SX	NC	IOS	IOS	IOS	IOS
H	SX	SX	NC	IOS	IOS	IOS
W	SX	SX	SX	NC	NC	IOS
F	CF	CF	CF	NC	NC	IOS
D	CDF	CDF	CDF	32LZ	32LZ	NC

Fig. 8.16 Treatment of constants as operands

8.3.10. Register addressing

Assembly notation		Name	Hex code	Octal code
Rn	(n=1..4)	Register	0D0H+y	320B+y

One of the registers may be the operand of an instruction. If the data type of an instruction is integer or it does not contain a data type specification, one of the integer registers is taken as the operand. If the data type of the instruction is float or double float, one of the float or double float registers is taken as the operand.

A register operand is not legal in the argument list of a CALL or CALLG instruction, or as destination in the BMOVE instruction.

8.3.11. Alternative addressing

Assembly notation	Name	Hex code	Octal code
ALT(<operand>)	alternative domain addressing	0C8H	310B

With this operand specifier prefix, it is possible to address operands on the alternative domain of the process. Parameter access to the segment on the alternative domain is required. See the memory management section for further explanation of domain, alternative domain and parameter access. <operand> can be any operand specifier that does not contain a new ALT operand specifier prefix. If the operand specifies indirect addressing, the indirect address is taken from the current addressing domain. If the operand specifies descriptor access, the descriptor is taken from the current addressing domain. Only the last memory access which actually fetches the data goes to the alternative addressing domain.

8.3.12. Descriptor addressing

Assembly notation	Name	Hex code	Octal code
DESC(<operand>)(Rn)	descriptor	0F0H+y	360B+y

ea = A + p*(Rn), A = contents of second word of <operand>

<operand> is the address of a descriptor, and it can be any operand specifier not containing an operand specifier prefix. <operand> may be post indexed, in which case the post index scaling factor p is 8 (the size of a descriptor). The post index scaling factor of the descriptor addressing itself is determined by the data type specified in the instruction code.

A descriptor comprises two words in memory accessed via a general operand. The first word contains the length of a data array, the second contains the start address of the array. The operand element of the array is addressed post indexed relative to the start address in the descriptor. The index is incremented and checked against the length in the descriptor. If the index is greater than this length after it is incremented, a descriptor range trap condition will occur.

The hardware will report if the last element of the array is addressed by setting the K flag. If an element beyond the array is addressed the K flag and the descriptor range status bit are set.

The index register is incremented by a data access via descriptor. It is not incremented when accessing only the address of the operand (load address and call instructions).

```
if (Rn)+1 > descriptor.length then
   descriptor range trap condition
endif
if (Rn)+1 > = descriptor.length then
   1 =: status.K
endif
perform addressing with Rn as post index
if data access then
   (Rn+1) =: Rn
endif
```

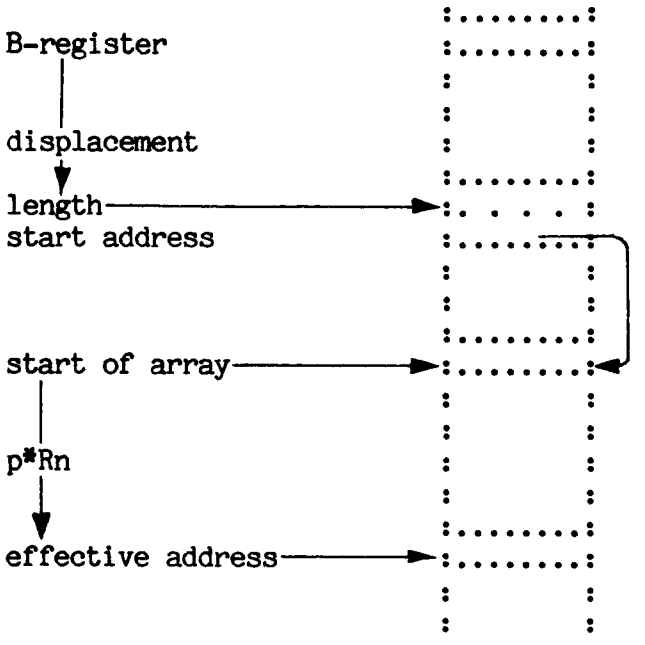
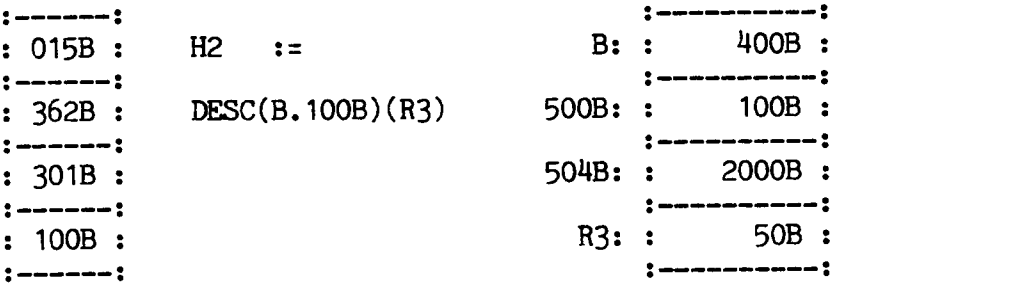


Fig. 8.17 Addressing with a descriptor

In this example the descriptor is addressed locally

OPERAND SPECIFIERS AND ADDRESSING

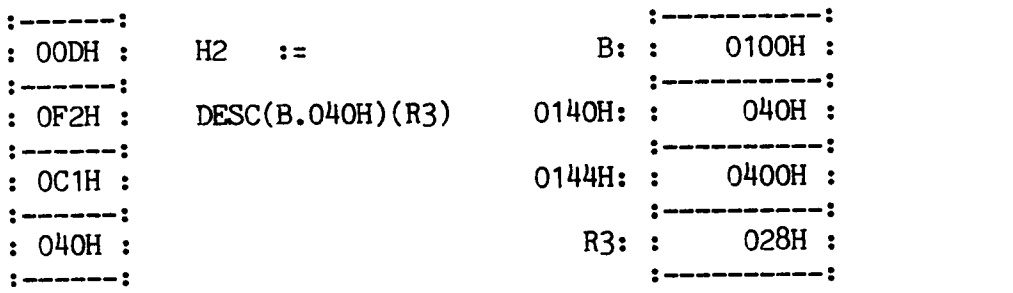
Example:



ea= A + p*(Rn) = (400B+100B+4) + 2*50B = (504B) + 120B = 2120B

Octal

Hexadecimal



ea= A + p*(Rn) = (0100H+040H+4)+2*028H = (0144H)+050H = 0450H

8.4. Direct operands

8.4.1. Introduction

Direct operands are those found in the bytes immediately following the instruction code or the preceding operand specifier. There is no prefix or address code part in the operand specifier. Direct operands are in the syntax definitions in this manual written using the form <<direct operand>>.

The interpretation of a direct operand depends on the instruction and applies to specific instructions only. The data part of the operand specifier is taken either as a displacement or as an absolute address. Absolute addresses may be to the program or the data area; in either case a segment number of zero will be taken as current segment.

8.4.2. Displacement addressing

The NORD-500 instructions LOOP, LOOPI, LOOPD, GO and IF <rel> GO have displacement (program relative) addressing. Each instruction has two instruction codes, one for byte displacement part and one for halfword displacement part. The displacement is signed.

(P) + d -> (P)

8.4.3. Absolute program addressing

The instruction CALL subroutine have absolute addressing. When using CALL the address follows the instruction code in the succeeding four bytes.

When executing CALLG the address is accessed via a general, not a direct, operand. Complete information is given in the description of the CALL instruction.

8.4.4. Absolute data addressing

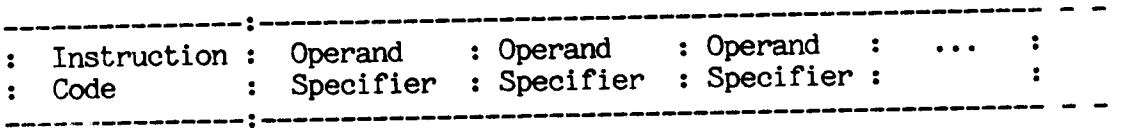
The INIT and ENTM instructions are followed by an absolute address of the bottom of the new stack. The ENTF and ENTFN instructions are followed by the address of the local data area.

9. THE NORD-500 INSTRUCTION SET

9.1. Introduction

The NORD-500 instruction set has a variable length instruction format, the length determined by the type of instruction and the operands used. The shortest instructions are one byte long, the longest may be several thousand bytes long.

Each instruction consists of an instruction code and zero or more operand specifiers. The general instruction format is shown in the figure below:



1 or 2 bytes Zero or more operand specifiers, each 1 to 9 bytes

Fig. 9.1 Instruction format

The following chapters describe each instruction code in detail. Operand specifiers are described in the previous chapter.

The term instruction code is used to indicate both the octal or hexadecimal value and the assembly notation. The octal or hexadecimal value of an instruction code is a numeric representation of the bit pattern inside the NORD-500. The assembly notation is used by the assembler programmer to symbolically represent the binary code.

An instruction code specifies the operation to be performed and the data types of the operands. It may consist of one or two bytes. One byte instruction codes are used for the operations most frequently generated by compilers.

In many NORD-500 instructions one of the general registers or one of the floating point registers are used as argument or result. The two lower bits of the instruction code then specifiy the register number, meaning a floating point or double precision floating point register (Fn or Dn) when the data type is floating or double floating, a general register (Rn) when the data type is integer.

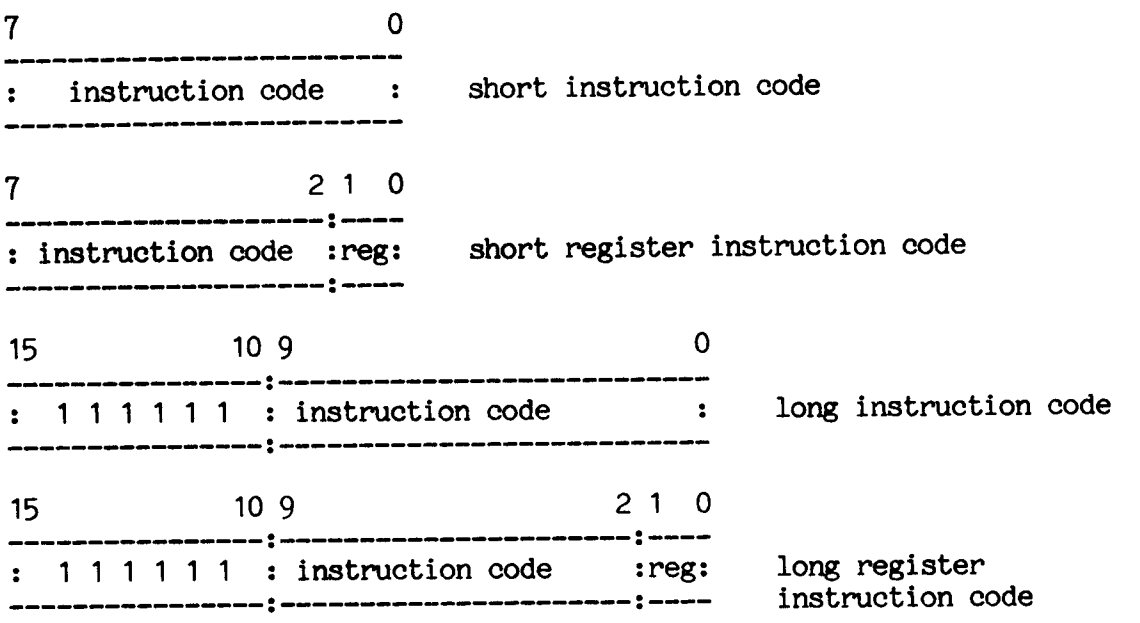


Fig. 9.2 Instruction code formats

All the upper 6 bits of a long (two byte) instruction code are set, which means that such codes are in the range 176000B to 177777B, 0FC00H to 0FFFFH.

The instruction set is described using the syntax explained below. Syntax elements enclosed in brackets, [ ] , are elements used in some instructions, but not all. Brackets followed by an n indicate that more than one occurrence of an optional syntax element may be specified. The sign ::= means "is defined as".

instruction format ::= [[datatype specifier] [register number]]
instruction code name
[operand specifier] [, operand specifier]n

t = data type specifier ::= BI, BY, H, W, F, D
t is a subset of the data type specifiers

n = register number ::= 1,2,3,4

instruction code name ::= text or character string

operand specifier ::= <general operand> <<direct operand>>

<general operand> - the operand is accessed via a general addressing mode

<<direct operand>> - the operand is found in the bytes immediately following the instruction code or the preceding operand specifier

When describing the operand, the description string is divided in three or four parts, as follows:

operand ::= operand name/access code/datatype /pointer register

Operand name is a text string.

The operand name is used as a descriptive term. Eg. the load instruction format uses the term <source> as the operand name; the store instruction format uses <dest> as the destination operand name.

The access code may have the following abbreviations:

r	-	read access
w	-	write access
rw	-	read and write access
rwl	-	locked swap access
aa	-	address access
s	-	special, explained explicitly in the instruction descriptions

Locked swap access applies to the TSET instruction only.

Address access (aa) together with descriptor addressing will not cause the index register to be incremented. If the access code is read (r) or write (w), the index register will be incremented.

The pointer register applies to string instruction descriptions only.

Data status bits not mentioned in the instruction description are always cleared after the instruction has been executed.

Before going on to the instruction set, an example will be explained:

EXAMPLE:

Load bit register number 2 with the bit number found in R3 from the bit array BITA. BITA is displaced 078H, or 170B, bytes from the base address of the local data area. The size of the displacement part is forced to half word.

Assembly code notation: BI2 := B.BITA(R3) : H

Description:

The instruction code for loading bit register 2 is 0FC05H, or 176005B, written as 374B,005B when treated as two octal bytes.

B.BITA(R3) is the local post indexed addressing mode, address code 0DAH, or 332B.

The :H length specifier tells the assembler to store the displacement in halfword format. Normally the assembler should be allowed to select the storage format, in order to achieve optimal program encoding. In this example the assembler would have stored the displacement in byte format if :H had been omitted.

The address of the byte containing the bit in question is calculated as follows (See figure the next page) :

ea = (B) + d + p * (Rn)

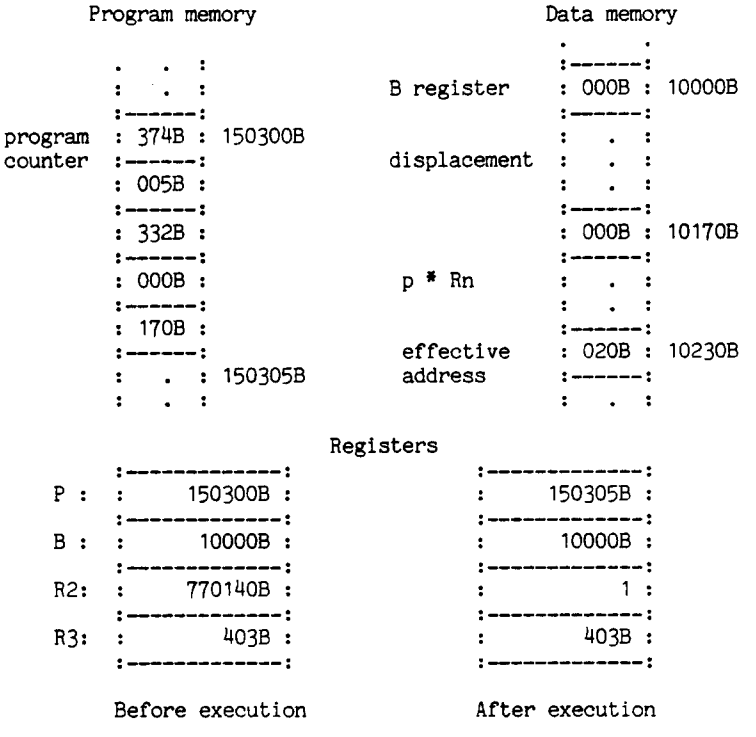
Hex: 01000H + 078H + INT(0103H/08H) = 01098H

Octal: 10000B + 170B + INT(403B/10B) = 10230B

Post indexing always counts the data elements from the left, consequently the bit number within the addressed byte is

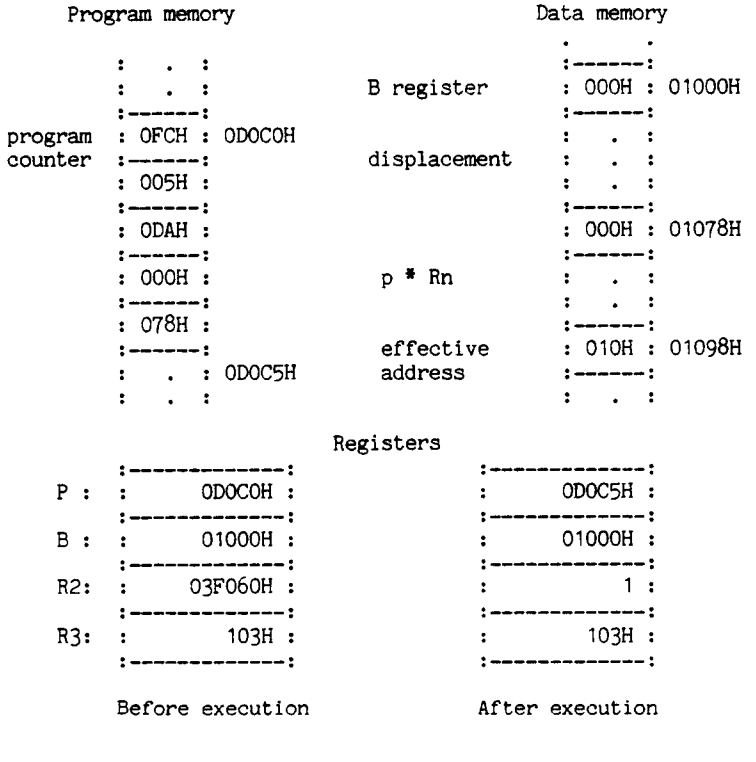
bn = 7-REM(403B/10B) = 7-REM(0103H/08H) = 7-3 = 4

THE NORD-500 INSTRUCTION SET



Octal

Hexadecimal



10. DATA TRANSFER, ARITHMETICAL AND LOGICAL INSTRUCTIONS

10.1. Load

Format: tn := <source/r/t>

Assembly notation	Name	Hex code	Octal code
BIn :=	load bit	0FC04H+(n-1)	176004B+(n-1)
BYn :=	load byte	004H+(n-1)	004B+(n-1)
Hn :=	load halfword	008H+(n-1)	010B+(n-1)
Wn :=	load word	00CH+(n-1)	014B+(n-1)
Fn :=	load float	010H+(n-1)	020B+(n-1)
Dn :=	load double float	014H+(n-1)	024B+(n-1)

Operation: <source> -> Rn

Description:

The value of the operand (source) is loaded into the register specified in the instruction code. When the data type is BI, BY, H or W one of the I registers is loaded. The value is right justified in the register, the least significant bit of the operand in the least significant bit of the register. With BI, BY, or H as data type, the rest of the register is zero filled. One of the floating point registers is loaded when the data type is F or D.

Trap conditions: Addressing traps

Data status bits:

```
<source> = 0      -> Z
<source>.signbit -> S
```

Example:

Load local halfword variable MEMBERS into R3

```
H3 := B.MEMBERS
```

10.2. Load local base register

Format: B := <source/r/W>

Assembly notation	Name	Hex code	Octal code
B :=	load base register	0FC08H	176010B

Operation: <source> -> B

Description:

The contents of <source> is loaded into the local base register.

Trap conditions: Addressing traps

Data status bits:

```
<source> = 0      -> Z
<source>.signbit -> S
```

Example:

Load the word variable GLOBBASE into B

```
B := GLOBBASE
```

10.3. Load record register

Format: R := <source/r/W>

Assembly notation	Name	Hex code	Octal code
R :=	load record register	018H	030B

Operation: <source> -> R

Description:

The contents of <source> is loaded into the record base register.

Trap conditions: Addressing traps

Data status bits:

```
<source> = 0      -> Z
<source>.signbit -> S
```

Example:

Load R with the base of the R2nd element of the word array RECPTRS

```
R := RECPTRS(R2)
```

10.4. Store

Format: tn =: <dest/w/t>

Assembly notation	Name	Hex code	Octal code
BIn =:	store bit	0FC0CH+(n-1)	176014B+(n-1)
BYn =:	store byte	01CH+(n-1)	034B+(n-1)
Hn =:	store halfword	0FC10H+(n-1)	176020B+(n-1)
Wn =:	store word	020H+(n-1)	040B+(n-1)
Fn =:	store float	024H+(n-1)	044B+(n-1)
Dn =:	store double float	028H+(n-1)	050B+(n-1)

Operation:

(Rn) -> <dest>
(datatype dependent part of register) -> <dest>

Description:

The data type dependent part of the specified register is stored in the memory location or register specified in the operand specifier. The data type dependent part of the register is the least significant bits of the register needed to represent the data type in question. Constant operands are illegal. The source register is unaffected.

If the destination is a register, the instruction has the same effect as a load destination register. If the data type is BI, BY, or H the upper part of the register is zero filled.

Trap conditions: Addressing traps

Data status bits:

<datatype dependent part of register> = 0 -> Z
<datatype dependent part of register>.signbit -> S

Example:

Store byte in R4 into the 6th byte of the record pointed to by R and force the displacement part to occupy one word

```
BY4 =: R.6:W
```

10.5. Store local base register

Format: B =: <operand/w/W>

Assembly notation	Name	Hex code	Octal code
B =:	store local base register	0FC0AH	176012B

Operation: B -> <operand>

Description:

The contents of the local base register is stored in the <operand>.

Trap conditions: Addressing traps

Data status bits:

```
(register) = 0      -> Z
(register).signbit -> S
```

Example:

Store B in local variable CURRB indexed by R1

```
B =: B.CURRB(R1)
```

10.6. Store record register

Format: R =: <operand/w/W>

Assembly notation	Name	Hex code	Octal code
R =:	store record register	0FC09H	176011B

Operation: R -> <operand>

Description:

The contents of the record register is stored in the <operand>.

Trap conditions: Addressing traps

Data status bits:

```
(register) = 0      -> Z
(register).signbit -> S
```

Example:

Store R in register R2

```
R =: R2
```

10.7. Move

Format: t MOVE <source/r/t>,<dest/w/t>

Assembly notation	Name	Hex code	Octal code
BI MOVE	move bit	0FC0BH	176013B
BY MOVE	move byte	019H	031B
H MOVE	move halfword	0FC14H	176024B
W MOVE	move word	01AH	032B
F MOVE	move float	01BH	033B
D MOVE	move double float	02CH	054B

Operation: <source> -> <dest>

Description:

The number of bits needed to represent the data type are moved from source to destination. The source is unaffected, and a constant destination operand is illegal.

Trap conditions: Addressing traps

Data status bits:

```
<source> = 0     -> Z
<source>.signbit -> S
```

Example:

Move the double precision value in GLOBAL to local variable LOCAL

```
D  MOVE  GLOBAL, B.LOCAL
```

10.8. Swap

Format: t SWAP <op1/rw/t>,<op2/rw/t>

Assembly notation	Name	Hex code	Octal code
BI SWAP	bit swap	0FCBDH	176275B
BY SWAP	byte swap	0FCBEH	176276B
H SWAP	halfword swap	0FCBFH	176277B
W SWAP	word swap	052H	122B
F SWAP	float swap	0FCDCH	176334B
D SWAP	double float swap	0FCDDH	176335B

Operation: <op1> :=: <op2>

Description:

The contents of the first operand is stored in the second, and the original contents of the second operand is stored in the first. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps

Data status bits:

```
<op1>.original contents = 0      -> Z
<op1>.original contents.signbit -> S
```

Example:

Exchange contents of word variables EAST and WEST

```
W SWAP EAST, WEST
```

10.9. Compare

Format: tn COMP <operand/r/t>

Assembly notation		Name	Hex code	Octal code
BIn	COMP	register bit compare	0FC18H+(n-1)	176030B+(n-1)
BYn	COMP	register byte compare	030H+(n-1)	060B+(n-1)
Hn	COMP	register halfword compare	0FC1CH+(n-1)	176034B+(n-1)
Wn	COMP	register word compare	034H+(n-1)	064B+(n-1)
Fn	COMP	register float compare	038H+(n-1)	070B+(n-1)
Dn	COMP	register double float compare	03CH+(n-1)	074B+(n-1)

Operation: Rn - <operand>

Description:

The compare instruction subtracts the operand from the specified register. The result of the subtraction is not saved, but rather compared to zero, and this result is saved in the data status bits. The instruction is a true comparison, hence the sign bit is changed in case of integer overflow.

Trap conditions: Addressing traps, floating underflow, floating overflow

Data status bits:

```
<result> = 0                      -> Z
<result>.signbit XOR Overflow     -> S
carry from most significant bit   -> C
(floating underflow)              -> FU
(floating overflow)               -> FO
```

Example:

Compare bit zero in R1 with one

```
BI1 COMP 1
```

10.10. Compare two operands

Format: t COMP2 <1st operand/r/t>,<2nd operand/r/t>

Assembly notation	Name	Hex code	Octal code
BI COMP2	bit compare	0FC15H	176025B
BY COMP2	byte compare	02DH	055B
H COMP2	halfword compare	0FC16H	176026B
W COMP2	word compare	02EH	056B
F COMP2	float compare	02FH	057B
D COMP2	double float compare	040H	100B

Operation: <1st operand> - <2nd operand>

Description:

Compare two operands subtracts the second operand from the first. The result sets the data status bits accordingly, but the result is otherwise discarded.

Trap conditions: Addressing traps, floating underflow, floating overflow

Data status bits:

```
<result> = 0                          -> Z
<result>.signbit XOR Overflow         -> S
carry from most significant bit       -> C
(floating underflow)                  -> FU
(floating overflow)                   -> FO
```

Example:

Compare record variable floating point DELTA with 0.005

```
F  COMP2 R.DELTA, 0.005
```

10.11. Test against zero

Format: t TEST <operand/r/t>

Assembly notation	Name	Hex code	Octal code
BI TEST	bit test against zero	041H	101B
BY TEST	byte test against zero	042H	102B
H TEST	halfword test against zero	043H	103B
W TEST	word test against zero	044H	104B
F TEST	float test against zero	045H	105B
D TEST	double test against zero	046H	106B

Operation: <operand> - 0

Description:

This instruction is similar to comparing two operands except that the second operand is implicitly zero.

Trap conditions: Addressing traps

Data status bits:

```
<result> = 0                      -> Z
<result>.signbit XOR Overflow     -> S
1                                 -> C  (integer)
```

Example:

Test if local byte variable COUNTER has reached zero

```
BY TEST B.COUNTER
```

10.12. Negate

Format: tn NEG

Assembly notation	Name	Hex code	Octal code
BYn NEG	byte register negate	0FE08H+(n-1)	177010B+(n-1)
Hn NEG	halfword register negate	0FE0CH+(n-1)	177014B+(n-1)
Wn NEG	word register negate	090H+(n-1)	220B+(n-1)
Fn NEG	float register negate	094H+(n-1)	224B+(n-1)
Dn NEG	double float register negate	094H+(n-1)	224B+(n-1)

Operation: -Rn -> Rn

Description:

The contents of the specified register is negated. An integer value is negated by taking the two's complement of its value. A floating point value is negated by inverting its sign bit. Byte and halfword negate will clear the upper part of the register.

Integer overflow occurs if and only if the greatest negative integer is negated. Carry is zero except when integer zero is negated.

Trap conditions: Integer overflow

Data status bits:

```
(negated register) = 0      -> Z
(negated register).signbit  -> S
(carry)                     -> C
(overflow)                  -> O
```

Example:

Negate double precision register D3

```
D3 NEG
```

10.13. Invert

Format: tn INV

Assembly notation		Name	Hex code	Octal code
BIn	INV	bit invert register	0FE10H+(n-1)	177020B+(n-1)
BYn	INV	byte invert register	0FE14H+(n-1)	177024B+(n-1)
Hn	INV	halfword invert register	0FE18H+(n-1)	177030B+(n-1)
Wn	INV	word invert register	098H+(n-1)	230B+(n-1)

Operation: One's complement of Rn -> Rn

Description:

The one's complement of the specified register is calculated and stored in the same register. When the datatype is BI, BY, or H only the lower part of the register is complemented and the rest of the register is cleared.

Trap conditions: None

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
```

Example:

Invert the lowermost bit of R4 and clear the upper 31 bits

```
BI4 INV
```

10.14. Invert with carry add

Format: Wn INVC

Assembly notation	Name	Hex code	Octal code
Wn INVC	word invert register w/carry	0FF10H+(n-1)	177420B+(n-1)

Operation: One's complement of Rn + C -> Rn

Description:

The one's complement of the specified word register is calculated. The carry is added and the result is loaded into the specified register. This instruction is used for multiple precision arithmetic.

Trap conditions: Integer overflow

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
(carry)           -> C
(overflow)        -> O
```

Example:

Invert W2 and add carry

```
W2 INVC
```

10.15. Absolute value

Format: tn ABS

Assembly notation	Name	Hex code	Octal code
BYn ABS	byte absolute value	0FF00H+(n-1)	177400B+(n-1)
Hn ABS	halfword absolute value	0FF04H+(n-1)	177404B+(n-1)
Wn ABS	word absolute value	0FF08H+(n-1)	177410B+(n-1)
Fn ABS	float absolute value	0FF0CH+(n-1)	177414B+(n-1)
Dn ABS	double float absolute value	0FF0CH+(n-1)	177414B+(n-1)

Operation: Absolute value of Rn -> Rn

Description:

The absolute value of the specified register is calculated and stored in the same register. When the datatype is either BY or H, the result is stored in the least significant bits and the rest of the register is cleared. Overflow occurs if and only if the greatest negative integer is negated.

Trap conditions: Integer overflow

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
(overflow)        -> O  (integer)
```

Example:

Take the absolute value of double precision register D1

```
D1  ABS
```

10.16. Add

Format: tn + <addend/r/t>

Assembly notation	Name	Hex code	Octal code
BYn +	byte add	0FC34H+(n-1)	176064B+(n-1)
Hn +	halfword add	0FC38H+(n-1)	176070B+(n-1)
Wn +	word add	054H+(n-1)	124B+(n-1)
Fn +	floating add	058H+(n-1)	130B+(n-1)
Dn +	double float add	05CH+(n-1)	134B+(n-1)

Operation: Rn + <addend> -> Rn

Description:

The <addend> operand is added to the contents of the specified register. The carry bit is set if a carry occurs from the sign bit position of the adder, otherwise it is reset. For overflow, see the section on arithmetical traps.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<sum>.signbit          -> S
<sum> = 0              -> Z
0                      -> O (float)
(overflow)             -> O (integer)
0                      -> C (float)
(carry from most significant bit) -> C (integer)
(floating underflow) -> FU
(floating overflow)  -> FO
```

Example:

Add byte argument FIFTHARG to R3

```
BY3  +  IND(B.FIFTHARG)
```

10.17. Subtract

Format: tn - <subtrahend/r/t>

Assembly notation	Name	Hex code	Octal code
BYn -	byte subtract	0FC3CH+(n-1)	176074B+(n-1)
Hn -	halfword subtract	0FC40H+(n-1)	176100B+(n-1)
Wn -	word subtract	060H+(n-1)	140B+(n-1)
Fn -	float subtract	064H+(n-1)	144B+(n-1)
Dn -	double float subtract	068H+(n-1)	150B+(n-1)

Operation: Rn - <subtrahend> -> Rn

Description:

The <subtrahend> operand is subtracted from the specified register. The same rules as for ADD apply for the setting of the carry bit. For overflow, see section on arithmetical traps.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

<difference> = 0	-> Z
<difference>.signbit	-> S
(overflow)	-> O (integer)
0	-> O (float)
(carry from the most significant bit)	-> C
(floating underflow)	-> FU
(floating overflow)	-> FO

Example:

Subtract register F1 from register F4

```
F4 - F1
```

10.18. Multiply

Format: tn * <multiplier/r/t>

Assembly notation	Name	Hex code	Octal code
BYn *	byte multiply	0FC44H+(n-1)	176104B+(n-1)
Hn *	halfword multiply	0FC48H+(n-1)	176110B+(n-1)
Wn *	word multiply	06CH+(n-1)	154B+(n-1)
Fn *	floating multiply	070H+(n-1)	160B+(n-1)
Dn *	double float multiply	074H+(n-1)	164B+(n-1)

Operation: Rn * <multiplier> -> Rn

Description:

The <multiplier> operand is multiplied by the specified register with the product stored in this register. Integer overflow occurs if the upper half of the double length result is not equal to the sign extension of the lower half.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<product> = 0          -> Z
<product>.signbit      -> S
(overflow)             -> O (integer)
0                      -> O (float)
(floating underflow)   -> FU
(floating overflow)    -> FO
```

Example:

Multiply halfword register R2 with 5

```
H2 * 5
```

10.19. Divide

Format: tn / <divisor/r/t>

Assembly notation	Name	Hex code	Octal code
BYn /	byte divide	0FC4CH+(n-1)	176114B+(n-1)
Hn /	halfword divide	0FC50H+(n-1)	176120B+(n-1)
Wn /	word divide	078H+(n-1)	170B+(n-1)
Fn /	float divide	07CH+(n-1)	174B+(n-1)
Dn /	double float divide	0E8H+(n-1)	350B+(n-1)

Operation: Rn / <divisor> -> Rn

Description:

The contents of the specified register is divided by the <divisor> operand. The quotient is left in the same register. In integer division the remainder (unless it is zero) has the same sign as the register contents, ie. the quotient is truncated towards 0. Integer overflow occurs if and only if the largest possible negative integer is divided by -1.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow, divide by zero

Data status bits:

```
<quotient> = 0          -> Z
<quotient>.signbit      -> S
(overflow)              -> 0 (integer)
0                       -> 0 (float)
(floating underflow)    -> FU
(floating overflow)     -> FO
<divisor> = 0           -> DZ
```

Example:

Divide float register A3 with the R4th element of argument ARR

```
F3 / IND(B.ARR)(R4)
```

10.20. Add two operands

Format: t ADD2 <a/rw/t>,<b/r/t>

Assembly notation	Name	Hex code	Octal code
BY ADD2	byte add two operands	0FC17H	176027B
H ADD2	halfword add two operands	0FC54H	176124B
W ADD2	word add two operands	053H	123B
F ADD2	float add two operands	0FC56H	176126B
D ADD2	double float add two operands	0FC57H	176127B

Operation: <a> + <b> -> <a>

Description:

The <b> operand is added to the <a> operand and the result is put in the <a> operand. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<result> = 0                -> Z
<result>.signbit            -> S
0                           -> O (float)
(overflow)                  -> O (integer)
0                           -> C (float)
(carry from most significant bit) -> C (integer)
(floating underflow)        -> FU
(floating overflow)         -> FO
```

Example:

Add float argument X2 to argument X1

```
F ADD2 IND(B.X1), IND(B.X2)
```

10.21. Subtract two operands

Format: t SUB2 <a/rw/t>,<b/r/t>

Assembly notation	Name	Hex code	Octal code
BY SUB2	byte subtract two operands	0FC58H	176130B
H SUB2	halfword subtract two operands	0FC59H	176131B
W SUB2	word subtract two operands	0E0H	340B
F SUB2	float subtract two operands	0FC5BH	176133B
D SUB2	double float subtract two operands	0FC5CH	176134B

Operation: <a> - <b> -> <a>

Description:

The <b> operand is subtracted from the <a> operand and the result is put in the <a> operand. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<difference> = 0      -> Z
<difference>.signbit -> S
0                     -> C (float)
(overflow)            -> O (integer)
0                     -> O (float)
(carry from most significant bit) -> C (integer)
(floating underflow) -> FU
(floating overflow)  -> FO
```

Example:

Subtract 4 from the R3rd element of the byte array whose descriptor is the global VALUES

```
BY SUB2 DESC(VALUES) (R3),4
```

10.22. Multiply two operands

Format: t MUL2 <a/rw/t>,<b/r/t>

Assembly notation	Name	Hex code	Octal code
BY MUL2	byte multiply two operands	0FC5DH	176135B
H MUL2	halfword multiply two operands	0FC5EH	176136B
W MUL2	word multiply two operands	0FC5FH	176137B
F MUL2	float multiply two operands	0FC60H	176140B
D MUL2	double float multiply two operands	0FC61H	176141B

Operation: <a> * <b> -> <a>

Description:

The <a> operand is multiplied by the <b> operand and the product is stored in the <a> operand. Integer overflow occurs if the upper half of the double length result is not equal to the sign extension of the lower half.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<product> = 0          -> Z
<product>.signbit      -> S
(overflow)             -> O (integer)
0                      -> O (float)
(floating underflow)   -> FU
(floating overflow)    -> FO
```

Example:

Multiply the argument double float PROD on the alternative domain with the contents of D4

```
D MUL2 ALT(B.PROD), D4
```

10.23. Divide two operands

Format: t DIV2 <a/rw/t>,<b/r/t>

Assembly notation	Name	Hex code	Octal code
BY DIV2	byte divide two operands	0FC62H	176142B
H DIV2	halfword divide two operands	0FC63H	176143B
W DIV2	word divide two operands	0FC64H	176144B
F DIV2	float divide two operands	0FC65H	176145B
D DIV2	double float divide two operands	0FC66H	176146B

Operation: <a> / <b> -> <a>

Description:

The <a> operand is divided by the <b> operand and the quotient is stored in the <a> operand. In integer division the remainder (unless it is zero) has the same sign as the <a> operand, ie. the quotient is truncated towards zero. Integer overflow occurs if and only if the largest possible negative integer is divided by -1.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow, divide by zero

Data status bits:

```
<quotient> = 0          -> Z
<quotient>.signbit      -> S
(overflow)              -> O (integer)
0                       -> O (float)
(floating underflow)    -> FU
(floating overflow)     -> FO
<divisor> = 0           -> DZ
```

Example:

Divide the local float variable KVOT with the R1st element of the array on the alternative domain described by local descriptor LIST

```
F DIV2 B.KVOT, ALT(DESC(B.LIST)(R1))
```

10.24. Add three operands

Format: t ADD3 <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation	Name	Hex code	Octal code
BY ADD3	byte add three operands	0FC67H	176147B
H ADD3	halfword add three operands	0FC68H	176150B
W ADD3	word add three operands	0FC69H	176151B
F ADD3	float add three operands	0FC6AH	176152B
D ADD3	double float add three operands	0FC6BH	176153B

Operation: <a> + <b> -> <c>

Description:

The <a> operand is added to the <b> operand and the result is stored in the <c> operand. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<sum> = 0                -> Z
<sum>.signbit            -> S
0                        -> O (float)
(overflow)               -> O (integer)
0                        -> C (float)
(carry from most significant bit) -> C (integer)
(floating underflow) -> FU
(floating overflow)  -> FO
```

Example:

Add R1 and R2 leaving the result in R3

```
W ADD3 R1,R2,R3
```

10.25. Subtract three operands

Format: t SUB3 <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation	Name	Hex code	Octal code
BY SUB3	byte subtract three operands	0FC6CH	176154B
H SUB3	halfword subtract three operands	0FC6DH	176155B
W SUB3	word subtract three operands	0FC6EH	176156B
F SUB3	float subtract three operands	0FC6FH	176157B
D SUB3	double float subtract three operands	0FC70H	176160B

Operation: <a> - <b> -> <c>

Description:

The <b> operand is subtracted from the <a> operand and the result is stored in the <c> operand. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<difference> = 0      -> Z
<difference>.signbit -> S
0                     -> O (float)
(overflow)            -> O (integer)
0                     -> C (float)
(carry from most significant bit) -> C (integer)
(floating underflow) -> FU
(floating overflow)  -> FO
```

Example:

Store the difference between byte arguments X1 and X2 in local variable DIFF

```
B SUB3 IND(B.X1), IND(B.X2), B.DIFF
```

10.26. Multiply three operands

Format: t MUL3 <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation	Name	Hex code	Octal code
BY MUL3	byte multiply three operands	0FC71H	176161B
H MUL3	halfword multiply three operands	0FC72H	176162B
W MUL3	word multiply three operands	0FC73H	176163B
F MUL3	float multiply three operands	0FC74H	176164B
D MUL3	double float multiply three operands	0FC75H	176165B

Operation: <a> * <b> -> <c>

Description:

The <a> operand is multiplied by the <b> operand and the product is stored in the <c> operand. Integer overflow occurs if the upper half of the double length result is not equal to the sign extension of the lower half. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
<product> = 0          -> Z
<product>.signbit      -> S
0                      -> C
(overflow)             -> O (integer)
0                      -> O (float)
(floating underflow)   -> FU
(floating overflow)    -> FO
```

Example:

The product of the second and third element of the word array pointed to by R2 is stored in the first element

```
W MUL3 R2.2, R2.3, R2.1
```

10.27. Divide three operands

Format: t DIV3 <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation	Name	Hex code	Octal code
BY DIV3	byte divide three operands	0FC76H	176166B
H DIV3	halfword divide three operands	0FC77H	176167B
W DIV3	word divide three operands	0FC78H	176170B
F DIV3	float divide three operands	0FC79H	176171B
D DIV3	double float divide three operands	0FC7AH	176172B

Operation: <a> / <b> -> <c>

Description:

The <a> operand is divided by the <b> operand and the quotient is stored in the <c> operand. In integer division the remainder (unless it is zero) has the same sign as the <a> operand, ie. the quotient is truncated towards zero. Integer overflow occurs if and only if the largest possible negative integer is divided by -1. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow, divide by zero

Data status bits:

```
<quotient> = 0          -> Z
<quotient>.signbit      -> S
0                       -> C
(overflow)              -> O (integer)
0                       -> O (float)
(floating underflow)    -> FU
(floating overflow)     -> FO
<divisor> = 0           -> DZ
```

Example:

Divide the float value whose address is in PTR with the contents of F1, storing the quotient in record variable Q (record base in R)

```
F DIV3 IND(PTR), F1, R.Q
```

10.28. Multiply with overflow to register

Format: tn MUL4 <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation		Name	Hex code	Octal code
BYn	MUL4	byte multiply w/overflow	0FC20H+(n-1)	176040B+(n-1)
Hn	MUL4	halfword multiply w/overflow	0FC24H+(n-1)	176044B+(n-1)
Wn	MUL4	word multiply w/overflow	0FC28H+(n-1)	176050B+(n-1)

Operation: <a> * <b> -> <c>;
(overflow part) -> Rn

Description:

The <a> operand is multiplied by the <b> operand. The product is stored in the <c> operand. The upper half of the double length result is stored in the specified register. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, integer overflow

Data status bits:

(lower part of double length result) = 0 -> Z
(lower part of double length result).signbit -> S
(overflow) -> O

Example:

Multiply word arguments M and N and store product in local TEMP and the overflow in R1

```
W1 MUL4  IND(B.M), IND(B.N), B.TEMP
```

10.29. Divide with remainder to register (modulo)

Format: tn DIV4 <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation		Name	Hex code	Octal code
BYn	DIV4	byte divide w/remainder	0FC2CH+(n-1)	176054B+(n-1)
Hn	DIV4	halfword divide w/remainder	0FC30H+(n-1)	176060B+(n-1)
Wn	DIV4	word divide w/remainder	0FC7CH+(n-1)	176174B+(n-1)

Operation:

<a> / <b> -> <c>
(remainder) -> Rn

Description:

The <a> operand is divided by the <b> operand with the quotient stored in the <c> operand. The remainder is stored in the specified register. The operands are assumed to have the same data type (see chapter 7.3 page 53).

Trap conditions: Addressing traps, divide by zero

Data status bits:

<quotient> = 0 -> Z
<quotient>.signbit -> S
(overflow) -> O
<divisor> = 0 -> DZ

Example:

Divide record variable BYTECOUNT by 4 and store the quotient in record variable WORDCOUNT with the quotient in R2

```
BY2 DIV4 R.BYTECOUNT, 4, WORDCOUNT
```

10.30. Unsigned multiply with overflow to register

Format: Wn UMUL <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation	Name	Hex code	Octal code
Wn UMUL	word unsigned multiply	0FC80H+(n-1)	176200B+(n-1)

Operation:

word unsigned multiplication
<a> * <b> -> <c>
(overflow part) -> Rn

Description:

The operands are treated as unsigned.
The <a> operand is multiplied by the <b> operand with the product stored in the <c> operand. The upper half of the double length result is stored in the specified register. Byte and halfword integer constants are sign extended and the result of the sign extension is treated unsigned. Integer overflow occurs when the upper part is different from the sign extension of the lower half.

Trap conditions: Addressing traps, integer overflow

Data status bits:

(product) = 0 -> Z
(product).signbit -> S
(overflow) -> O

Example:

Multiply local variable LEASTX with local LEASTY storing the result in R2 with the upper half of the result in R1

W1 UMUL B.LEASTX, B.LEASTY, R2

10.31. Unsigned divide

Format: Wn UDIV <a/r/t>,<b/r/t>,<c/w/t>

Assembly notation	Name	Hex code	Octal code
Wn UDIV	word unsigned divide	0FE48H+(n-1)	177110B+(n-1)

Operation:

word unsigned division
<a> / <b> -> <c>
(remainder) -> Rn

Description:

The operands are treated as unsigned.
The <a> operand is divided by the <b> operand and the quotient is stored in the <c> operand. The remainder is stored in the specified register. Byte and halfword integer constants are sign extended and the result of the sign extension is treated unsigned.

Trap conditions: Addressing traps, divide by zero

Data status bits:

(quotient) = 0 -> Z
(quotient).signbit -> S
<divisor) = 0 -> DZ

Example:

Divide the arguments LONG and FACT on the alternative domain and leave the result in the address on the alternative domain contained in RES, and the remainder in R3

```
W3 UDIV ALT(B.LONG), ALT(B.FACT), ALT(IND(RES))
```

10.32. Add with carry

Format: Wn ADDC <addend/r/t>

Assembly notation	Name	Hex code	Octal code
Wn ADDC	word add with carry	0FE40H+(n-1)	177100B+(n-1)

Operation: Rn + C + <addend> -> Rn

Description:

<addend> and Carry are added to the specified register and the result is stored in this register. This instruction is used for multiple precision arithmetic.

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
<sum> = 0          -> Z
<sum>.signbit      -> S
(integer overflow) -> 0
(carry from most significant bit) -> C
```

Example:

Add variable MOST to R2 with carry

```
W2 ADDC MOST
```

10.33. Subtract with carry

Format: Wn SUBC <subtrahend/r/t>

Assembly notation	Name	Hex code	Octal code
Wn SUBC	word subtract with carry	0FE44H+(n-1)	177104B+(n-1)

Operation: Rn + C - <subtrahend> -1 -> Rn

Description:

Carry and the one's complement of <subtrahend> is added to the specified register. The result is then stored in the specified register. This instruction is used for multiple precision arithmetic.

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
(result) = 0          -> Z
(result).signbit      -> S
(carry)               -> C
(integer overflow)    -> O
```

Example:

Subtract 400 hexadecimal from W2

```
W2 SUBC  0400H
```

10.34. Clear register

Format: tn CLR

Assembly notation	Name	Hex code	Octal code
BIn CLR	bit register clear	084H+(n-1)	204B+(n-1)
BYn CLR	byte register clear	084H+(n-1)	204B+(n-1)
Hn CLR	halfword register clear	084H+(n-1)	204B+(n-1)
Wn CLR	word register clear	084H+(n-1)	204B+(n-1)
Fn CLR	float register clear	088H+(n-1)	210B+(n-1)
Dn CLR	double float register clear	08CH+(n-1)	214B+(n-1)

Operation: 0 -> Rn

Description:

The register is set to all zeros.

Trap conditions: None

Data status bits: 1 -> Z

Example:

Clear double register D3

```
    D3 CLR
```

10.35. Store zero

Format: t STZ <operand/w/t>

Assembly notation	Name	Hex code	Octal code
BI STZ	bit store zero	0FC85H	176205B
BY STZ	byte store zero	048H	110B
H STZ	halfword store zero	049H	111B
W STZ	word store zero	04AH	112B
F STZ	float store zero	04BH	113B
D STZ	double float store zero	04CH	114B

Operation: 0 -> <operand>

Description:

The contents of the destination operand is replaced by zero.

Trap conditions: Addressing traps

Data status bits: 1 -> Z

Example:

Clear the byte FLAGS

```
    BY  STZ FLAGS
```

10.36. Set to one

Format: t SET1 <operand/w/t>

Assembly notation	Name	Hex code	Octal code
BI SET1	bit set to one	0FC86H	176206B
BY SET1	byte set to one	0FC87H	176207B
H SET1	halfword set to one	0FC88H	176210B
W SET1	word set to one	04DH	115B
F SET1	float set to one	047H	107B
D SET1	double float set to one	0FC89H	176211B

Operation: 1 -> <operand>

Description:

The contents of the destination operand is replaced by one.

Trap conditions: Addressing traps

Data status bits: All cleared

Example:

Set float argument START to one

```
    F SET1 IND(B.START)
```

10.37. Increment

Format: t INCR <operand/rw/t>

Assembly notation	Name	Hex code	Octal code
BY INCR	byte increment	0FC8AH	176212B
H INCR	halfword increment	04EH	116B
W INCR	word increment	04FH	117B
F INCR	float increment	050H	120B
D INCR	double float increment	0FC8BH	176213B

Operation: <operand> + 1 -> <operand>

Description:

The <operand> is incremented by one. The Carry bit is set if a carry occurs from the sign bit position of the adder, otherwise it is reset. Carry will occur if and only if integer -1 is incremented.

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
<sum>.signbit -> S
<sum> = 0     -> Z
0             -> O (float)
(overflow)    -> O (integer)
0             -> C (float)
(carry from most significant bit) -> C (integer)
```

Example:

Increment the halfword record variable LOOPER and force the displacement part to occupy a halfword

```
H  INCR R.LOOPER:H
```

10.38. Decrement

Format: t DECR <operand/rw/t>

Assembly notation	Name	Hex code	Octal code
BY DECR	byte decrement	0FC86H	176214B
H DECR	halfword decrement	0FC87H	176215B
W DECR	word decrement	051H	121B
F DECR	float decrement	0FC88H	176216B
D DECR	double float decrement	0FC89H	176217B

Operation: <operand> - 1 -> <operand>

Description:

The <operand> is decremented by one.

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
<difference> = 0      -> Z
<difference>.signbit -> S
(overflow)           -> 0 (integer)
0                    -> 0 (float)
(carry from the most significant bit) -> C
```

Example:

Decrement the halfword record variable STEP on the alternative domain

```
H DECR ALT(R.STEP)
```

10.39. And

Format: tn AND <operand/r/t>

Assembly notation		Name	Hex code	Octal code
BIn	AND	bit 'and' register	0FDCCH+(n-1)	176714B+(n-1)
BYn	AND	byte 'and' register	0FC90H+(n-1)	176220B+(n-1)
Hn	AND	halfword 'and' register	0FC94H+(n-1)	176224B+(n-1)
Wn	AND	word 'and' register	0E4H+(n-1)	344B+(n-1)

Operation: Rn AND <operand> -> Rn

Description:

A bitwise AND is performed between the specified register and the <operand> and the result is stored in the register. When the data type is BI, BY, or H, the upper part of the register is zero filled.

Trap conditions: Addressing traps

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
```

Example:

AND operation between R2 and the R3rd element of the array described by the R1st array descriptor in the local array MASKS

```
W2 AND DESC(B.MASKS(R1))(R3)
```

10.41. Exclusive or

Format: tn XOR <operand/r/t>

Assembly notation		Name	Hex code	Octal code
BIn	XOR	bit 'xor' register	0FDCCH+(n-1)	176714B+(n-1)
BYn	XOR	byte 'xor' register	0FCA0H+(n-1)	176240B+(n-1)
Hn	XOR	halfword 'xor' register	0FCA4H+(n-1)	176244B+(n-1)
Wn	XOR	word 'xor' register	0A4H+(n-1)	244B+(n-1)

Operation: Rn XOR <operand> -> Rn

Description:

A bitwise exclusive OR is performed between the specified register and the <operand> and the result is stored in the register. When the data type is BI, BY, or H, the upper part of the register is zero filled.

Trap conditions: Addressing traps

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
```

Example:

Flip bits 0, 4, 8 and 12 of halfword register R4

```
H4 XOR 01111H
```

10.40. Or

Format: tn OR <operand/r/t>

Assembly notation	Name	Hex code	Octal code
BIn OR	bit 'or' register	0FDF8H+(n-1)	176770B+(n-1)
BYn OR	byte 'or' register	0FC98H+(n-1)	176230B+(n-1)
Hn OR	halfword 'or' register	0FC9CH+(n-1)	176234B+(n-1)
Wn OR	word 'or' register	0A0H+(n-1)	240B+(n-1)

Operation: Rn OR <operand> -> Rn

Description:

A bitwise OR is performed between the specified register and the <operand> and the result is stored in the register. When the data type is BI, BY, or H, the upper part of the register is zero filled.

Trap conditions: Addressing traps

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
```

Example:

OR byte register R1 with 111 octal

```
BY1 OR 111B
```

10.42. Logical shift

Format: t SHL <operand/rw/t>,<shiftcount/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SHL	byte shift logically	0FCA8H	176250B
H SHL	halfword shift logically	0FCA9H	176251B
W SHL	word shift logically	0FCAAH	176252B

Operation: <logically shifted operand> -> <operand>

Description:

A logical shift is performed on the byte, halfword or word operand. <shiftcount> is interpreted as a signed byte. Positive <shiftcount> implies left shift, negative <shiftcount> implies right shift. A shiftcount equal to or greater than the size of the operand will produce an illegal operand value trap condition. A shiftcount of zero is legal and leaves the operand unchanged.

Trap conditions: Addressing traps, illegal operand value

Data status bits:

<shifted operand> = 0 -> Z
<shifted operand>.signbit -> S

Example:

Shift local word COUNT TWOFACTORS places

```
W SHL B.COUNT, TWOFACTORS
```

10.43. Arithmetical shift

Format: t SHA <operand/rw/t>,<shiftcount/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SHA	byte shift arithmetically	0FCABH	176253B
H SHA	halfword shift arithmetically	0FCACH	176254B
W SHA	word shift arithmetically	0FCADH	176255B

Operation: <arithmetically shifted operand> -> <operand>

Description:

An arithmetic shift is performed on the byte, halfword or word operand. <shiftcount> is interpreted as a signed byte. Positive <shiftcount> implies left shift, negative <shiftcount> implies right shift. A shiftcount equal to or greater than the size of the operand will produce an illegal operand value trap condition. A shiftcount of zero is legal and leaves the operand unchanged.

Trap conditions: Addressing traps, illegal operand value

Data status bits:

```
<shifted operand> = 0      -> Z
<shifted operand>.signbit -> S
```

Example:

Shift byte register R4 two places right

```
BY SHA R4, -2
```

10.44. Rotational shift

Format: t SHR <operand/rw/t>,<shiftcount/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SHR	byte shift rotationally	0FCAEH	176256B
H SHR	halfword shift rotationally	0FCAFH	176257B
W SHR	word shift rotationally	0FCB0H	176260B

Operation: <rotationally shifted operand> -> <operand>

Description:

A rotational shift is performed on the byte, halfword or word operand. <shiftcount> is interpreted as a signed byte. Positve <shiftcount> implies left shift, negative <shiftcount> implies right shift. A shiftcount equal to or greater than the size of the operand will produce an illegal operand value trap condition. A shiftcount of zero is legal and leaves the operand unchanged.

Trap conditions: Addressing traps, illegal operand value

Data status bits:

```
<shifted operand> = 0      -> Z
<shifted operand>.signbit -> S
```

Example:

Exchange nibbles (4 bit groups) of variable pointed at by R4

```
BY SHR R4.0, 4
```

10.45. Get bit

Format: tn GETBI <operand/r/t>,<bit no/r/BY>

Assembly notation	Name	Hex code	Octal code
BYn GETBI	byte get bit	0FCB4H+(n-1)	176264B+(n-1)
Hn GETBI	halfword get bit	0FCB8H+(n-1)	176270B+(n-1)
Wn GETBI	word get bit	0FDD0H+(n-1)	176720B+(n-1)

Operation: bit <bit no> of <operand> -> (bit 0 of Rn)

Description:

Bit zero of the specified register is loaded with bit <bit no> of a BY, H, or W <operand>. A <bit no> greater than or equal to the number of bits of the data type or a negative <bit no> will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits: (transferred bit) = 0 -> Z

Example:

Load R1 with the BITNO bit of word variable STATUS

```
W1  GETBI STATUS, BITNO
```

10.46. Put bit

Format: tn PUTBI <operand/w/t>,<bit no/r/BY>

Assembly notation	Name	Hex code	Octal code
BYn PUTBI	byte put bit	0FDD4H+(n-1)	176724B+(n-1)
Hn PUTBI	halfword put bit	0FDD8H+(n-1)	176730B+(n-1)
Wn PUTBI	word put bit	0FDDCH+(n-1)	176734B+(n-1)

Operation: (bit 0 of Rn) -> bit <bit no> of <operand>

Description:

Bit zero of the specified register is stored in bit <bit no> of a BY, H, or W <operand>. The upper bits of the <operand> is unaffected, even when the destination is a word register. A <bit no> greater than or equal to the number of bits of the data type or a negative <bit no> will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits: (transferred bit) = 0 -> Z

Example:

Store bit zero of R4 in bit 4 of local byte variable FLAGS

```
BY4 PUTBI B.FLAGS, 4
```

10.47. Clear bit

Format: t CLEBI <operand/w/t>,<bit no/r/BY>

Assembly notation	Name	Hex code	Octal code
BY CLEBI	byte clear bit	0FE7DH	176175B
H CLEBI	halfword clear bit	0FE7EH	176176B
W CLEBI	word clear bit	0FE7FH	176177B

Operation: 0 -> bit <bit no> of <operand>

Description:

The specified bit of a BY, H, or W <operand> is cleared. A <bit no> greater than or equal to the number of bits of the data type or a negative <bit no> will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits: 1 -> Z

Example:

Clear bit N of word register R1

```
    W CLEBI R1, N
```

10.48. Set bit

Format: t SETBI <operand/w/t>,<bit no/r/BY>

Assembly notation		Name	Hex code	Octal code
BY	SETBI	byte set bit	0FE80H	176200B
H	SETBI	halfword set bit	0FE81H	176201B
W	SETBI	word set bit	0FE82H	176202B

Operation: 1 -> bit <bit no> of <operand>

Description:

The specified bit of a BY, H, or W <operand> is set. A <bit no> greater than or equal to the number of bits of the data type or a negative <bit no> will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits: (transferred bit) = 0 -> Z

Example:

Set bit FAILURE in word argument EXCEPTIONS on the alternative domain

```
W SETBI ALT(IND(B.EXCEPTIONS)), FAILURE
```

10.49. Get bit field

Format: tn GETBF <operand/r/t>,<bit no/r/BY>,<field size/r/BY>

Assembly notation	Name	Hex code	Octal code
BYn GETBF	byte get bit field	0FDE0H+(n-1)	176740B+(n-1)
Hn GETBF	halfword get bit field	0FDE4H+(n-1)	176744B+(n-1)
Wn GETBF	word get bit field	0FDE8H+(n-1)	176750B+(n-1)

Operation: specified bit field -> Rn

Description:

Bit 0 to <field size> - 1 of the specified register is loaded with the specified bit field. In the <operand>, the bit field is composed of the <bit no> bit and the higher numbered bits to a field size of <field size> bits. (See the section on data types in memory for an explanation of bit numbers within data types.) The <operand> may have BY, H, or W as the data type. <bit no> and <field size> are interpreted as signed byte integers.

An illegal operand value trap condition is caused if <bit no> is negative, if <field size> is zero or negative, or if <bit no> or <bit no> + <field size> is greater than the number of bits in the data type.

The upper bits of the register are zero filled.

Trap conditions: Addressing traps, illegal operand value

Data status bits:

```
(bit field) = 0          -> Z
(bit field).leftmost bit -> S
```

Example:

Load R2 with a field consisting of bits 11 to 18 of the word variable 16 bytes away from the current R register

```
W2 GETBF R.16, 11, 8
```

10.50. Put bit field

Format: tn PUTBF <operand/w/t>,<bit no/r/BY>,<field size/r/BY>

Assembly notation	Name	Hex code	Octal code
BYn PUTBF	byte put bit field	0FDECH+(n-1)	176754B+(n-1)
Hn PUTBF	halfword put bit field	0FDF0H+(n-1)	176760B+(n-1)
Wn PUTBF	word put bit field	0FDF4H+(n-1)	176764B+(n-1)

Operation: Rn -> specified bit field

Description:

Bit 0 to <field size> - 1 of the specified register is stored in the specified bit field of the operand. In the <operand>, the bit field is composed of the <bit no> bit and the higher numbered bits to a field size of <field size> bits. (See the section on data types in memory for an explanation of bit numbers within data types.) The <operand> may have BY, H, or W as the data type. <bit no> and <field size> are interpreted as signed byte integers.

An illegal operand value trap condition is caused if <bit no> is negative, if <field size> is zero or negative, or if <bit no> or <bit no> + <field size> is greater than the number of bits in the data type.

Trap conditions: Addressing traps, illegal operand value

Data status bits:

```
(bit field) = 0           -> Z
(bit field).leftmost bit -> S
```

Example:

Put the 8 lower bits of R2 into the the record variable FLAGSET from bit ERRFLAGS and up

```
W2 PUTBF R.FLAGSET, ERRFLAGS, 8
```

10.51. A to the I'th power

Format: tn AXI <a/r/t>,<i/r/W>

Assembly notation	Name	Hex code	Octal code
Fn AXI	float A to the I'th power	0FCC0H+(n-1)	176300B+(n-1)
Dn AXI	double float A to the I'th power	0FCC4H+(n-1)	176304B+(n-1)

Operation: <A>**<I> -> Rn

Description:

<A> to the <I>'th power is calculated and loaded into the specified float or double float register. <A> can be float or double float. <I> is word integer. When <I> is negative and <A> is equal to zero, it causes an illegal operand value trap condition and the result is set to the largest floating point number. When <I> is zero, the result is one.

Trap conditions: Addressing traps, floating overflow, floating underflow, illegal operand value

Data status bits:

```
(result) = 0          -> Z
(result).signbit      -> S
(floating underflow)  -> FU
(floating overflow)   -> FO
```

Example:

Load 2.0 to the STATE power into F3

```
F3 AXI 2.0, STATE
```

10.52. I to the J'th power

Format: tn IXI <i/r/t>,<j/r/t>

Assembly notation	Name	Hex code	Octal code
BYn IXI	byte I to the J'th power	0FCC8H+(n-1)	176310B+(n-1)
Hn IXI	halfword I to the J'th power	0FCCCH+(n-1)	176314B+(n-1)
Wn IXI	word I to the J'th power	0FCD0H+(n-1)	176320B+(n-1)

Operation: <I>**<J> -> (datatype dependent part of register)

Description:

I to the <J>'th power is calculated and the result is loaded into the specified register. When the data type is BY or H, the result is loaded into the lower part of the specified register. If <J> is negative and <I> is different from 1 or -1, the result will be set to zero. If <J> is negative and <I> is 0, it will cause an illegal operand value trap condition and the result is set to zero.

Trap conditions: Addressing traps, illegal operand value, integer overflow

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
(overflow)        -> O
```

Example:

Load the byte register R1 with the cube of argument SIDE

```
BY1 IXI IND(B.SIDE), 3
```

10.53. Square root

Format: tn SQRT <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn SQRT	float square root	0FCD4H+(n-1)	176324B+(n-1)
Dn SQRT	double float square root	0FCD8H+(n-1)	176330B+(n-1)

Operation: SQRT(<argument>) -> Rn

Description:

The square root of the argument is calculated and the result is loaded into the specified float or double float register. A negative argument is illegal and will give a result of zero and an invalid operation trap.

Trap conditions: Addressing traps, invalid operation

Data status bits:

(result) = 0 -> Z
(argument) < 0 -> IVO

Example:

Load double float register D1 with the square root of AREA

```
D1 SQRT AREA
```

10.54. Polynomial

Format: tn POLY <x/r/t>,<n/s/BY>,
<cn/r/t>,....,<c1/r/t>,<c0/r/t>;

Assembly notation	Name	Hex code	Octal code
Fn POLY	floating polynomial	0FCE0H+(n-1)	176340B+(n-1)
Dn POLY	double float polynomial	0FCE4H+(n-1)	176344B+(n-1)

Operation:

$$<c0> + <c1>*<x> + <c2>*<x>^2 + \ldots\ldots\ldots + <cn>*<x>^n \rightarrow Rn$$

Description:

This instruction calculates a polynomial of the degree n. The result is loaded into the specified float or double float register. The instruction requires <n>+1 coefficients. <n> must always be a positive constant less than 256, otherwise an illegal operand specifier trap condition occurs.

Trap conditions: Addressing traps, floating overflow, floating underflow

Data status bits:

(result) = 0 -> Z
(result).signbit -> S
(floating underflow) -> FU
(floating overflow) -> FO

Example:

Calculate the expression A * X**2 + B * X + C and leave the result in F3. A, B and C are the coefficients

```
F3 POLY X, 2, A, B, C
```

10.55. Floating point remainder

Format: tn REM <x/r/t>,<y/r/t>,<q/w/t>

Assembly notation	Name	Hex code	Octal code
Fn REM	float divide with remainder	0FE58H+(n-1)	177130B+(n-1)
Dn REM	double float divide with remainder	0FE5CH+(n-1)	177134B+(n-1)

Operation:

The remainder of <x> / <y> in float format -> Rn
The integer part of <x> / <y> in float format -> <q>

Description:

<x> is divided by <y> and the integer part of the quotient in float format stored in <q>. The remainder of the quotient in float format is loaded into the specified register.

Trap conditions: Addressing traps, floating overflow, floating underflow, divide by zero

Data status bits:

remainder = 0 -> Z
remainder.signbit -> S
(floating underflow) -> FU
(floating overflow) -> FO
<y> = 0 -> DZ

Example:

Divide record variables EXPENSES with AMOUNT giving UNITCOST and a remainder in F2

```
F2 REM R.EXPENSES, R.AMOUNT, R.UNITCOST
```

10.56. Integer part

Format: tn INT <x/r/t>

Assembly notation	Name	Hex code	Octal code
Fn INT	float integer part	0FE60H+(n-1)	177140B+(n-1)
Dn INT	double float integer part	0FE64H+(n-1)	177144B+(n-1)

Operation: Truncated integer part of <x> in float format -> Rn

Description:

The truncated integer part of the <x> operand is calculated and loaded into the specified floating register in float format. No rounding is performed.

Trap conditions: Addressing traps

Data status bits:

```
result = 0      -> Z
result.signbit -> S
```

Example:

Load F4 with the integer part of EXACT

```
F4 INT EXACT
```

10.57. Integer part with rounding

Format: tn INTR <x/r/t>

Assembly notation	Name	Hex code	Octal code
Fn INTR	float integer part with rounding	0FE68H+(n-1)	177150B+(n-1)
Dn INTR	double float integer part with rounding	0FE6CH+(n-1)	177154B+(n-1)

Operation: rounded integer part of <x> in float format -> Rn

Description:

The rounded integer part of the <x> operand is calculated and loaded into the specified floating point register in float format. The result is rounded.

Trap conditions: Addressing traps

Data status bits:

result = 0 -> Z
result.signbit -> S

Example:

Load F4 with the rounded value 4 bytes away from the location pointed to by R3 on the alternative domain and force the displacement to occupy one word

```
F4 INTR ALT(R3.4:W)
```

10.58. Multiply and add

Format: tn MULAD <x/r/t>,<y/r/t>

Assembly notation	Name	Hex code	Octal code
BYn MULAD	byte multiply and add	0FCE8H+(n-1)	176350B+(n-1)
Hn MULAD	halfword multiply and add	0FCECH+(n-1)	176354B+(n-1)
Wn MULAD	word multiply and add	0A8H+(n-1)	250B+(n-1)
Fn MULAD	float multiply and add	0FCF0H+(n-1)	176360B+(n-1)
Dn MULAD	double float multiply and add	0FCF4H+(n-1)	176364B+(n-1)

Operation: Rn * <x> + <y> -> Rn

Description:

The specified register is multiplied by the <x> operand, the <y> operand is added to the product and the result loaded into the register.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
(result) = 0          -> Z
(result).signbit      -> S
(carry)               -> C (integer)
0                     -> C (float)
(overflow)            -> O (integer)
0                     -> O (float)
(floating underflow)  -> FU
(floating overflow)   -> FO
```

Example:

Multiply halfword register R2 with 60, forcing byte constant, and add MINUTES

```
H2 MULAD 60:B, MINUTES
```

10.59. Sum of products

Format: tn PSUM <x/r/t>,<y/r/t>

Assembly notation		Name	Hex code	Octal code
BYn	PSUM	byte add and multiply	0FCF8H+(n-1)	176370B+(n-1)
Hn	PSUM	halfword add and multiply	0FCFCH+(n-1)	176374B+(n-1)
Wn	PSUM	word add and multiply	0FD00H+(n-1)	176400B+(n-1)
Fn	PSUM	float add and multiply	0FD04H+(n-1)	176404B+(n-1)
Dn	PSUM	double float add and multiply	0FD08H+(n-1)	176410B+(n-1)

Operation: <x> * <y> + Rn -> Rn

Description:

The <x> operand is multiplied by the <y> operand and the product added to the specified register.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow

Data status bits:

```
(result) = 0          -> Z
(result).signbit      -> S
(carry)               -> C (integer)
0                     -> C (float)
(overflow)            -> O (integer)
0                     -> O (float)
(floating underflow)  -> FU
(floating overflow)   -> FO
```

Example:

Add local floats UNITCOST times UNITS to F4

```
F4 PSUM B.UNITCOST, B.UNITS
```

10.60. Load index

Format: tn LIND <index/r/t/>,<lower/r/t>,<upper/r/t>

Assembly notation	Name	Hex code	Octal code
BYn LIND	byte load index	0FD0CH+(n-1)	176414B+(n-1)
Hn LIND	halfword load index	0FD10H+(n-1)	176420B+(n-1)
Wn LIND	word load index	0ACH+(n-1)	254B+(n-1)

Operation:

```
<index> -> Rn
if <index> is less than <lower>
or <index> is greater than <upper> then
    1->K
    illegal index trap condition
 else
    0->K
 endif
```

Description:

An array index value is loaded into the specified register, checking the value against the <lower> and <upper> bounds. If the <index> operand is less than the <lower> operand or greater than the <upper> operand, the status flag bit (K) is set and an illegal index trap condition occurs. Otherwise the K flag is reset.

Trap conditions: Addressing traps, illegal index

Data status bits:

```
<index> = 0      -> Z
<index>.signbit -> S
```

Example:

Load R2 with the byte value IX, with limits -10 and 10

```
BY2 LIND IX, -10, 10
```

10.61. Calculate index

Format: tn CIND <index/r/t>,<lower/r/t>,<upper/r/t>

Assembly notation	Name	Hex code	Octal code
BYn CIND	byte calculate index	0FD14H+(n-1)	176424B+(n-1)
Hn CIND	halfword calculate index	0FD18H+(n-1)	176430B+(n-1)
Wn CIND	word calculate index	0B0H+(n-1)	260B+(n-1)

Operation: Rn * (<upper> - <lower> + 1) + <index> -> Rn

```
if <index> is less than <lower>
or <index> is greater than <upper> then
   1->K
   illegal index trap condition
else
   0->K
endif
```

Description:

The address of an element in a multiple dimensional array is calculated. The range of the dimension, <upper> - <lower> + 1, is multiplied by the contents of the specified register. <index> is added to the product and the result loaded into the specified register. If <index> is less than the contents of the <lower> operand or greater than the <upper> operand, the flag bit is set and an illegal index trap condition occurs.

Trap conditions: Addressing traps, integer overflow, illegal index

Data status bits:

```
(result) = 0          -> Z
(result).signbit = 0  -> S
(overflow)            -> O
```

Example:

Assuming ARRAY is declared with limits ARR(1..3,5..10,2..9), load R1 with the address of ARR(IX1,IX2,IX3), where the indexes are local word variables

```
W1 CIND IX1, 1, 3
W1 CIND IX2, 5, 10
W1 CIND IX3, 2, 9
```

11. CONTROL INSTRUCTIONS

11.1. Unconditional relative jump

Format: GO <<displacement>>

Assembly notation	Name	Hex code	Octal code
GO:B	jump byte	0C0H	300B
GO:H	jump halfword	0C1H	301B
GO:W	jump word	0C2H	302B

Operation: P + <<displacement>> -> P

Description:

Performs a jump relative to the current program counter value. GO uses a direct operand and has three formats, with byte, halfword, or word displacement part. The displacement is signed and is found in the 1, 2 or 4 bytes following the instruction code. A jump to another segment is illegal and will cause an instruction sequence error trap condition.

Trap conditions: Addressing traps, instruction seqence error

Data status bits: Unaffected

Example:

Jump to BACK (Assembler will calculate displacement)

```
BACK:   .
        .
        .
    GO BACK
```

11.2. Unconditional absolute jump

Format: JUMPG <address/r/W>

Assembly notation	Name	Hex code	Octal code
JUMPG	jump general	0B4H	264B

Operation: <address> -> P

Description:

Perform a jump to the absolute address given by the operand. JUMPG requires a general operand. Jump to alternative domain is illegal, ie. the prefix ALT is illegal in <address>. Jump to another segment will cause an instruction sequence error.

If a descriptor range trap occurs, the next instruction to be executed is the one following the JUMPG instruction ("fall through").

Trap conditions: Addressing traps, illegal operand specifier, instruction seqence error

Data status bits: Unaffected

Example:

Jump to the R1st address in a jump table described by CASETABLE

```
JUMPG DESC(CASETABLE)(R1)
```

11.3. Conditional jump

Formats:

IF <rel> GO <<displacement>>

IF <rel> GO <bit no/r/BY>, <<displacement>>

Operation:

```
if  <rel> then
    (P)+<<displacement>> -> (P)
else
    <start of next instruction> -> (P)
endif
```

Description:

A conditional jump will cause a transfer of control if and only if a specified condition is true. Otherwise the instruction following the IF <rel> GO will be the next to be executed.

The condition is specified in terms of the status bits set by instructions operating on data values. If the condition indicated by the instruction is true, the sign-extended <<displacement>> is added to the program counter.

Conditional jump on specified bits in the status register is possible by the second format of the instruction. In this case the <rel> operand may be ST or -ST, and the <bit no> operand specifies which bit in the status register to test. <bit no> has the range 0 to 29 inclusive. Other values for <bit no> will cause an illegal operand value trap condition.

Magnitude tests are only meaningful after compare and subtract instructions, as carry is reset in load instructions. IF >>= GO and IF << GO may be used as explicit tests on carry.

Trap conditions: Addressing traps, illegal operand value

Data status bits: Unaffected

In the following table all conditional jump instructions are listed with operation code, assembly notation, data status test for jumping and name. They all have conditional jump as the first part of the name. (alt. is an abbreviation for alternate)

CONTROL INSTRUCTIONS

Instruction Codes

Assembly notation	Condition	Name	Hex code	Octal code
IF = GO	Z=1	equal		
IF Z GO		(alt. assembly notation)		
IF = GO:B		byte displacement	0C4H	304B
IF = GO:H		halfword displacement	0C5H	305B
IF >< GO	Z=0	unequal		
IF -Z GO		(alt. assembly notation)		
IF >< GO:B		byte displacement	0C6H	306B
IF >< GO:H		halfword displacement	0C7H	307B
IF > GO	S=0 and Z=0	greater signed		
IF > GO:B			0C8H	310B
IF > GO:H			0C9H	311B
IF < GO	S=1	less signed		
IF S GO		(alt. assembly notation)		
IF < GO:B			0CAH	312B
IF < GO:H			0CBH	313B
IF >= GO	S=0	greater or equal signed		
IF -S GO		(alt. assembly notation)		
IF >= GO:B			0CCH	314B
IF >= GO:H			0CDH	315B
IF <= GO	S=1 or Z=1	less or equal signed		
IF <= GO:B			0CEH	316B
IF <= GO:H			0CFH	317B
IF K GO	K=1	flag		
IF K GO:B			0D0H	320B
IF K GO:H			0D1H	321B
IF -K GO	K=0	not flag		
IF -K GO:B			0D2H	322B
IF -K GO:H			0D3H	323B
IF >> GO	C=1 and Z=0	greater magnitude		
IF >> GO:B			0D4H	324B
IF >> GO:H			0D5H	325B
IF >>= GO	C=1	greater or equal magnitude		
IF C GO		(alt. assembly notation)		
IF >>= GO:B			0D6H	326B
IF >>= GO:H			0D7H	327B
IF << GO	C=0	less magnitude		
IF -C GO		(alt. assembly notation)		
IF << GO:B			0D8H	330B
IF << GO:H			0D9H	331B
IF <<= GO	C=0 or Z=1	less or equal magnitude		
IF <<= GO:B			0DAH	332B
IF <<= GO:H			0DBH	333B
IF ST GO		specified bit in status		
IF ST GO:B		register set	0FC7BH	176173B
IF ST GO:H			0FD64H	176544B
IF -ST GO		specified bit in status		
IF -ST GO:B		register not set	0FD65H	176545B
IF -ST GO:H			0FC84H	176204B

11.4. Loop with increment

Format: t LOOPI <index/rw/t>,<limit/r/t>,<<displacement>>

Assembly notation	Name	Hex code	Octal code
BY LOOPI:B	byte loop increment	0FCDEH	176336B
BY LOOPI:H	byte loop increment	0FD1EH	176436B
H LOOPI:B	halfword loop increment	0FCDFH	176337B
H LOOPI:H	halfword loop increment	0FD1FH	176437B
W LOOPI:B	word loop increment	0BFH	277B
W LOOPI:H	word loop increment	0E1H	341B
F LOOPI:B	float loop increment	0FD1CH	176434B
F LOOPI:H	float loop increment	0FD21H	176441B
D LOOPI:B	double float loop increment	0FD1DH	176435B
D LOOPI:H	double float loop increment	0FD22H	176442B

Operation:

```
if <index>+1 -> <index>  >  <limit> then
   (address of next instruction) -> P
else
   P+<<displacement>> -> P
endif
```

Description:

The index is incremented by one and compared with the limit. If it is less than or equal to the limit the signed displacement is added to the program counter, otherwise the control goes to the next instruction.

Normally the LOOPI instruction will be placed at the end of the loop, with a negative displacement. The displacement is the number of bytes from the first byte of the loop to the first byte of the LOOPI instruction.

<index> and <limit> have the same data type, which may be BY, H, W, F or D. <<displacement>> is a byte or halfword direct operand, depending on the instruction.

CONTROL INSTRUCTIONS

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
<modified index> = 0          -> Z
<modified index>.signbit      -> S
0                             -> O   (float)
(overflow)                    -> O   (integer)
0                             -> C   (float)
(carry from most significant bit) -> C   (integer)
```

Example:

Repeat the instructions from AGAIN until local byte COUNTER reaches 100

```
AGAIN: .
       .
       .
      BY LOOPI B.COUNTER, 100, AGAIN
```

11.5. Loop with decrement

Format: t LOOPD <index/rw/t>,<limit/r/t>,<<displacement>>

Assembly notation	Name	Hex code	Octal code
BY LOOPD:B	byte loop decrement	0FD23H	176443B
BY LOOPD:H	byte loop decrement	0FD28H	176450B
H LOOPD:B	halfword loop decrement	0FD24H	176444B
H LOOPD:H	halfword loop decrement	0FD29H	176451B
W LOOPD:B	word loop decrement	0FD25H	176445B
W LOOPD:H	word loop decrement	0FD2AH	176452B
F LOOPD:B	float loop decrement	0FD26H	176446B
F LOOPD:H	float loop decrement	0FD2BH	176453B
D LOOPD:B	double float loop decrement	0FD27H	176447B
D LOOPD:H	double float loop decrement	0FD2CH	176454B

Operation:

```
if <index>-1 -> <index> < <limit> then
   (address of next instruction) -> P
else
   P+ <<displacement>> -> P
endif
```

Description:

The index is decremented by one and compared with the limit. If it is greater than or equal to the limit the signed displacement is added to the program counter, otherwise control goes to the next instruction.

Normally the LOOPD instruction will be placed at the end of the loop, with a negative displacement. The displacement is the number of bytes from the first byte of the loop to the first byte of the LOOPD instruction.

<index> and <limit> have the same data type, which may be BY, H, W, F or D. <<displacement>> is a byte or halfword direct operand, depending on the instruction.

CONTROL INSTRUCTIONS

Trap conditions: Addressing traps, integer overflow

Data status bits:

<modified index> = 0	-> Z	
<modified index>.signbit	-> S	
0	-> O	(float)
(overflow)	-> O	(integer)
0	-> C	(float)
(carry from most significant bit)	-> C	(integer)

Example:

Repeat from TOP until word register R3 is decremented to zero

```
TOP:  .
      .
      .
      W LOOPD R3, 0:W, TOP
```

11.6. Loop general

Format: LOOP <index/rw/t>,<step/r/t>,
<limit/r/t>,<<displacement>>

Assembly notation	Name	Hex code	Octal code
BY LOOP:B	byte loop general step	0FD2DH	176455B
BY LOOP:H	byte loop general step	0FD32H	176462B
H LOOP:B	halfword loop general step	0FD2EH	176456B
H LOOP:H	halfword loop general step	0FD33H	176463B
W LOOP:B	word loop general step	0FD2FH	176457B
W LOOP:H	word loop general step	0FD34H	176464B
F LOOP:B	float loop general step	0FD30H	176460B
F LOOP:H	float loop general step	0FD35H	176465B
D LOOP:B	double float loop general step	0FD31H	176461B
D LOOP:H	double float loop general step	0FD36H	176466B

Operation:

```
if  <step> positive  then
    if <index>+<step> -> <index>  >  <limit> then
       (address of next instruction) -> P
    else
       P+ <<displacement>> -> P
    endif
endif
if  <step> negative then
    if <index>+<step> -> <index>  <  <limit> then
       (address of next instructon) -> P
    else
       P+ <<diplacement>> -> P
    endif
endif
if  <step> = zero then
    illegal operand value trap condition
endif
```

Description:

<step> is added to <index>. If the sign of <index> - <limit> i s equal to the sign of <step> the control goes to the next instruction. Otherwise the signed displacement is added to the program counter.

Normally the LOOP instruction will be placed at the end of the loop, with a negative displacement. The displacement is the number of bytes from the first byte of the loop to the first byte of the LOOPI instruction.

CONTROL INSTRUCTIONS

<index>, <step> and <limit> have the same data type, which may be BY, H, W, F or D. <<displacement>> is a byte or halfword direct operand, depending on the instruction.

A step size of zero will cause an illegal operand value trap.

Trap conditions: Addressing traps, integer overflow, floating overflow, floating underflow, illegal operand value

Data status bits:

<modified index> = 0	-> Z	
<modified index>.signbit	-> S	
(carry from most significant bit)	-> C	(integer)
0	-> C	(float)
(overflow)	-> O	(integer)
0	-> O	(float)
(floating underflow)	-> FU	
(floating overflow)	-> FO	

Example:

Execute the statements from LABELL with float record variable SIZE being incremented by 3.5 for each iteration up to a maximum of 35

```
LABELL: .
        .
        .
      F LOOP SIZE, 3.5, 35, LABELL
```

11.7. Call subroutine general

Format: CALLG <subr. addr/r/W>,<no of arg/s/BY>,
<arg1/aa/W>,...,<argn/aa/W>;

Assembly notation	Name	Hex code	Octal code
CALLG	call subroutine general	0B5H	265B

Operation:

Calculate the effective addresses of the arguments and prepare for the entry point at <subr. addr.>.
Jump to the subroutine entry point found at that address.

Description:

Call the subroutine at the address specified by the <subr. addr.> argument. This is a general operand and it must refer to an entry point instruction. Otherwise an instruction sequence error trap condition occurs.

The effective address of the arguments in the instruction is calculated and temporarily stored for use by the entry point instruction.

The <no of arg> operand must be a constant byte integer less than 256. Other data types which are not constants will cause an illegal operand specifier trap condition.

The arguments are always interpreted as word integer. The data type dependent addressing modes (post indexed or descriptor address code format) should be used with care, as the result will be wrong for arguments of other data types than word. <argn> operands of type register or constant will cause an illegal operand specifier trap condition, as neither registers nor constants have an address in data memory. The arguments may not be prefixed by the operand specifier prefix ALT; this will cause an illegal operand specifier trap condition.

A subroutine on the current segment is called by its address. A subroutine on another segment is called by its segment number in the upper 5 bits of the address and the routine number on the segment in the lower 27 bits. A detailed discussion is found in the memory management section.

CONTROL INSTRUCTIONS

Trap conditions: Addressing traps, call trap, illegal operand specifier, instruction seqence error

Data status bits: Unaffected

Example:

Call routine PRINT with arguments UNIT, FORMAT and the local variable VALUE

```
CALLG PRINT, 3, UNIT, FORMAT, B.VALUE
```

11.8. Call subroutine absolute

Format: CALL <<subr. addr.>>,<no of arg/s/BY>,
<arg1/aa/W>,...<argn/aa/W>

Assembly notation	Name	Hex code	Octal code
CALL	call subroutine absolute	0C3H	303B

Operation:

Calculate the effective addresses of the arguments and prepare for the entry point at <<subr. addr.>>.
Jump to the subroutine entry point found at that address.

Description:

Call the subroutine at the address specified by the <<subr. addr.>> argument. The subroutine address is a direct operand; the four bytes following the instruction code are taken as the subroutine address. The address must refer to an entry point instruction. Otherwise an instruction sequence error trap occurs.

The effective address of the arguments in the instruction is calculated and temporarily stored for use by the entry point instruction.

The <no of arg> operand must be a constant byte integer, ie. less than 256. Other data types which are not constants will cause an illegal operand specifier trap condition.

The arguments are always interpreted as word integer. The data type dependent addressing modes (post indexed or descriptor address code format) should be used with care, as the result will be wrong for arguments of other data types than word. <argn> operands of type register or constant will cause an illegal operand specifier trap condition, as neither registers nor constants have an address in data memory. The arguments may not be prefixed by the operand specifier prefix ALT; this will cause an illegal oprand specifier trap condition.

A subroutine on the current segment is called by its address. A subroutine on another segment is called by its segment number in the upper 5 bits of the address and the routine number on the segment in the lower 27 bits. A detailled discussion is found in the memory management section.

CONTROL INSTRUCTIONS

Trap conditions: Addressing traps, call trap, illegal operand specifier, instruction seqence error

Data status bits: Unaffected

Example:

Call SUBR with the value of local word variable READONLY. Value transfer should be used with word size data items only

```
CALL SUBR, 1, IND(B.READONLY)
```

11.9. Initialize stack

Format: INIT <<bottom of stack/r/W>>,
<stack demand of main program/r/W>,
<total system stack demand/r/W>

Assembly notation	Name	Hex code	Octal code
INIT	initialize stack	0DCH	334B

Operation:

```
<<bottom of stack>>                -> new.B

<<bottom of stack>> +
<total system stack demand>        -> TOS  (top of stack register)

<<bottom of stack>> +
<stack demand of main program>     -> newB.SP (next B)
0                                  -> newB.PREVB
0                                  -> newB.RETA -> L
```

Description:

The stack is initialized according to the instruction operands: The direct operand <<bottom of stack>> is a 4 byte absolute address, which is loaded into the B register. The B.SP location, the stack pointer, is loaded with the sum of <<bottom of stack>> and <stack demand of main program>. <<bottom of stack>> and <total system stack demand> is added and the result is loaded into the top of stack register, TOS. PREVB and RETA are cleared.

Trap conditions: Addressing traps

Data status bits: Unaffected

Example:

Initialize a new stack at FRAME, requiring 010000H stack locations for the system, 01000H for the main program

```
INIT FRAME, 010000H, 01000H
```

11.10. Subroutine entry points

Formats:

ENTM <<bottom of stack/r/W>>,<stack demand of main program/r/W>,
<total system stack demand/r/W>

ENTD

ENTS <stack demand/r/W>

ENTSN <stack demand/r/W>,<max no. of arg./r/W>

ENTF <<address of local data area/r/W>>

ENTFN <<address of local data area/r/W>>,<max no. of arg./r/W>

ENTT <trap handler main program stack demand/r/W>,
<total trap handler stack demand/r/W>

ENTB <log size/r/BY>

Operation:

Perform local data area initialization depending on the type of entry point.

Description:

The entry point instruction specifies the kind of local data area initialization performed on execution of a subroutine call instruction. This initialization includes transfer of the argument addresses to the new local data area at subroutine entry points, and saving of the current register block in the new local data area at the trap handler entry point.

Execution of an entry point instruction (except ENTT) not as a result of a subroutine call will cause an instruction sequence error trap condition. ENTT may only be executed as a result of a trap, and may not be used as an entry point by a CALL of CALLG.

ENTM - enter module

Assembly notation	Hex code	Octal code
ENTM <<bottom of stack/r/W>>, <stack demand of main program/r/W>, <total system stack demand/r/W>	ODFH	337B

Description:

When the ENTM entry point is used, a new stack is initialized. There will be a stack overflow trap if <stack demand of main program> is greater than or equal to <total stack demand of program system>.

Trap conditions: Addressing traps, instruction seqence error, stack overflow

Initializations performed:

```
TOS  (top of stack register)    -> IND(oldB.SP)
<<bottom of stack>>             -> new.B

<<bottom of stack>> +
<total system stack demand>     -> TOS

oldB                            -> newB.PREVB
(return address)                -> newB.RETA -> L

<<bottom of stack>> +
<stack demand of main program>  -> newB.SP

(number of arguments)           -> newB.N
(addresses of arguments)        -> newB.arg
```

CONTROL INSTRUCTIONS

ENTD - enter subroutine directly

Assembly notation	Hex code	Octal code
ENTD	09CH	234B

Description:

With ENTD as entry point, no initialization of local data area or parameter address transfer is performed. If the subroutine calls other subroutines, the L register must be saved and restored explicitly.

Trap conditions: Addressing traps, instruction seqence error

Initializations performed:

(return address) -> L

ENTS - enter stack subroutine

Assembly notation	Hex code	Octal code
ENTS <stack demand/r/W>	0B8H	270B

Description:

<stack demand> is the number of bytes needed for the local data field of the subroutine, including the predefined locations PREVB, RETA, SP, AUX and N (a total of 20 bytes). There will be a stack overflow trap condition if newB + <stack demand> is greater than or equal to TOS.

Trap conditions: Addressing traps, stack overflow, instruction seqence error

ENTSN - enter maximum number of arguments stack subroutine

Assembly notation	Hex code	Octal code
ENTSN <stack demand/r/W>,<max no. of arg./r/W>	0BAH	272B

Description:

ENTSN is similar to ENTS, but in addition the <no. of arg.> operand of the call subroutine instruction is compared against the <max no. of arguments> operand. If the number of arguments supplied with the call instruction is greater than the maximum number of arguments in the ENTSN instruction this will cause an illegal operand value trap condition and the program may be trapped. The maximum number of parameters allowed will be transferred to the stack, the remaining ignored.

Trap conditions: Address trap fetch, instruction seqence error, illegal operand value, stack overflow

Initializations performed:

```
oldB.SP                    -> newB
oldB                       -> newB.PREVB
(return address)           -> newB.RETA -> L
newB + <stackdemand>       -> newB.SP
(number of arguments)      -> newB.N
(addresses of arguments)   -> newB.arg
```

CONTROL INSTRUCTIONS

ENTF - enter subroutine

Assembly notation	Hex code	Octal code
ENTF <<address of local data area/r/W>>	0DDH	335B

Description:

Enter subroutine with fixed data area. Variables will keep their values between calls.

Trap conditions: Addressing traps, instruction seqence error

ENTFN - enter maximum number of arguments subroutine

Assembly notation	Hex code	Octal code
ENTFN <<address of local data area/r/W>>, <max no. of arg./r/W>	0DEH	336B

Description:

ENTFN is similar to ENTF, but in addition the <no. of arg.> operand of the call subroutine instruction is compared against the <max no. of arguments> operand. If the number of arguments supplied with the call instruction is greater than the maximum number of arguments in the ENTFN instruction this will cause an illegal operand value trap condition and the program may be trapped. The maximum number of parameters allowed will then be transferred to the stack, the remaining ignored.

Trap conditions: Addressing traps, illegal operand value, instruction seqence error

Initializations performed:

```
<<address of local data area>> -> newB
oldB                           -> newB.PREVB
(return address)               -> newB.RETA  -> L
oldB.SP                        -> newB.SP
(number of arguments)          -> newB.N
(addresses of arguments)       -> newB.ARG
```

ENTT - enter trap handler

Assembly notation		Hex code	Octal code
ENTT	<trap handler main program stack demand/r/W>, <total trap handler stack demand/r/W>	0BCH	274B

Description:

ENTT is trap handler entry point. A trap handler is invoked when a trap condition arises and the trap enable bit is set for the trap in question. When a trap handler routine is invoked, the start address is taken from a trap handler entry point vector. The THA register holds the address of this vector. The area following the trap handler vector is used as local data area for the invoked trap handler routine. It has a special layout illustrated in the chapter on traps.

The register block is stacked in the following sequence:

arg2 : P	arg18 : ST1	arg34 : CTE2
3 : L	19 : ST2	35 : MTE1
4 : B	20 : PS	36 : MTE2
5 : R	21 : TOS	37 : TEMM1
6 : I1	22 : LL	38 : TEMM2
7 : I2	23 : HL	39 : mic
8 : I3	24 : THA	40 : mic
9 : I4	25 : CED	
10 : A1	26 : CAD	
11 : A2	27 : CES	
12 : A3	28 : CAS	
13 : A4	29 : mic	
14 : E1	30 : mic	
15 : E2	31 : OTE1	
16 : E3	32 : OTE2	
17 : E4	33 : CTE1	

'mic' indicates registers accessible to microprogram only.

Following the register block is 10 words of program memory, starting at the first word containing the first byte of the instruction causing the trap. 'Trapping P' (the address of the first byte of the instruction causing the trap) points to a byte in the first word; the byte number is the two lower bits of 'trapping P'.

Trap conditions: Addressing traps, instruction seqence error

CONTROL INSTRUCTIONS

Initializations performed:

```
THA (trap handler register) + 256  -> newB
0                                  -> newB.PREVB
0                                  -> newB.RETA -> L

newB + <trap handler main
  program stack demand/r/W>        -> newB.SP

(address of the instruction
  that caused the trap)            -> newB.arg1

(register block)                   -> newB.arg2..newB.arg40
(10 words op program memory)       -> newB.arg41..newB.arg50

newB + <total trap handler
  stack demand/r/W>                -> TOS (top of stack)
```

ENTB - enter subroutine with buddy allocation

Assembly notation	Hex code	Octal code
ENTB <log size/r/BY>	0BDH	275B

Description:

A local data area of size 2**<log size> number of words is allocated from the heap and the subroutine is entered. There will be a stack overflow trap if there are no elements of the specified size (or larger) available from the heap. (See the chapter on buddy allocation for a detailed discussion.)

Trap conditions: Addressing traps, stack overflow, instruction seqence error

Initializations performed:

(address of an element from the heap)	-> newB
oldB	-> newB.PREVB
(return address)	-> newB.RETA -> L
oldB.SP	-> newB.SP
log size	-> newB.LOG
(number of arguments)	-> newB.N
(addresses of arguments)	-> newB.ARG

11.11. Subroutine return

Assembly notation	Name	Hex code	Octal code
RET	clear flag return from subroutine	080H	200B
RETK	set flag return from subroutine	081H	201B
RETD	return from direct subroutine	082H	202B
RETT	trap handler return	083H	203B
IF K RET	if flag set subroutine return	09DH	235B
RETB	buddy subroutine return	0FE1CH	177034B
RETBK	set flag buddy subroutine return	0FE1DH	177035B

Operation:

RET: 0 -> STATUS.K oldB.PREVB -> newB oldB.RETA -> P -> L

RETK: 1 -> STATUS.K oldB.PREVB -> newB oldB.RETA -> P -> L

RETD: L -> P

RETT: The register block is loaded from B.arg2..B.arg40
OTE is loaded from the domain information table
The status register is loaded partly from B.arg18..Barg19
and partly from domain information table (see chapter 3)

IF K RET: If STATUS.K = 1 then
oldB.PREVB -> newB oldB.RETA -> P -> L
endif

RETB: Local data area released to heap
0 -> STATUS.K oldB.PREVB -> newB oldB.RETA -> P -> L

RETBK: Local data area released to heap
1 -> STATUS.K oldB.PREVB -> newB oldB.RETA -> P -> L

Description:

RET, RETK

Return from subroutine with local data area. The new base register and return address are taken from the current local data area. RETK will set the flag bit of the status register; RET will clear it.

IF K RET

If the flag bit K is set when the IF K RET instruction is executed, a subroutine return is performed with the flag bit remaining set. Otherwise the control goes to the next instruction.

RETD

Load the new program counter from the link register.

RETT

Return from the trap handler. When RETT is executed, the register block is loaded from the first part of trap handler data area. The non-ignorable and fatal status bits are loaded from the domain information table. The OTE register is loaded from the domain information table. PREVB and RETA are not used or tested. CED of the trapped domain is compared to actual CED. If they are unequal, CED is changed back to trapped domain.

RETB, RETBK

Return from subroutines with heap elements as local data area. The local data field is released to the heap described by the variables pointed at by the TOS register. (See section about heap management for further explanation.)

Trap conditions: Stack underflow, address zero trap

Data status bits: Unaffected

The programmer must ensure that the appropriate return instruction is executed. Subroutines entered through an ENTS, ENTSN, ENTF or ENTFN instruction should be left through a RET, RETK or IF K RET instruction. ENTD routines should be left through RETD, ENTT routines through RETT, ENTB routines through RETB or RETBK.

If oldB.PREVB or oldB.RETA (L register if RETD) is zero, the return instruction (except RETT) will compare CAD to CED. If they are equal or CAD is zero a stack underflow trap condition occurs. If CAD is not equal to CED the current domain is changed back to CAD and the B, P and CAD registers are loaded from the domain information table.

RETT will compare the domain number of the trapped domain (saved on the trap handler stack) to the current executing domain. If they are equal, RETT returns within the same domain. Otherwise RETT changes the domain to the domain number saved on the stack.

12. STRING INSTRUCTIONS

12.1. Introduction

The string handling instructions make special use of the I1 and I2 registers as pointers to the source and destination string respectively. I2 is used only for those instructions which have a destination operand.

The operand in the instruction is the address of a string descriptor giving the start address of the string and its length. A DESC prefix is not allowed in the operand specifier; the descriptor addressing format is implicit in string instructions.

The register contains the character number within the string, starting at zero. It is not initialized before the instruction is executed and may be set by the user to point at any character. Characters outside the range indexed by the string instruction are unaffected.

Some instructions will refer to a translation table. This is 256 contiguous bytes and a translation is a reference in this table with the byte to be translated as an index. In the instruction descriptions Tr(S(I1)) means that the specified element is translated via a translation table.

The data status bits Z, O, S and the K flag may be affected by the string operations. The data status bits not mentioned in the string instruction description are all zero after the execution of the instruction. Carry is always cleared.

Execution of an instruction may terminate for various reasons and the termination condition sets the K, Z and/or S status bits. Destination full termination implies that 1 -> K. Execution termination for reasons other than destination full implies that 0 -> K. Status of the Z and S bit depends on the instruction.

After execution I1 and I2 point to the next element or the last element, depending on the termination conditions. Source string empty or destination full implies that I1 and I2 point to the next element. Next element is the first one not referred to by the instruction; if the end of the string is reached it is to the first character beyond the end of the string. String compare, until, while, translate until etc. have a third termination condition which implies that I1 and I2 point to the last element which was examined/moved.

When more than one termination condition is reached at the same time, the instruction terminates with the first of these mentioned in the termination condition list of the instruction.

STRING INSTRUCTIONS

Strings occupying the same locations in memory are said to be overlapping. If the source and destination operands overlap, the result will be as intended only if the old contents an element in the source string is moved out before it is overwritten with a new value. In cases where the length of the string operands can be determined prior to start of execution, the microcode will take care of overlap if neccessary by operating on the string elements in reverse order.

For instructions containing a 'while' or 'until' condition, the length is not determined before execution has been started, and it is not possible to predict the degree of overlapping. The programmer must ensure that strings do not overlap, otherwise the results are unpredictable.

Instruction descriptions use the following notation:

<=operand=> : Implicit descriptor operand, ie. the specified operand is a descriptor and the operand of the instruction is accessed via this descriptor.

:- : "is set to point at."

S(I1) : I1'th character in source string
D(I2) : I2'th character in destination or source-2 string

12.2. String move

Format: t SMOVE <=source/r/t/I1=>,<=dest/w/t/I2=>

Assembly notation	Name	Hex code	Octal code
BI SMOVE	bit string move	0FD66H	176546B
BY SMOVE	byte string move	0FD67H	176547B
H SMOVE	halfword string move	0FD68H	176550B
W SMOVE	word string move	0FD69H	176551B
F SMOVE	float string move	0FD6AH	176552B
D SMOVE	double float string move	0FD6BH	176553B

Operation:
```
while not end of string do
   S(I1) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

String elements are moved from the source string to the destination string until the source is empty or the destination is full.
Overlap is taken care of.

Terminating conditions:

source empty : K = 0 I1, I2 :- next element

destination full : K = 1 I1, I2 :- next element

Example:

Move the double float array whose descriptor is argument DATABLOCK to the area described by local descriptor COPY

```
W1 CLR; W2 CLR
D  SMOVE IND(B.DATABLOCK), B.COPY
```

12.3. String move while

Format: BY SMVWH <=source/r/BY/I1=>,<=dest/w/BY/I2=>, <mask/r/BY>, <test/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SMVWH	byte string move while	0FD72H	176562B

Operation:

```
while S(I1) AND <mask> = <test> do
  S(I1) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes are moved from the source string to the destination string. When the moved byte "anded" with <mask> is equal to <test>, the moving continues until the source string is empty or the destination string is full. Overlap is not taken care of.

Terminating conditions:

source empty:	K = 0	Z = 1	I1, I2 :- next element
destination full:	K = 1	Z = 1	I1, I2 :- next element
different byte found:	K = 0	Z = 0	I1, I2 :- last element

Example:

Copy characters from INPUT to BUFFER as long as the characters are in the range 100B to 177B, starting at current character positions in I1 and I2

```
BY SMVWH INPUT, BUFFER, 300B, 100B
```

12.4. String move until

Format: BY SMVUN <=source/r/BY/I1=>,<=dest/w/BY/I2=>,
<mask/r/BY>, <test/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SMVUN	byte string move until	0FD73H	176563B

Operation:
```
while S(I1) AND <mask> >< <test> do
   S(I1) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes are moved from source to destination until the source is empty, the destination is full or the next byte "anded" with <mask> to be moved is equal to <test>. Overlap is not taken care of.

Terminating conditions:

source empty:	K = 0	Z = 0	I1, I2 :- next element
destination full:	K = 1	Z = 0	I1, I2 :- next element
equal byte found:	K = 0	Z = 1	I1, I2 :- last element

Example:

Copy characters from argument ARG on the alternative domain to the global string LINE in the current domain

```
W1 CLR; W2 CLR
BY SMVUN ALT(IND(B.ARG)), LINE, 377B, 47B
```

12.5. String move translated

Format: BY SMVTR <=source/r/BY/I1=>,<=dest/w/BY/I2=>,
<trans table/aa/BY>

Assembly notation	Name	Hex code	Octal code
BY SMVTR	byte string move translated	0FD74H	176564B

Operation:
```
while not end of strings do
   tr(S(I1)) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes from the source string are translated via a translation table found at the address specified in the operand <trans table>. Translated bytes are moved from source to destination string until the source is empty or the destination is full. Overlap is taken care of.

Terminating conditions:

source empty: K = 0 I1, I2 :- next element

destination full: K = 1 I1, I2 :- next element

Example:

Convert the string CHARACTERS from EBCDIC to ASCII

```
W1 CLR; W2 CLR
BY SMVTR CHARACTERS, CHARACTERS, EBCDIC2ASCII
```

12.6. String move translated until

Format: BY SMVTU <=source/r/BY/I1=>,<=dest/w/BY/I2=>,
<trans table/aa/BY>

Assembly notation	Name	Hex code	Octal code
BY SMVTU	byte string move translated until	0FD75H	176565B

Operation:

```
while tr(S(I1))  ><  ASCII "escape"
and not end of string do
   if tr(S(I1))  ><  zero  then
      tr(S(I1)) -> D(I2),  I2+1 -> I2
   endif
   I1+1 -> I1
enddo
```

Description:

Bytes from the source string are translated via the translation table found at the address specified in the operand <trans table>. Translated bytes are moved from source to destination string if they are not zero. The move operation stops if the translated byte is equal to ASCII "escape" (01BH or 33B), the source is empty or the destination full. Overlap is not taken care of.

Terminating conditions:

source empty:	K = 0	Z = 0	I1, I2 :- next element
destination full:	K = 1	Z = 0	I1, I2 :- next element
"escape" found:	K = 0	Z = 1	I1, I2 :- last element

Example:

Remove ASCII Nulls and translate to uppercase the string described by record variable TEXT, copying it to the string described by TEXT2, starting at the current position

```
BY SMVTU R.TEXT, TEXT2, UPPERCASETABLE
```

12.7. String move n elements

Format: t SMOVN <=source/r/t/I1=>,<=dest/w/t/I2=>,<n/r/W>

Assembly notation	Name	Hex code	Octal code
BI SMOVN	string move n bits	0FD76H	176566B
BY SMOVN	string move n bytes	0FD77H	176567B
H SMOVN	string move n halfwords	0FD78H	176570B
W SMOVN	string move n words	0FD79H	176571B
F SMOVN	string move n floats	0FD7AH	176572B
D SMOVN	string move n double floats	0FD7BH	176573B

Operation: Move n elements from source to destination

Description:

N items are moved from source to destination string, unless the source is empty or the destinaton full. Overlap is taken care of.

Terminating conditions:

source empty:	K = 0	Z = 0	I1, I2 :- next element
destination full:	K = 1	Z = 0	I1, I2 :- next element
n items moved:	K = 0	Z = 1	I1, I2 :- next element

Example:

Copy next 64 bits from S1 to start of S2, both global descriptors

```
W2 CLR
BI SMOVN S1, S2, 64
```

12.8. String fill

Format: tn SFILL <=dest/w/t/I2=>

Assembly notation	Name	Hex code	Octal code
BIn SFILL	bit string fill	0FD7CH+(n-1)	176574B+(n-1)
BYn SFILL	byte string fill	0FD80H+(n-1)	176600B+(n-1)
Hn SFILL	halfword string fill	0FD84H+(n-1)	176604B+(n-1)
Wn SFILL	word string fill	0FD88H+(n-1)	176610B+(n-1)
Fn SFILL	float string fill	0FD8CH+(n-1)	176614B+(n-1)
Dn SFILL	double float string fill	0FD90H+(n-1)	176620B+(n-1)

Operation: Rn -> every element of <=dest=>

Description:

The contents of the specified register are put into every element of the destination string.

Terminating conditions:
K = 1 I2 :- next element

Data status bits: All cleared

Example:

Fill the remaining characters of STRING with ASCII spaces (40B)

```
BY3 := 40B
BY3 SFILL STRING
```

12.9. String fill n elements

Format: tn SFILLN <=dest/w/t/I2=>,<n/r/W>

Assembly notation	Name	Hex code	Octal code
BIn SFILLN	string fill n bits	0FD94H+(n-1)	176624B+(n-1)
BYn SFILLN	string fill n bytes	0FD98H+(n-1)	176630B+(n-1)
Hn SFILLN	string fill n halfwords	0FD9CH+(n-1)	176634B+(n-1)
Wn SFILLN	string fill n words	0FDA0H+(n-1)	176640B+(n-1)
Fn SFILLN	string fill n floats	0FDA4H+(n-1)	176644B+(n-1)
Dn SFILLN	string fill n double float	0FDA8H+(n-1)	176650B+(n-1)

Operation: Rn -> n first elements of <=dest=>

Description:

If the number of elements in the destination string is greater than n, the contents of the specified register are stored in the n first elements of the destination string. Otherwise all elements of the destination string are filled with the contents of the register.

Terminating conditions:

destination full: K = 1 Z = 0 I2 :- next element

n elements filled: K = 0 Z = 1 I2 :- next element

Example:

Zero fill the lower 100 words of the word string described by local FI

```
W1 CLR; W2 CLR
W1 SFILLN B.FI, 100
```

12.10. String compare

Format: BY SCOMP <=source-1/r/BY/I1=>,<=source-2/r/BY/I2=>

Assembly notation	Name	Hex code	Octal code
BY SCOMP	byte string compare	0FDACH	176654B

Operation:
```
while S(I1)  =  D(I2) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes from the source-1 string are compared to the corresponding bytes in the source-2 string until unequal bytes are found, or until the end of source-1 or source-2 string is reached. When unequal bytes are found, the status bits Z and S and the K flag will indicate the termination condition.
The byte elements are considered to be unsigned values.

Terminating conditions:

exact match:	K = 0	Z = 1	S = 0	I1, I2 :- next element
source-1 string longer than source-2:	K = 0	Z = 0	S = 0	I1, I2 :- next element
source-1 string shorter than source-2:	K = 0	Z = 0	S = 1	I1, I2 :- next element
greater byte in source-1 found:	K = 1	Z = 0	S = 0	I1, I2 :- last element
less byte in source-1 found:	K = 1	Z = 0	S = 1	I1, I2 :- last element

Example:

Scan INPUTLINE and local COMMAND from the current positions until different characters or end of string is detected

```
BY SCOMP INPUTLINE, B.COMMAND
```

12.11. String compare translated

Format: BY SCOTR <=source-1/r/BY/I1=>,<=source-2/w/BY/I2=>,
<trans table/aa/BY>

Assembly notation	Name	Hex code	Octal code
BY SCOTR	byte string compare translated	0FDADH	176655B

Operation:
```
while tr(S(I1)) = tr(D(I2)) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Translated bytes from the source-1 string are compared to the corresponding translated bytes in the source-2 string. This comparison continues until unequal bytes are found, or until the end of the source-1 or source-2 string is reached.
The byte elements are considered to be unsigned values.

Terminating conditions:

exact match:	K = 0	Z = 1	S = 0	I1, I2 :-	next element
source-1 string longer than source-2:	K = 0	Z = 0	S = 0	I1, I2 :-	next element
source-1 string shorter than source-2:	K = 0	Z = 0	S = 1	I1, I2 :-	next element
greater byte in source-1 found:	K = 1	Z = 0	S = 0	I1, I2 :-	last element
less byte in source-1 found:	K = 1	Z = 0	S = 1	I1, I2 :-	last element

Example:

Scan INPUTLINE and local COMMAND from the current position until end of string or different characters, converting to uppercase

```
BY SCOTR  INPUTLINE, B.COMMAND, UPPERCASE
```

12.12. String compare with pad

Format: BY SCOPA <=source-1/r/BY/I1=>,
<=source-2/r/BY/I2=>,<pad/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SCOPA	string compare with pad	0FDBEH	176676B

Operation:
```
while S(I1) = D(I2) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes from the source-1 string are compared to the corresponding bytes in the source-2 string until unequal bytes are found, or until the end of strings are reached. If the lengths of the source-1 and source-2 strings are not equal, the shortest string is concatenated with a string of pad bytes. The length of the pad string is equal to the difference in length of the source-1 and the source-2 string. When unequal bytes are found, the status bits Z and S and the K flag will indicate the termination condition.
The byte elements are considered to be unsigned values.

Terminating conditions:

exact match:	K = 0 Z = 1 S = 0	I1, I2 :- next element
greater byte in source-1 found:	K = 1 Z = 0 S = 0	I1, I2 :- last element
less byte in source-1 found:	K = 1 Z = 0 S = 1	I1, I2 :- last element

Example:

Compare argument ITEM with global TABLE, padding with ASCII spaces

```
BY SCOPA IND(B.ITEM), TABLE, 20H
```

12.13. String compare translated with pad

Format: BY SCOPT <=source-1/r/BY/I1=>,<=source-2/w/BY/I2=>,
<trans table/aa/BY>,<pad/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SCOPT	string compare translated with pad	0FDBFH	176677B

Operation:

```
while tr(S(I1)) = tr(D(I2)) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Translated bytes from the source-1 string are compared to the corresponding translated bytes in the source-2 string. This logical comparison continues until unequal bytes are found, or until the end of the strings are reached. If the lengths of the source-1 and source-2 strings are not equal, the shortest string is concatenated with a string of pad bytes. The length of the pad string is equal to the difference in length of the source-1 and the source-2 string. The pad byte is also translated.
The byte elements are considered to be unsigned values.

Terminating conditions:

exact match:	K = 0 Z = 1 S = 0 I1, I2 :- next element
greater byte in source-1 found:	K = 1 Z = 0 S = 0 I1, I2 :- last element
less byte in source-1 found:	K = 1 Z = 0 S = 1 I1, I2 :- last element

Example:

Compare ITEM on the alternate domain from the 10th character to LIST from the 0th character, translating to uppercase. Pad byte is zero

```
W1 := 10; W2 CLR
BY SCOPT ALT(ITEM), LIST, UPPERCASE, 0
```

12.14. Skip elements

Format: BY SSKIP <=source/r/BY/I1=>,<test/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SSKIP	skip elements	0FDAEH	176656B

Operation:

```
while S(I1)  =  <test> do
   I1 + 1 -> I1
enddo
if S(I1)  >  <test> then
   0 -> S
else
   1 -> S
endif
```

Description:

Bytes in the source string are examined one by one until an examined byte is different from the <test> operand or until the end of source string is reached.
The byte elements are considered to be unsigned values.

Terminating conditions:

source empty:	K = 0	Z = 1		I1 :- next element
byte >> <test> found:	K = 0	Z = 0	S = 0	I1 :- last element
byte << <test> found:	K = 0	Z = 0	S = 1	I1 :- last element

Example:

Skip ASCII spaces in the string described by record addressed LINE from the current character on

```
BY SSKIP R.LINE, 32
```

12.15. String locate elements

Format: t SLOCA <=source/r/t/I1=>,<test/r/BI,BY>

Assembly notation	Name	Hex code	Octal code
BI SLOCA	string locate bit	0FDAFH	176657B
BY SLOCA	string locate byte	0FDB0H	176660B

Operation:

```
while S(I1) >< <test> do
   I1 + 1 -> I1
enddo
```

Description:

The source string is examined element by element until an examined element is equal to the <test> operand or until the end of source string is reached.

Terminating conditions:

source empty:	K = 0	Z = 0	I1 :- next element
byte = <test> found:	K = 0	Z = 1	I1 :- last element

Example:

Find the next reset bit in the bit string on the alternative domain described by the record variable RESERVED

```
BI SLOCA ALT(R.RESERVED), 0
```

12.16. String scan

Format: BY SSCAN <=source/r/BY/I1=>,<mask/r/BY>,
<trans table/aa/BY>

Assembly notation	Name	Hex code	Octal code
BY SSCAN	string scan	0FDB1H	176661B

Operation:
```
while tr(S(I1)) AND <mask>  =  zero  do
   I1 + 1 -> I1
enddo
```

Description:

The source string is scanned until the current translated byte "anded" with <mask> is different from zero, or until the end of source string is reached.

Terminating conditions:

source empty:	K = 0	Z = 1	I1 :- next element
byte >< zero found:	K = 0	Z = 0	I1 :- last element

Example:

Skip through argument FUNCTION until a byte with one of the bits set in the mask ACTIVE, translated through the table FNTAB in the alternative domain, is encountered

```
BY SSCAN IND(B.FUNCTION), ACTIVE, ALT(FNTAB)
```

12.17. String span

Format: BY SSPAN <=source/r/BY/I1=>,<mask/r/BY>,
<trans table/aa/BY>

Assembly notation	Name	Hex code	Octal code
BY SSPAN	string span	0FDB2H	176662B

Operation:
```
while tr(S(I1)) AND <mask>  >< zero do
   I1 + 1 -> I1
enddo
```

Description:

The source string is examined until the examined byte translated and "anded" with <mask> is equal to zero, or until the end of source string is reached.

Terminating conditions:

source empty:	K = 0	Z = 0	I1 :- next element
byte = zero found :	K = 0	Z = 1	I1 :- last unequal element

Example:

Skip the remaining of a string fragment DIRECTIVE terminated by a character translating to zero in the local table CODETABLE

```
BY SSPAN DIRECTIVE, 0FFH, B.CODETABLE
```

12.18. String match

Format: BY SMATCH <=substring/r/BY/I1=>,<=source/r/BY/I2=>

Assembly notation	Name	Hex code	Octal code
BY SMATCH	string match	0FDB3H	176663B

Operation:

```
while <=substring=> >< <=source=>(I2..I2 + substring length - 1)
and end of <source> not reached do
   I2 + 1 -> I2
enddo
if  <=substring=> = <=source=>(I2..I2 + substring length - 1) then
   1 -> Z
else
   0 -> Z
endif
```

Description:

The source string is examined until either a substring equal to <=substring=> is found or the end of source string is reached. The I1 register is left unmodified.

Terminating conditions:

substring found: K = 0 Z = 1 I2 :- first matching byte

source empty: K = 0 Z = 0 I2 :- next element

Example:

Set I1 to point to the next occurence of COMMA in PARAMETERS

```
BY SMATCH COMMA, PARAMETERS
```

12.19. Set parity in string

Format: BY SSPAR <=source/rw/BY/I1=>,<mode/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SSPAR	set parity in string	0FDB4H	176664B

Operation: Set parity in all bytes in <=source=>

Description:

The parity bit (bit 7) in every byte is set according to the following values of the <mode> operand:

0 clear parity
1 set parity
2 even parity
3 odd parity

A <mode> different from 0-3 will cause an illegal operand value trap condition.

Terminating conditions: K = 1

Example:

Set even parity in local string OUTPUT

```
BY SSPAR B.OUTPUT, 3
```

12.20. Check parity in string

Format: BY SCHPAR <=source/r/BY/I1=>,<mode/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SCHPAR	check parity in string	0FDB5H	176665B

Operation: Parity is checked in all bytes in <=source=>

Description:

The parity bit (bit 7) in every byte is checked according to the following values of the <mode> operand:

0	clear parity
1	set parity
2	even parity
3	odd parity

A <mode> different from 0-3 will cause an illegal operand trap condition.

Terminating conditions:

source empty: Z = 0 I1 :- next element

parity error found: Z = 1 I1 :- last element

Example:

Check that parity is set according to argument MODE in all characters in record variable BUFFER

```
W1 CLR
BY SCHPAR R.BUFFER, IND(B.MODE);
```

13. MISCELLANEOUS INSTRUCTIONS

13.1. Block move and Fill

Format: t BMOVE <source/r/t>,<dest/w/t>,<n/r/W>

Assembly notation	Name	Hex code	Octal code
BY BMOVE	byte block move	0FD20H	176440B
H BMOVE	halfword block move	0FE78H	177170B
W BMOVE	word block move	0FE79H	177171B
F BMOVE	float block move	0FE7AH	177172B
D BMOVE	double float block move	0FE7BH	177173B

Operation:

```
i = 0
while i < n do
   S(i) -> D(i); i + 1 -> i
enddo
```

Description:

<n> elements are moved from the source to the destination. The operands are pointers to the start of the blocks. Overlap is taken care of. Constant and register are illegal as destination operands. When a register or a constant is specified as a source operand, the destination string is filled with <n> elements equal to the value of the source.

Trap conditions: Addressing traps

Data status bits: All cleared

Terminating conditions: n bytes moved

Example:

Fill local data area of routine (excluding header) with the largest negative word value (bit pattern equivalent to float minus zero) with the intention to facilitate detection of uninitialized variables

```
W1 := 080000000H
W BMOVE W1, B.20, AREASIZE
```

13.2. Data type conversion

Format: t1 t2CONV <source/r/t1>,<dest/w/t2>

Assembly notation	Name	Hex code	Octal code
BI BYCONV	bit to byte convert	0FD44H	176504B
BI HCONV	bit to halfword convert	0FD45H	176505B
BI WCONV	bit to word convert	0FD46H	176506B
BI FCONV	bit to float convert	0FD47H	176507B
BI DCONV	bit to double float convert	0FD48H	176510B
BY BICONV	byte to bit convert	0FD49H	176511B
BY HCONV	byte to halfword convert	0FD4AH	176512B
BY WCONV	byte to word convert	0FD4BH	176513B
BY FCONV	byte to float convert	0FD4CH	176514B
BY DCONV	byte to double float convert	0FD4DH	176515B
H BICONV	halfword to bit convert	0FD4EH	176516B
H BYCONV	halfword to byte convert	0FD4FH	176517B
H WCONV	halfword to word convert	0FD50H	176520B
H FCONV	halfword to float convert	0FD51H	176521B
H DCONV	halfword to double float convert	0FD52H	176522B
W BICONV	word to bit convert	0FD53H	176523B
W BYCONV	word to byte convert	0FD54H	176524B
W HCONV	word to halfword convert	0FD55H	176525B
W FCONV	word to float convert	0FD56H	176526B
W DCONV	word to double float convert	0FD57H	176527B
F BICONV	float to bit convert	0FD58H	176530B
F BYCONV	float to byte convert	0FD59H	176531B
F HCONV	float to halfword convert	0FD5AH	176532B
F WCONV	float to word convert	0FD5BH	176533B
F DCONV	float to double float convert	0FD5CH	176534B
D BICONV	double float to bit convert	0FD5DH	176535B
D BYCONV	double float to byte convert	0FD5EH	176536B
D HCONV	double float to halfword convert	0FD5FH	176537B
D WCONV	double float to word convert	0FD60H	176540B
D FCONV	double float to float convert	0FD61H	176541B

MISCELLANEOUS INSTRUCTIONS

Operation: <source> type converted from t1 to t2 -> <dest>

Description:

The <source> operand of type t1 is converted to data type t2 with the result stored in the <dest> operand. The result is not rounded.

For integer types, conversion of shorter to a longer data type is by sign extension. Conversion of longer to shorter data types is by truncation of the most significant bits and may cause integer overflow. Conversion from float to integer may also cause integer overflow.

Conversion from bit implies that the result is zero if the bit is cleared and is one if the bit is set. Conversion to bit implies that the bit is set if the source is different from zero and cleared otherwise.

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
```

Example:

Load the byte variable SHORTINT to W2 with sign extension to word

```
BY WCONV SHORTINT, W2
```

13.3. Data type conversion with rounding

Format: t1 t2CONR <source/r/t1>,<dest/w/t2>

Assembly notation	Name	Hex code	Octal code
F BYCONR	float to byte convert with rounding	0FE70H	177160B
D BYCONR	double float to byte convert with rounding	0FE71H	177161B
F HCONR	float to halfword convert with rounding	0FE72H	177162B
D HCONR	double float to halfword convert with rounding	0FE73H	177163B
F WCONR	float to word convert with rounding	0FE74H	177164B
D WCONR	double float to word convert with rounding	0FE75H	177165B
W FCONR	word to float convert with rounding	0FE83H	177203B
D FCONR	double float to float convert with rounding	0FE84H	177204B

Operation: <source> converted from t1 to t2 with rounding -> <dest>

Description:

The <source> operand of type t1 is converted to data type t2 with the result stored in the <dest> operand. The result is rounded.

Trap conditions: Addressing traps, integer overflow

Data status bits:

```
(result) = 0      -> Z
(result).signbit -> S
```

Example:

The R2nd value in the double precision array described by RESULTS is rounded to R2nd element of halfword argument ROUNDEDRESULT

```
D HCONR DESC(RESULTS)(R2), IND(B.ROUNDEDRESULT)(R2)
```

13.4. Load address

Format: tn LADDR <operand/aa/t>

Assembly notation	Name	Hex code	Octal code
BIn LADDR	bit load address	0FE20H+(n-1)	177040B+(n-1)
BYn LADDR	byte load address	0FE24H+(n-1)	177044B+(n-1)
Hn LADDR	halfword load address	0FE28H+(n-1)	177050B+(n-1)
Wn LADDR	word load address	0FD3CH+(n-1)	176474B+(n-1)
Fn LADDR	float load address	0FD3CH+(n-1)	176474B+(n-1)
Dn LADDR	double float load address	0FE2CH+(n-1)	177054B+(n-1)

Operation: (address of <operand>) -> Rn

Description:

The address of the operand is loaded into the specified register. Registers and constants have no address in memory and are illegal as operands. If the segment number of the calculated adress is zero the current executing segment number is inserted into the result.

Trap conditions: Addressing traps

Data status bits: result = 0 -> Z

Example:

Load the address of the R3rd element of the halfword array argument TABLE into R1

```
H1 LADDR B.TABLE(R3)
```

13.5. Load address into record register

Format: t RLADDR <operand/aa/t>

Assembly notation		Name	Hex code	Octal code
BI	RLADDR	bit load address to R	0FC55H	176125B
BY	RLADDR	byte load address to R	0FC5AH	176132B
H	RLADDR	halfword load address to R	0FCB1H	176261B
W	RLADDR	word load address to R	0BEH	276B
F	RLADDR	float load address to R	0BEH	276B
D	RLADDR	double float load address to R	0FCB2H	176262B

Operation: (address of <operand>) -> R

Description:

The address of the operand is loaded into the record register. Registers and constants have no address in memory and are illegal as operands. If the segment number of the calculated address is zero the current executing segment number is inserted into the result.

Trap conditions: Addressing traps

Data status bits: result = 0 -> Z

Example:

Load R with the base address of the first stack frame below the current

```
W RLADDR IND(B.0)
```

13.6. Load address into base register

Format: t BLADDR <operand/aa/t>

Assembly notation		Name	Hex code	Octal code
BI	BLADDR	bit load address to B	0FCB3H	176263B
BY	BLADDR	byte load address to B	0FCBCH	176274B
H	BLADDR	halfword load address to B	0FD37H	176467B
W	BLADDR	word load address to B	0FD63H	176543B
F	BLADDR	float load address to B	0FD63H	176543B
D	BLADDR	double float load address to B	0FD38H	176470B

Operation: (address of <operand>) -> B

Description:

The address of the operand is loaded into the local base register. Registers and constants have no address in memory and are illegal as operands. If the segment number of the calculated address is zero the current executing segment number is inserted into the result.

Trap conditions: Addressing traps

Data status bits: result = 0 -> Z

Example:

Load B with the address of argument NEWB

```
W BLADDR B.NEWB
```

13.7. Load address of multilevel link

Format: Wn CHAIN <address/aa/W>,<offset/r/W>,<no of levels>

Assembly notation	Name	Hex code	Octal code
Wn CHAIN	load address of multilevel link to register	0FD6CH+(n-1)	176554+(n-1)

Operation:

```
<address> -> Wn
for i in (1..<no of levels>) do
   ((Wn) + <offset>) -> Wn
enddo
```

Description:

Follow a link <no of levels> steps and load the specified register with the base address of the next data element. This instruction is used by language processors for making references to variables declared in an outer procedure. <offset> will usually be the B relative address of the static link (the base address of the local variables of an enclosing procedure), <address> the current B register value, and <no of levels> the difference between the current static level and the level where the variable was declared.

Trap conditions: Addressing traps

Data status bits:

Example:

Load R1 with stack base address of a procedure five static levels up, the static link is found in local variable STATLINK

```
W1 CHAIN B.STATLINK, STATL, 5
```

13.8. No operation

Format: NOOP

Assembly notation	Name	Hex code	Octal code
NOOP	no operation	003H	003B

Operation: None

Description:

The no operation instruction may be used for deleting code from a program or to leave open space for later modifications.

Trap conditions: None

Data status bits: Unaffected

Example:

```
NOOP
```

13.9. Set flag

Format: SETK

Assembly notation	Name	Hex code	Octal code
SETK	set flag	0FE02H	177002B

Operation: 1 -> (flag bit of status register)

Description:

Set the flag bit of the status register

Trap conditions: None

Data status bits: Unaffected

Example:

```
SETK
```

13.10. Clear flag

Format: CLRK

Assembly notation	Name	Hex code	Octal code
CLRK	clear flag	0FE03H	177003B

Operation: 0 -> (flag bit of status register)

Description:

Clear the flag bit of the status register

Trap conditions: None

Data status bits: Unaffected

Example:

```
CLRK
```

13.11. Get buddy element

Format: Wn GETB <log size/r/BY>

Assembly notation	Name	Hex code	Octal code
Wn GETB	get buddy element from heap	0FE4CH+(n-1)	177114B+(n-1)

Operation: Allocates element of size 2**<log size> words
Address of element -> Wn

Description:

Allocates an element of size 2**<log size> words from the heap.

If an element of the given size is available it is removed from the freelist and its address is returned in the specified register. Otherwise the list is examined for larger elements. If none is available this will cause a stack overflow trap condition. If a larger element is found, it is removed from its freelist and chopped into halves until an element of the desired size can be allocated. The other half of the chopped element(s) will be appended to the appropriate freelist.

The administration of the heap is described in chapter 3.3. When executing the GETB instruction, the TOS register must point to the variables describing the heap.

Trap conditions: Addressing traps, stack overflow

Data status bits: Unaffected

Example:

Allocate a 64 word data block from the heap, leaving its address in R3

```
W3 GETB 6
```

13.12. Free buddy element

Format: FREEB <log size/r/BY>,<element/aa/W>

Assembly notation	Name	Hex code	Octal code
FREEB	free buddy	0FDB6H	176666B

Operation: Release <element> of size 2**<log size> words to heap

Description:

The specified <element> is appended to the appropriate freelist of the heap. Elements are not combined; this may be done by a trap handler for the stack overflow condition.

The administration of the heap is described in chapter 3.3. When executing the FREEB instruction, the TOS register must point to the variables describing the heap.

Trap conditions: Addressing traps

Data status bits: Unaffected

Example:

Release string LINE of length 128 bytes to heap (LINE is a descriptor)

```
    FREEB 5, IND(LINE)
```

14. SPECIAL INSTRUCTIONS

14.1. Disable process switch

Format: SOLO

Assembly notation	Name	Hex code	Octal code
SOLO	disable process switch	0FE00H	177000B

Operation: disables process switch for maximum 256 micro-cycles

Description:

Ensure that instructions up to the next TUTTI instruction is executed as an indivisible sequence of operations. SOLO is used for syncronizing purposes and implementation of protection mechanisms.

If the disable process switch is disabled for more than 256 micro-cycles, a disable process switch timeout occurs. Most simple instructions execute in one micro-cycle per operand specifier.

If a non-ignorable trap condition occurs when the process switch is disabled a disable process switch error trap condition occurs.

Trap conditions: Disable process switch timeout,disable process switch error

Data status bits: Unaffected

Example:

```
SOLO
```

14.2. Enable process switch

Format: TUTTI

Assembly notation	Name	Hex code	Octal code
TUTTI	enable process switch	0FE01H	177001B

Operation: enables process switch

Description:

The opposite of SOLO; allows normal interleaving of process execution in the system.

Trap conditions: None

Data status bits: Unaffected

Example:

```
      TUTTI
```

14.3. Set bit in trap enable register

Format: SETE <bit no/r/BY>

Assembly notation	Name	Hex code	Octal code
SETE	set bit in own trap enable register	0FD39H	176471B

Operation: Set bit <bit no> in own trap enable register

Description:

The specified bit in the own trap enable (OTE) register is set. The <bit no> operand is compared to a modify mask (TEMM) found in the domain description table. If a bit in this mask is set, the corresponding bit in the local trap enable register is modifiable. An attempt to modify a non-modifiable bit will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits: Unaffected

Example:

Enable the integer Overflow trap

```
SETE 9
```

14.4. Clear bit in trap enable register

Format: CLTE <bit no/r/BY>

Assembly notation	Name	Hex code	Octal code
CLTE	clear bit in own trap enable register	0FD3AH	176472B

Operation: Clear bit <bit no> in own trap enable register

Description:

The specified bit in the own trap enable register is cleared. An ignorable trap condition will be ignored and no trap handler invoked unless the corresponding MTE bit is set. A non-ignorable trap condition will be propagated to the mother domain.

The <bit no> operand is compared to a modify mask found in the domain description table. If a bit in this mask is set, the corresponding bit in the local trap enable register is modifiable. An attempt to modify a non-modifiable bit will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits: Unaffected

Example:

Disable Single Instruction Trap

```
    CLTE 17
```

14.5. Break point

Format: BP

Assembly notation	Name	Hex code	Octal code
BP	break point instruction	002H	002B

Operation: Cause a break point instruction trap condition

Description:

This instruction causes a break point instruction trap condition. If the break point trap is not enabled, it will cause an illegal instruction code trap condition.

The BP instruction is intended for program debugging and the trap handler will normally invoke a debug routine.

Trap conditions: breakpoint instruction trap, illegal instruction code

Data status bits: Unaffected

Example:

```
BP
```

14.6. Test and set

Format: BY TSET <operand/rwl/BY>

Assembly notation	Name	Hex code	Octal code
BY TSET	test and set	0FD40H	176500B

Operation:
lock
read operand and set status bits
set operand to all ones
unlock

Description:

The TSET instruction will use a feature to be supplied in a future multiport memory system that will replace "Big multiport system", ND-143 to 146. In this future multiport system a single cycle memory swap may be executed by the process. This swap access is not interruptible by other processes or by channels connected to the memory system, therefore it may be used to implement processor synchronizing. It may be noted that no locking of any shared hardware resource is done by the TSET instruction.

Trap conditions: Addressing traps

Data status bits:

operand was zero before store -> Z
operand was negative before store -> S

Example:

Set byte variable RESERVE to all ones

```
BY TSET RESERVE
```

14.7. Load special register

Format: (special register) := <operand/r/W>

Assembly notation	Name	Hex code	Octal code
L:=	load link register	0FD3BH	176473B
HL:=	load upper limit register	0FDB7H	176667B
LL:=	load lower limit register	0FDB8H	176670B
ST1:=	load 1st status register	0FDB9H	176671B
OTE1:=	load 1st own trap enable register	0FDBBH	176673B
OTE2:=	load 2nd own trap enable register	0FDBCH	176674B
TOS:=	load top of stack register	0FDBDH	176675B
THA:=	load trap handler register	0FDCAH	176712B

Operation: <operand> -> (special register)

Description:

Special registers can be loaded with this group of instructions.

Some of the bits in the status register (listed in the Status bits survey section) are not modifiable. When loading the own trap enable register, the operand is compared to a modify mask (TEMM) found in the domain description table. If a bit in this mask is set, the corresponding bit in the trap enable register is modifiable. An attempt to modify a non-modifiable bit in the own trap enable register will cause an illegal operand value trap condition.

Trap conditions: Addressing traps, illegal operand value

Data status bits:

```
<operand> = 0      -> Z
<operand>.signbit -> S
```

The instruction ST1:= will load the data status bits from the operand.

Example:

Restore the TOS register from the current top of stack after a call to a routine entered through ENTM

```
TOS:= B.SP
```

14.8. Store special registers

Format: (special register) =: <operand/w/W>

Assembly notation	Name	Hex code	Octal code
L=:	store link register	0FDC0H	176700B
HL=:	store upper limit register	0FDC1H	176701B
LL=:	store lower limit register	0FDC2H	176702B
ST1=:	store 1st status register	0FDC3H	176703B
OTE1=:	store 1st own trap enable register	0FDC5H	176705B
OTE2=:	store 2nd own trap enable register	0FDC6H	176706B
MTE1=:	store 1st mother trap enable register	0FD70H	176560B
MTE2=:	store 2nd mother trap enable register	0FD71H	176561B
CTE1=:	store 1st child trap enable register	0FE50H	177120B
CTE2=:	store 2nd child trap enable register	0FE51H	177121B
TEMM1=:	store 1st trap enable modification mask	0FE52H	177122B
TEMM2=:	store 2nd trap enable modification mask	0FE53H	177123B
CED=:	store current executing domain	0FE54H	177124B
CAD=:	store current alternative domain	0FE55H	177125B
CES=:	store current executing segment	0FE56H	177126B
CAS=:	store current segment alternative domain	0FE57H	177127B
PS=:	store process segment	0FE7CH	177174B
TOS=:	store top of stack register	0FDC9H	176711B
THA=:	store trap handler register	0FDCBH	176713B
P=:	store program counter	0FD62H	176542B

Operation: (special register) -> <operand>

Description:

Store the content of a special register into a specified operand.

When storing the program counter (P=:), the content of the operand will be the address of the first instruction following the P=: instruction.

Trap conditions: Addressing traps, illegal operand specifier

Data status bits:

<operand> = 0 -> Z
<operand>.signbit -> S

The instruction ST1=: does not affect the data status bits.

14.9. Integer float register communication

Format:

```
An=:  <operand/w/W>
En=:  <operand/w/W>
An:=  <operand/r/W>
En:=  <operand/r/W>
```

Assembly notation	Name	Hex code	Octal code
An:=	load most significant part of double float register	0FE30H+(n-1)	177060B+(n-1)
En:=	load least significant part of double float register	0FE34H+(n-1)	177064B+(n-1)
An=:	store most significant part of double float register	0FE38H+(n-1)	177070B+(n-1)
En=:	store least significant part of double float register	0FE3CH+(n-1)	177074B+(n-1)

Operation:

```
An:=   store most significant part of double float register
En:=   store least significant part of double float register
An=:   load most significant part of double float register
En=:   load least significant part of double float register
```

Description:

Load/store the most significant or least significant 32 bits of the double float registers. Note that a float register is equivalent to the most significant part of a double float register.

When a register is specified as an operand, the general integer registers are used. Thus, these instructions can transfer data between integer and float registers without performing any type conversion.

Trap conditions: Addressing traps

Data status bits:

```
(source register) = 0      -> Z
(source register).signbit -> S
```

SPECIAL INSTRUCTIONS

Example:

Store least significant part of D3 in local variable LEAST

```
E3 =: B.LEAST
```

15. COMMUNICATION BETWEEN NORD-500 AND NORD-100

15.1. Hardware interconnection

The interconnection between NORD-100 and NORD-500 consists of

* 5 control lines from NORD-100 to NORD-500
* 3 control lines from NORD-500 to NORD-100
* a 5 bit tag bus, two-way
* a 16 bit data bus, two-way

The control lines and the tag bus set up the data paths for the data transmitted through the data bus. NORD-100 accesses the control lines and the tag bus through IOX instructions, NORD-500 through microcode routines. By writing to the tag registers the data bus may be set up to transfer the contents of the following registers:

* Control register (16 bits)
  (for NORD-100 to give NORD-500 a command)
* Status register (16 bits)
  (for NORD-500 to give NORD-100 status)
* Address register (24 bits)
  (a pointer to NORD-100 memory where chains of commands or data will be found or where NORD-500 can store extended status information)

The NORD-500 microprogram can read and write by means of DMA in the memory of NORD-100. NORD-500 looks like another DMA device to NORD-100, and both processors may run in parallel. The NORD-100 starts the NORD-500 by writing an initiating command into the control word. While NORD-500 is running the communications is reserved for NORD-500; the communication interface is in the locked mode. For NORD-100 to diagnose NORD-500 operation the communication interface may be in the test mode. NORD-100 sets locked and test modes by writing to the control word, bit 2 and 3. The setting of locked and test modes determines the set of commands available to NORD-100. Under normal operation, test mode is reset, and locked mode is reset when NORD-100 is writing orders to NORD-500 or NORD-500 has completed all jobs submitted.

COMMUNICATION BETWEEN NORD-500 AND NORD-100

These commands are:

IOX instr	Locked Not test	Locked Test	Not locked Not test	Not locked Test
0000			Read addr	Read addr
0001			Load addr	Load addr
0010	Read status	Read status	Read status	Read status
0011				Load status
0100		Read control		Read control
0101			Load control	Load control
0110	Master clear		Master clear	Read data
0111	Terminate		Terminate	Load data
1000				
1001			Read tag	
1010			Load tag	
1011				
1100			Write data	
1101				
1110	Release lock		Release lock	
1111			Return tag	

The Read addr (read address register) and Load addr (load address register) instructions must be executed twice to transfer the full 24 bit address. The 16 least significant bits are transferred first, and then the 8 most significant. As the NORD-100 memory is addressed in units of 16 bit words, the 24 bit NORD-100 address range covers the same amount of physical memory as a NORD-500 25 bit byte address range.

While the mailbox is locked, the terminate command acts as a request, and NORD-500 will not honor the request until it has finished the current instruction. The interface will then be unlocked by the NORD-500 microprogram.

The N500 master clear will stop the NORD-500 immediately.

When the interface is unlocked, the NORD-100 may write a new command into the control word. The NORD-500 will then execute the new command.

The control register bits have the following interpretation:

bit no.

0	Enable interrupt from NORD-500
1	Not used
2	Activate NORD-500, lock mailbox
3	Test mode
4	NORD-500 programmed clear
5	Not used
6	DMA error
7	Command chaining
8..14	NORD-500 operation
15	Not used

The status register can always be read by NORD-100, and the bits are defined as follows:

bit no

0	Interrupt enabled
1	
2	NORD-500 busy
3	NORD-500 finished
4	Error
5	Interface locked
6	DMA error
7	NORD-500 power failure
8..15	NORD-500 process indentifier

15.2. Data packet format

Under normal operation, NORD-100 hands orders to NORD-500 as a linked list of packets residing in NORD-100 memory. The address register is loaded with the head of the chain, and NORD-500 is activated by setting bit 2 in the control register. NORD-500 will then start executing the first uncompleted packet in the queue, and continue with the next in the list as it finishes.

The data packets have a standard header and a trailer depending on the order type. The first two 16 bit words contain the link to the next element in the chain. The third contains the status code, taking the following values:

0 - element free, ignored by NORD-500
1 - message from NORD-100 to NORD-500, set by NORD-100
2 - waiting, set by NORD-500 as soon as execution of this order is started
3 - answer, set by NORD-500 as soon as execution of this order is completed
4 - error answer, set by NORD-500 when an abnormal situation caused execution of this order to terminate

Currently status code 4 is used to indicate page fault only.

The next two words identify the sending and the receiving process, respectively. The sender is the NORD-100 process and the receiver the corresponding NORD-500 process. The next word, the last word in the header, contains the length of the command dependent data part.

The data part of the order contains in the first word a function code. The length of the data part is dependent on the function code, and the layout varies. Its contents can be data, addresses or both. For example, the start order is read by NORD-500 when the order is considered for execution. When completed the fourth word contains the stop reason, which under normal conditions will be either a monitor call or a trap. In case of a monitor call the fifth word contains the number of parameters, the sixth the monitor call number. Then follow 16 32 bit parameter addresses and 16 32 bit data values. In case of a trap, the second and third word contains the P register, the forth the trap number and the following the contents of the registers as determined by the kind of trap.

The format of the data part is determined by microcode, and is subject to extentions and modifications in the future.

When NORD-500 is activated it will start searching the queue at the packet pointed to by the address register. If the status code in the packet is 0, 2, 3 or 4 the order is skipped and the next one considered. Status code 1 is the only one causing NORD-500 to start "executing" a packet. When a program stops due to a monitor call or trap, NORD-500 will give an interrupt to NORD-100 and continue with the next packet in the queue, without waiting for NORD-100 to react. The NORD-500 finished bit in the status register is not set until the end of the queue is encountered.

If required, NORD-100 may halt NORD-500 with the terminate IOX instruction. The interface will then be unlocked when NORD-500 is finished with its current instruction. NORD-500 may be restarted after, for example, the queue has been modified. When restarted, the address register may point to another entry, causing another process to be the one selected for execution.

While NORD-500 is executing, NORD-100 may read data in the packets handled by NORD-500 (recognized by a status code of 3 or 4), but no queue entry should be modified while NORD-500 is running.

APPENDIX A Address codes

Hexadecimal:

Name	Size	Operation		Hex layout
LOCAL	:S	ea=(B)+d*4	080H+xx	
LOCAL	:B	ea=(B)+d	0C1H	dd
LOCAL	:H	ea=(B)+d	0C2H	dd dd
LOCAL	:W	ea=(B)+d	0C3H	dd dd dd dd
LOCAL P.I.	:B	ea=(B)+d+p*(Rn)	0D4H+y	dd
LOCAL P.I.	:H	ea=(B)+d+p*(Rn)	0D8H+y	dd dd
LOCAL P.I.	:W	ea=(B)+d+p*(Rn)	0DCH+y	dd dd dd dd
LOCAL INDIRECT	:B	ea=((B)+d)	0C5H	dd
LOCAL INDIRECT	:H	ea=((B)+d)	0C6H	dd dd
LOCAL INDIRECT	:W	ea=((B)+d)	0C7H	dd dd dd dd
LOCAL INDIRECT P.I.	:B	ea=((B)+d)+p*(Rn)	0E4H+y	dd
LOCAL INDIRECT P.I.	:H	ea=((B)+d)+p*(Rn)	0E8H+y	dd dd
LOCAL INDIRECT P.I.	:W	ea=((B)+d)+p*(Rn)	0ECH+y	dd dd dd dd
RECORD	:S	ea=(R)+d*4	080H+xx	
RECORD	:B	ea=(R)+d	0C9H	dd
RECORD	:H	ea=(R)+d	0CAH	dd dd
RECORD	:W	ea=(R)+d	0CBH	dd dd dd dd
PRE INDEXED	:B	ea=(Rn)+d	0F4H+y	dd
PRE INDEXED	:H	ea=(Rn)+d	0F8H+y	dd dd
PRE INDEXED	:W	ea=(Rn)+d	0FCH+y	dd dd dd dd
ABSOLUTE		ea=a	0C4H	aa aa aa aa
ABSOLUTE P.I.		ea=a+(Rn)*p	0E0H+y	aa aa aa aa
CONSTANT	:S	op=c	000H+cc	
CONSTANT	:B	op=c	0CDH	cc
CONSTANT	:H	op=c	0CEH	cc cc
CONSTANT	:W	op=c	0CFH	cc cc cc cc
CONSTANT	:F	op=c	0CFH	cc cc cc cc
CONSTANT	:D	op=c	0CCH	cc cc cc cc cc cc cc cc
REGISTER		op=(Rn)	0D0H+y	
DESCRIPTOR		ea=A+p*(Rn)	0F0H+y	<operand>
ALTERNATIVE			0C8H	<operand>
Not used			0C0H	

Octal:

Name	Size	Operation	Octal layout	
LOCAL	:S	ea=(B)+d*4	100B+dd	
LOCAL	:B	ea=(B)+d	301B	ddd
LOCAL	:H	ea=(B)+d	302B	ddd ddd
LOCAL	:W	ea=(B)+d	303B	ddd ddd ddd ddd
LOCAL P.I.	:B	ea=(B)+d+p*(Rn)	324B+y	ddd
LOCAL P.I.	:H	ea=(B)+d+p*(Rn)	330B+y	ddd ddd
LOCAL P.I.	:W	ea=(B)+d+p*(Rn)	334B+y	ddd ddd ddd ddd
LOCAL INDIRECT	:B	ea=((B)+d)	305B	ddd
LOCAL INDIRECT	:H	ea=((B)+d)	306B	ddd ddd
LOCAL INDIRECT	:W	ea=((B)+d)	307B	ddd ddd ddd ddd
LOCAL INDIRECT P.I.	:B	ea=((B)+d)+p*(Rn)	344B+y	ddd
LOCAL INDIRECT P.I.	:H	ea=((B)+d)+p*(Rn)	350B+y	ddd ddd
LOCAL INDIRECT P.I.	:W	ea=((B)+d)+p*(Rn)	354B+y	ddd ddd ddd ddd
RECORD	:S	ea=(R)+d*4	200B+dd	
RECORD	:B	ea=(R)+d	311B	ddd
RECORD	:H	ea=(R)+d	312B	ddd ddd
RECORD	:W	ea=(R)+d	313B	ddd ddd ddd ddd
PRE INDEXED	:B	ea=(Rn)+d	364B+y	ddd
PRE INDEXED	:H	ea=(Rn)+d	370B+y	ddd ddd
PRE INDEXED	:W	ea=(Rn)+d	374B+y	ddd ddd ddd ddd
ABSOLUTE		ea=a	304B	aaa aaa aaa aaa
ABSOLUTE P.I.		ea=a+(Rn)*p	340B+y	aaa aaa aaa aaa
CONSTANT	:S	op=c	000B+cc	
CONSTANT	:B	op=c	315B	ccc
CONSTANT	:H	op=c	316B	ccc ccc
CONSTANT	:W	op=c	317B	ccc ccc ccc ccc
CONSTANT	:F	op=c	317B	ccc ccc ccc ccc
CONSTANT	:D	op=c	314B	ccc ccc ccc ccc ccc ccc ccc ccc
REGISTER		op=(Rn)	320B+y	
DESCRIPTOR		ea=A+p*(Rn)	360B+y	<operand>
ALTERNATIVE			310B	<operand>
Not used			300B	

APPENDIX B Address code table

Hexadecimal:

	:S	:B	:H	:W	:F	:D	PREFIX
LOCAL	040H+dd	0C1H	0C2H	0C3H			
LOCAL P.I.		0D4H+	0D8H+	0DCH+			
LOCAL INDIRECT		0C5H	0C6H	0C7H			
LOCAL INDIRECT P.I.		0E4H+	0E8H+	0ECH			
RECORD	080H+dd	0C9H	0CAH	0CBH			
PRE INDEXED		0F4H+	0F8H+	0FCH+			
ABSOLUTE				0C4H			
ABSOLUTE P.I.				0E0H+			
CONSTANT	000H+cc	0CDH	0CEH	0CFH	0CFH	0CCH	
REGISTER	0D0H+						

Address code prefixes:

DESCRIPTOR	0F0H+
ALTERNATIVE	0C8H

APPENDIX B Address code table

Octal:

	:S	:B	:H	:W	:F	:D	PREFIX
LOCAL	1ddB	301B	302B	303B			
LOCAL P.I.		324B+	330B+	334B+			
LOCAL INDIRECT		305B	306B	307B			
LOCAL INDIRECT P.I.		344B+	350B+	354B+			
RECORD	2ddB	311B	312B	313B			
PRE INDEXED		364B+	370B+	374B+			
ABSOLUTE				304B			
ABSOLUTE P.I.				340B+			
CONSTANT	0ccB	315B	316B	317B	317B	314B	
REGISTER	320B+						

Address code prefixes:

DESCRIPTOR	360B+
ALTERNATIVE	310B

APPENDIX C Symbols and abbreviations

METALANGUAGE SYMBOLS:

[ ]	optional syntax element
n	more than one optional syntax element
( )	contents of
::=	defined as
:=:	exchange contents of
:-	is set to point to
**	to the power of
< >	general operand
<< >>	direct operand
<=operand=>	implicit descriptor operand
P.I.	post index
alt.	alternative
no.	number
ea	effective address
op	value of operand, op=(ea)
A	descriptor.address
a	absolute address
c	constant
d	displacement
x	0,1,2,3,4,5,6,7 (octal) 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F (hexadecimal)
y	0,1,2, or 3 - specifies the registers R1-R4
p	1/8 (bit), 1 (byte), 2 (halfword), 4 (word), 4 (float), and 8 (double float). Post index scaling factor.
t	a subset of data types
displ.	displacement
log size	the logarithm to the base two of the size of a data element, in number of words

I1 I2 I3 I4	integer accumulators or index registers

Access Codes:

r	read access
w	write access
rw	read and write access
rwl	read, write and locked swap access
aa	address access
s	special, explained explicitly in the instruction descriptions

APPENDIX C Symbols and abbreviations

ASSEMBLY NOTATION:

Registers:

Rn	n=1..4	register, type determined by context
An	n=1..4	upper half of double precision register
En	n=1..4	lower half of double precision register
BIn	n=1..4	integer type register used for bit data
BYn	n=1..4	integer type register used for byte data
Hn	n=1..4	integer type register used for halfword data
Wn	n=1..4	integer type register used for word data
Fn	n=1..4	float type register used for single precision float
Dn	n=1..4	float type register used for double precision float
P		program counter
L		link (return address) register
B		local variable base register
R		record base register
ST		status register
OTE		own trap enable register
MTE		mother trap enable register
CTE		child trap enable register
TEMM		trap enable modification mask
TOS		top of stack register
LL		low limit trap register
HL		high limit trap register
THA		trap handler address register

Data types:

BI	bit
BY	byte
H	halfword
W	word
F	float
D	double float
BCD	binary coded decimal

Data part length specifiers:

:S	short	6 bits
:B	byte	8 bits
:H	halfword	2 bytes
:W	word	4 bytes
:F	float	4 bytes
:D	double float	8 bytes

APPENDIX D Figures

Page	Figure	Name
2	1-1	The NORD-500 computer system
6	2-1	The register block
8	3-1	Local data area layout
10	3-2	Layout of heap variables
14	4-1	Logical addressing scheme
15	4-2	Logical address
16	4-3	Hierarchy of program domains
17	4-4	Memory management registers
19	4-5	Capability layout
20	4-6	Domain information table
21	4-7	Program segment layout
22	4-8	Indirect segment
25	4-9	Physical segment table
26	4-10	Physical segment table entry
26	4-11	Physical memory
27	4-12	Addressing a program capability
28	4-13	Translation speedup buffer
31	5-1	The cache system, 128 K byte cache
34	6-1	Treatment of non-fatal trap conditions
35	6-2	Trap handler start address and local data field
51	7-1	Floating point rounding
53	7-2	Data formats in main memory
54	7-3	Arithmetic registers
55	7-4	Data in registers
56	8-1	Instruction format
57	8-2	Operand specifier format
58	8-3	Operand specifier structures
58	8-4	Operand specifier layout
59	8-5	Data part length specifiers
60	8-6	NORD-500 address modes
65	8-7	Local addressing
67	8-8	Local, post indexed addressing
69	8-9	Local indirect addressing
71	8-10	Local indirect, post indexed addressing
73	8-11	Record addressing
75	8-12	Pre indexed addressing
77	8-13	Absolute addressing
79	8-14	Absolute, post indexed addressing
81	8-15	Examples of constants
82	8-16	Treatment of constants as operands
86	8-17	Addressing with a descriptor
89	9-1	Instruction format
90	9-2	Instruction code formats

APPENDIX E Instruction table

ARITHMETICAL, LOGICAL, and DATA TRANSFER INSTRUCTIONS

BIn :=	load bit
BYn :=	load byte
Hn :=	load halfword
Wn :=	load word
Fn :=	load float
Dn :=	load double float
B :=	load local base
R :=	load record base
BIn =:	store bit
BYn =:	store byte
Hn =:	store halfword
Wn =:	store word
Fn =:	store float
Dn =:	store double float
B =:	local base store
R =:	record base store
BI MOVE	move bit
BY MOVE	move byte
H MOVE	move halfword
W MOVE	move word
F MOVE	move float
D MOVE	move double float
BI SWAP	bit swap
BY SWAP	byte swap
H SWAP	halfword swap
W SWAP	word swap
F SWAP	float swap
D SWAP	double float swap
BIn COMP	register bit compare
BYn COMP	register byte compare
Hn COMP	register halfword compare
Wn COMP	register word compare
Fn COMP	register float compare
Dn COMP	register float compare
BI COMP2	bit compare
BY COMP2	byte compare
H COMP2	halfword compare
W COMP2	word compare
F COMP2	float compare
D COMP2	double float compare

APPENDIX E Instruction table

BI	TEST	bit test against zero
BY	TEST	byte test against zero
H	TEST	halfword test against zero
W	TEST	word test against zero
F	TEST	float test against zero
D	TEST	double float test against zero
BYn	NEG	byte register negate
Hn	NEG	halfword register negate
Wn	NEG	word register negate
Fn	NEG	float register negate
Dn	NEG	double float register negate
BIn	INV	bit invert register
BYn	INV	byte invert register
Hn	INV	halfword invert register
Wn	INV	word invert register
Wn	INVC	word invert register with carry
BYn	ABS	byte absolute value
Hn	ABS	halfword absolute value
Wn	ABS	word absolute value
Fn	ABS	float absolute value
Dn	ABS	double float absolute value
BYn	+	byte add
Hn	+	halfword add
Wn	+	word add
Fn	+	floating add
Dn	+	double float add
BYn	-	byte subtract
Hn	-	halfword subtract
Wn	-	word subtract
Fn	-	float subtract
Dn	-	double float subtract
BYn	*	byte multiply
Hn	*	halfword multiply
Wn	*	word multiply
Fn	*	floating multiply
Dn	*	double float multiply
BYn	/	byte divide
Hn	/	halfword divide
Wn	/	word divide
Fn	/	float divide
Dn	/	double float divide
BY	ADD2	byte add two arguments
H	ADD2	halfword add two arguments
W	ADD2	word add two arguments
F	ADD2	float add two arguments
D	ADD2	double float add two arguments

BY	SUB2	byte subtract two arguments
H	SUB2	halfword subtract two arguments
W	SUB2	word subtract two arguments
F	SUB2	float subtract two arguments
D	SUB2	double float subtract two arguments
BY	MUL2	byte multiply two arguments
H	MUL2	halfword multiply two arguments
W	MUL2	word multiply two arguments
F	MUL2	float multiply two arguments
D	MUL2	double float multiply two arguments
BY	DIV2	byte divide two arguments
H	DIV2	halfword divide two arguments
W	DIV2	word divide two arguments
F	DIV2	float divide two arguments
D	DIV2	double float divide two arguments
BY	ADD3	byte add three arguments
H	ADD3	halfword add three arguments
W	ADD3	word add three arguments
F	ADD3	float add three arguments
D	ADD3	double float add three arguments
BY	SUB3	byte subtract three arguments
H	SUB3	halfword subtract three arguments
W	SUB3	word subtract three arguments
F	SUB3	float subtract three arguments
D	SUB3	double float subtract three arguments
BY	MUL3	byte multiply three arguments
H	MUL3	halfword multiply three arguments
W	MUL3	word multiply three arguments
F	MUL3	float multiply three arguments
D	MUL3	double float multiply three arguments
BY	DIV3	byte divide three arguments
H	DIV3	halfword divide three arguments
W	DIV3	word divide three arguments
F	DIV3	float divide three arguments
D	DIV3	double float divide three arguments
BYn	MUL4	byte multiply with overflow
Hn	MUL4	halfword multiply with overflow
Wn	MUL4	word multiply with overflow
BYn	DIV4	byte divide with remainder
Hn	DIV4	halfword divide with remainder
Wn	DIV4	word divide with remainder
Wn	UMUL	word unsigned multiplication
Wn	UDIV	word unsigned divide
Wn	ADDC	word add with carry
Wn	SUBC	word subtract with carry
BIn	CLR	bit register clear

APPENDIX E Instruction table

BYn CLR	byte register clear
Hn CLR	halfword register clear
Wn CLR	word register clear
Fn CLR	float register clear
Dn CLR	double float register clear
BI STZ	bit store zero
BY STZ	byte store zero
H STZ	halfword store zero
W STZ	word store zero
F STZ	float store zero
D STZ	double float store zero
BI SET1	bit set to one
BY SET1	byte set to one
H SET1	halfword set to one
W SET1	word set to one
F SET1	float set to one
D SET1	double float set to one
BY INCR	byte increment
H INCR	halfword increment
W INCR	word increment
F INCR	float increment
D INCR	double float increment
BY DECR	byte decrement
H DECR	halfword decrement
W DECR	word decrement
F DECR	float decrement
D DECR	double float decrement
BIn AND	bit and register
BYn AND	byte and register
Hn AND	halfword and register
Wn AND	word and register
BIn OR	bit or register
BYn OR	byte or register
Hn OR	halfword or register
Wn OR	word or register
BIn XOR	bit exclusive or register
BYn XOR	byte exclusive or register
Hn XOR	halfword exclusive or register
Wn XOR	word exclusive or register
BY SHL	byte shift logical
H SHL	halfword shift logical
W SHL	word shift logical
BY SHA	byte shift arithmetical
H SHA	halfword shift arithmetical
W SHA	word shift arithmetical
BY SHR	byte shift rotational
H SHR	halfword shift rotational

W	SHR	word shift rotational
BYn	GETBI	byte get bit
Hn	GETBI	halfword get bit
Wn	GETBI	word get bit
BYn	PUTBI	byte put bit
Hn	PUTBI	halfword put bit
Wn	PUTBI	word put bit
BY	CLEBI	byte clear bit
H	CLEBI	halfword clear bit
W	CLEBI	word clear bit
BY	SETBI	byte set bit
H	SETBI	halfword set bit
W	SETBI	word set bit
BYn	GETBF	byte get bit field
Hn	GETBF	halfword get bit field
Wn	GETBF	word get bit field
BYn	PUTBF	byte put bit field
Hn	PUTBF	halfword put bit field
Wn	PUTBF	word put bit field
Fn	AXI	register float <A> to the <I>'th power
Dn	AXI	register double float <A> to the <I>'th power
BYn	IXI	register byte <I> to the <J>'th power
Hn	IXI	register halfword <I> to the <J>'th power
Wn	IXI	register word <I> to the <J>'th power
Fn	SQRT	register float square root
Dn	SQRT	register double float square root
Fn	POLY	floating polynomial
Dn	POLY	double float polynomial
Fn	REM	float divide with remainder
Dn	REM	double float divide with remainder
Fn	INT	float integer part
Dn	INT	double float integer part
Fn	INTR	float integer part with rounding
Dn	INTR	double float integer part with rounding
BYn	MULAD	byte multiply and add
Hn	MULAD	halfword multiply and add
Wn	MULAD	word multiply and add
Fn	MULAD	float multiply and add
Dn	MULAD	double float multiply and add
BYn	PSUM	byte add and multiply

APPENDIX E Instruction table

Hn	PSUM	halfword add and multiply
Wn	PSUM	word add and multiply
Fn	PSUM	float add and multiply
Dn	PSUM	double float add and multiply
BYn	LIND	byte load index
Hn	LIND	halfword load index
Wn	LIND	word load index
BYn	CIND	byte calculate index
Hn	CIND	halfword calculate index
Wn	CIND	word calculate index

APPENDIX E Instruction table

CONTROL INSTRUCTIONS

GO:B	jump byte
GO:H	jump halfword
GO:W	jump word
JUMPG	jump general

IF = GO	Z=1	equal
IF Z GO		(alt. assembly notation)
IF = GO:B		byte displacement
IF = GO:H		halfword displacement
IF >< GO	Z=0	unequal
IF -Z GO		(alt. assembly notation)
IF >< GO:B		byte displacement
IF >< GO:H		halfword displacement
IF > GO	S=0 and Z=0	greater signed
IF > GO:B		
IF > GO:H		
IF < GO	S=1	less signed
IF S GO		(alt. assembly notation)
IF < GO:B		
IF < GO:H		
IF >= GO	S=0	greater or equal signed
IF -S GO		(alt. assembly notation)
IF >= GO:B		
IF >= GO:H		
IF <= GO	S=1 or Z=1	less or equal signed
IF <= GO:B		
IF <= GO:H		
IF K GO	K=1	flag
IF K GO:B		
IF K GO:H		
IF -K GO	K=0	not flag
IF -K GO:B		
IF -K GO:H		
IF >> GO	C=1 and Z=0	greater magnitude
IF >> GO:B		
IF >> GO:H		
IF >>= GO	C=1	greater or equal magnitude
IF C GO		(alt. assembly notation)
IF >>= GO:B		
IF >>= GO:H		
IF << GO	C=0	less magnitude
IF -C GO		(alt. assembly notation)

APPENDIX E Instruction table

IF << GO:B		
IF << GO:H		
IF <<= GO	C=0 or Z=1	less or equal magnitude
IF <<= GO:B		
IF <<= GO:H		
IF ST GO		specified bit in status register set
IF ST GO:B		
IF ST GO:H		
IF -ST GO		specified bit in status register not set
IF -ST GO:B		
IF -ST GO:H		

BY	LOOPI:B	byte loop increment
BY	LOOPI:H	byte loop increment
H	LOOPI:B	halfword loop increment
H	LOOPI:H	halfword loop increment
W	LOOPI:B	word loop increment
W	LOOPI:H	word loop increment
F	LOOPI:B	float loop increment
F	LOOPI:H	float loop increment
D	LOOPI:B	double float loop increment
D	LOOPI:H	double float loop increment

BY	LOOPD:B	byte loop decrement
BY	LOOPD:H	byte loop decrement
H	LOOPD:B	halfword loop decrement
H	LOOPD:H	halfword loop decrement
W	LOOPD:B	word loop decrement
W	LOOPD:H	word loop decrement
F	LOOPD:B	float loop decrement
F	LOOPD:H	float loop decrement
D	LOOPD:B	double float decrement
D	LOOPD:H	double float decrement

BY	LOOP:B	byte loop general step
BY	LOOP:H	byte loop general step
H	LOOP:B	halfword loop general step
H	LOOP:H	halfword loop general step
W	LOOP:B	word loop general step
W	LOOP:H	word loop general step
F	LOOP:B	float loop general step
F	LOOP:H	float loop general step
D	LOOP:B	double float loop general step
D	LOOP:H	double float loop general step

CALL	call subroutine absolute
CALLG	call subroutine general

APPENDIX E Instruction table

INIT	initialize stack
ENTM	enter module
ENTD	enter subroutine directly
ENTS	enter stack subroutine
ENTF	enter subroutine
ENTSN	enter max argument stack subroutine
ENTFN	enter max argument subroutine
ENTT	enter trap handler
ENTB	enter buddy subroutine
RET	clear flag return from subroutine
RETK	set flag return from subroutine
RETD	return from direct subroutine
RETT	trap handler return
IF K RET	if flag set subroutine return
RETB	buddy subroutine return
RETBK	set flag buddy subroutine return

APPENDIX E Instruction table

STRING INSTRUCTIONS

BI	SMOVE	bit string move
BY	SMOVE	byte string move
H	SMOVE	halfword string move
W	SMOVE	word string move
F	SMOVE	float string move
D	SMOVE	double float string move
BY	SMVWH	byte move string while
BY	SMVUN	byte move string until
BY	SMVTR	move translated string
BY	SMVTU	move string translated until
BI	SMOVN	string move n bits
BY	SMOVN	string move n bytes
H	SMOVN	string move n halfwords
W	SMOVN	string move n words
F	SMOVN	string move n floats
D	SMOVN	string move n double floats
BIn	SFILL	bit string fill
Bn	SFILL	byte string fill
Hn	SFILL	halfword string fill
Wn	SFILL	word string fill
Fn	SFILL	float string fill
Dn	SFILL	double float string fill
BIn	SFILLN	string fill n bits
BYn	SFILLN	string fill n bytes
Hn	SFILLN	string fill n halfwords
Wn	SFILLN	string fill n words
Fn	SFILLN	string fill n floats
Dn	SFILLN	string fill n double floats
BY	SCOMP	string compare
BY	SCOTR	string compare translated
BY	SCOPA	string compare with pad
BY	SCOPT	string compare translated with pad
BY	SSKIP	skip elements
BI	SLOCA	string locate bit
BY	SLOCA	string locate byte
BY	SSCAN	string scan
BY	SSPAN	string span
BY	SMATCH	string match
BY	SSPAR	set parity in string
BY	SCHPAR	check parity in string

APPENDIX E Instruction table

MISCELLANEOUS INSTRUCTIONS

BY	BMOVE	byte block move
H	BMOVE	halfword block move
W	BMOVE	word block move
F	BMOVE	float block move
D	BMOVE	double float block move
BI	BYCONV	bit to byte convert
BI	HCONV	bit to halfword convert
BI	WCONV	bit to word convert
BI	FCONV	bit to float convert
BI	DCONV	bit to double float convert
BY	BICONV	byte to bit convert
BY	HCONV	byte to halfword convert
BY	WCONV	byte to word convert
BY	FCONV	byte to float convert
BY	DCONV	byte to double float convert
H	BICONV	halfword to bit convert
H	BYCONV	halfword to byte convert
H	WCONV	halfword to word convert
H	FCONV	halfword to float convert
H	DCONV	halfword to double float convert
W	BICONV	word to bit convert
W	BYCONV	word to byte convert
W	HCONV	word to halfword convert
W	FCONV	word to float convert
W	DCONV	word to double float convert
F	BICONV	float to bit convert
F	BYCONV	float to byte convert
F	HCONV	float to halfword convert
F	WCONV	float to word convert
F	DCONV	float to double float convert
D	BICONV	double float to bit convert
D	BYCONV	double float to byte convert
D	HCONV	double float to halfword convert
D	WCONV	double float to word convert
D	FCONV	double float to float convert
F	BYCONR	float to byte convert with rounding
D	BYCONR	double float to byte convert with rounding
F	HCONR	float to halfword convert with rounding
D	HCONR	double float to halfword convert with rounding
F	WCONR	float to word convert with rounding
D	WCONR	double float to word convert

APPENDIX E Instruction table

		with rounding
W	FCONR	word to float convert with rounding
D	FCONR	double float to float convert with rounding
BIn	LADDR	bit load address
BYn	LADDR	byte load address
Hn	LADDR	halfword load address
Wn	LADDR	word load address
Fn	LADDR	float load address
Dn	LADDR	double float load address
BI	RLADDR	bit load address record
BY	RLADDR	byte load address record
H	RLADDR	halfword load address record
W	RLADDR	word load address record
F	RLADDR	float load address record
D	RLADDR	double float load address record
BI	BLADDR	bit load address local
BY	BLADDR	byte load address local
H	BLADDR	halfword load address local
W	BLADDR	word load address local
F	BLADDR	float load address local
D	BLADDR	double float load address local
Wn	CHAIN	load address of multilevel link
	NOOP	no operation
	SETK	set flag
	CLRK	clear flag
Wn	GETB	get buddy
	FREEB	free buddy

APPENDIX E Instruction table

SPECIAL INSTRUCTIONS

	SOLO	disable process switch
	TUTTI	enable process switch
	SETE	set bit in trap enable register
	CLTE	clear bit in trap enable register
	BP	break point instruction
BYn	TSET	test and set
	L :=	load link register
	HL :=	load upper limit register
	LL :=	load lower limit register
	ST1 :=	load first status register
	OTE1 :=	load first own trap enable register
	OTE2 :=	load second own trap enable register
	TOS :=	load top of stack register
	THA :=	load trap handler register
	L =:	store link register
	HL =:	store upper limit register
	LL =:	store lower limit register
	ST1 =:	store first status register
	OTE1 =:	store first own trap enable register
	OTE2 =:	store second own trap enable register
	MTE2 =:	store first mother trap enable register
	MTE1 =:	store second mother trap enable register
	CTE1 =:	store first child trap enable register
	CTE2 =:	store second child trap enable register
	TEMM1 =:	store first trap enable modification mask
	TEMM2 =:	store second trap enable modification mask
	CED =:	store current executing domain register
	CAD =:	store current alternative domain register
	CES =:	store current executing segment register
	CAS =:	store current segment on alternative domain
	PS =:	store process segment register
	TOS =:	store top of stack register
	THA =:	store trap handler register
	P =:	store program counter
	An :=	load most significant part of double float register
	En :=	load least significant part of double float register
	An =:	store most significant part of double float register
	En =:	store least significant part of double float register

APPENDIX F Alphabetical instruction table

Legal data formats	Assembly notation	Name
BY H W F D	tn *	multiply
BY H W F D	tn +	add
BY H W F D	tn -	subtract
BY H W F D	tn /	divide
BI BY H W F D	tn :=	load
BI BY H W F D	tn =:	store
BY H W F D	tn ABS	absolute value
BY H W F D	t ADD2	add two arguments
BY H W F D	t ADD3	add three arguments
W	t ADDC	add with carry
BI BY H W	tn AND	AND register
F D	tn AXI	register <A> to the <I>'th power
	An :=	load most significant part of double float reg
	An =:	store most significant part of double float reg
	B :=	load local base
	B =:	local base store
BI BY H W F D	t BLADDR	load address local
BY H W F D	t BMOVE	block move
	BP	break point instruction
BI H W F D	t BYCONR	convert to byte with rounding
BI H W F D	t BYCONV	convert to byte
	CAD =:	store alternative domain register
	CAS =:	store current segment alternative domain
	CALL	call subroutine absolute
	CALLG	call subroutine general
	CED =:	store current executing domain register
	CES =:	store current executing segment register
W	tn CHAIN	load address of multilevel link
BY H W	tn CIND	calculate index
BY H W	t CLEBI	clear bit
BI BY H W F D	tn CLR	register clear
	CLRK	clear flag
	CLTE	clear bit in trap enable register
BI BY H W F D	tn COMP	register compare
BI BY H W F D	t COMP2	compare
	CTE1 =:	store first child trap enable register
	CTE2 =:	store second child trap enable register
BI BY H W F	t DCONV	convert to double float
BY H W F D	t DECR	decrement
BY H W F D	t DIV2	divide two arguments
BY H W F D	t DIV3	divide three arguments
BY H W F D	tn DIV4	divide with remainder
	ENTB	enter buddy subroutine
	ENTD	enter subroutine directly
	ENTF	enter subroutine

APPENDIX F Alphabetical instruction table

			ENTFN	enter max argument subroutine
			ENTM	enter module
			ENTS	enter stack subroutine
			ENTSN	enter max argument stack subroutine
			ENTT	enter trap handler
			En :=	load least significant part of double float register
			En =:	store least significant part of double float register
	W	D	t FCONR	convert to float with rounding
BI BY H	W	D	t FCONV	convert to float
			FREEB	free buddy
	W		t GETB	get buddy
BY H	W		tn GETBF	get bit field
BY H	W		tn GETBI	get bit
			GO:B	jump byte
			GO:H	jump halfword
			GO:W	jump word
	F	D	t HCONR	convert to halfword with rounding
BI BY	W	F D	t HCONV	convert to halfword
			HL :=	load upper limit register
			HL =:	store upper limit register
BY H			IF -ST GO:t	jump if specified status bit not set
BY H			IF -C GO:t	jump if magnitude less
BY H			IF -K GO:t	jump if flag not set
BY H			IF -S GO:t	jump if signed greater or equal
BY H			IF -Z GO:t	jump if not equal
BY H			IF <rel> GO:t	jump if relation true
BY H			IF C GO:t	jump if magnitude greater or equal
BY H			IF K GO:t	jump if flag set
BY H			IF K RET	subroutine return if flag set
BY H			IF S GO:t	jump if signed less
BY H			IF ST GO:t	jump if specified status bit set
BY H			IF Z GO:t	jump if equal
BY H W	F	D	t INCR	increment
			INIT	initialize stack
	F	D	tn INT	float integer part
	F	D	tn INTR	float integer part with rounding
BI BY H	W		tn INV	invert register
	W		tn INVC	word invert register with carry
	F	D	tn IXI	register I to the <J>'th power
			JUMPG	jump general
			L :=	load link register
			L =:	store link register
BI BY H W	F	D	tn LADDR	load address
BY H	W		tn LIND	load index
			LL :=	load lower limit register
			LL =:	store lower limit register
BY H W	F	D	t LOOP:B	loop general step
BY H W	F	D	t LOOP:H	loop general step
BY H W	F	D	t LOOPD:B	loop decrement
BY H W	F	D	t LOOPD:H	loop decrement
BY H W	F	D	t LOOPI:B	loop increment
BY H W	F	D	t LOOPI:H	loop increment
BI BY H W	F	D	t MOVE	move

APPENDIX F Alphabetical instruction table

		MTE1 :=	load first mother trap enable register
		MTE1 =:	store first mother trap enable register
		MTE2 :=	load second mother trap enable register
		MTE2 =:	store second mother trap enable register
BY H W F D	t	MUL2	multiply two arguments
BY H W F D	t	MUL3	multiply three arguments
BY H W F D	tn	MUL4	multiply with overflow
BY H W F D	tn	MULAD	multiply and add
BY H W F D	tn	NEG	register negate
		NOOP	no operation
BI BY H W	tn	OR	OR register
		OTE1 :=	load first own trap enable register
		OTE1 =:	store first own trap enable register
		OTE2 :=	load second own trap enable register
		OTE2 =:	store second own trap enable register
		P =:	store program counter
F D	tn	POLY	polynomial
		PS =:	store process segment register
BY H W F D	tn	PSUM	add and multiply
BY H W	tn	PUTBF	put bit field
BY H W	tn	PUTBI	put bit
		R :=	load record base
		R =:	record base store
F D	tn	REM	divide with remainder
		RET	clear flag return from subroutine
		RETB	buddy subroutine return
		RETBK	set flag buddy subroutine return
		RETD	return from direct subroutine
		RETK	set flag subroutine return
		RETT	trap handler return
BI BY H W F D	t	RLADDR	load address record
BY	t	SCHPAR	check parity in string
BY	t	SCOMP	string compare
BY	t	SCOPA	string compare with pad
BY	t	SCOPT	string compare translated with pad
BY	t	SCOTR	string compare translated
BI BY H W F D	t	SET1	set to one
BY H W	t	SETBI	set bit
		SETE	set bit in trap enable register
		SETK	set flag
BI BY H W F D	tn	SFILL	string fill
BI BY H W F D	tn	SFILLN	string fill n elements
BY H W	t	SHA	shift arithmetical
BY H W	t	SHL	shift logical
BY H W	t	SHR	shift rotational
BI BY	t	SLOCA	string locate
BY	t	SMATCH	string match
BI BY H W F D	t	SMOVE	string move
BI BY H W F D	t	SMOVN	string move n elements
BY	t	SMVTR	move translated string
BY	t	SMVTU	move string translated until
BY	t	SMVUN	move string until
BY	t	SMVWH	move string while
		SOLO	disable process switch
F D	tn	SQRT	register square root
BY	t	SSCAN	string scan
BY	t	SSKIP	skip elements

APPENDIX F Alphabetical instruction table

	BY					t	SSPAN	string span
	BY					t	SSPAR	set parity in string
							ST1 :=	load first status register
							ST1 =:	store first status register
BI	BY	H	W	F	D	t	STZ	store zero
	BY	H	W	F	D	t	SUB2	subtract two arguments
	BY	H	W	F	D	t	SUB3	subtract three arguments
			W			tn	SUBC	subtract with carry
BI	BY	H	W	F	D	t	SWAP	swap
							TEMM1 =:	store 1st trap enable modification mask
							TEMM2 =:	store 2nd trap enable modification mask
BI	BY	H	W	F	D	t	TEST	test against zero
							THA :=	load trap handler register
							THA =:	store trap handler register
							TOS :=	load top of stack register
							TOS =:	store top of stack register
			W			tn	TSET	test and set
							TUTTI	enable process switch
			W			tn	UDIV	unsigned divide
			W			tn	UMUL	unsigned multiply
BI	BY	H		F	D	t	WCONR	convert to word with rounding
BI	BY	H		F	D	t	WCONV	convert to word
BI	BY	H	W			tn	XOR	eXclusive OR register

APPENDIX G Instruction code table

APPENDIX G Instruction code table

This table gives an overview of the octal codes for the various instructions. A blank indicates that that data type may not be used for that instruction. The column REF. gives the cross reference number used in appendix H.

	BI	BY	H	W	F	D	REF.	MANUAL
tn :=	176004	004	010	014	020	024	1	10.1
B :=				176010			2	10.2
R :=				030			3	10.3
tn =:	176014	034	176020	040	044	050	4	10.4
B =:				176012			5	10.5
R =:				176011			6	10.6
t MOVE	176013	031	176024	032	033	054	7	10.7
t SWAP	176275	176276	176277	122	176334	176335	8	10.8
tn COMP	176030	060	176034	064	070	074	9	10.9
t COMP2	176025	055	176026	056	057	100	10	10.10
t TEST	101	102	103	104	105	106	11	10.11
tn NEG		177010	177014	220	224	224	12	10.12
tn INV	177020	177024	177030	230			13	10.13
tn INVC				177420			14	10.14
tn ABS		177400	177404	177410	177414	177414	15	10.15
tn +		176064	176070	124	130	134	16	10.16
tn -		176074	176100	140	144	150	17	10.17
tn *		176104	176110	154	160	164	18	10.18
tn /		176114	176120	170	174	350	19	10.19
t ADD2		176027	176124	123	176126	176127	20	10.20
t SUB2		176130	176131	340	176133	176134	21	10.21
t MUL2		176135	176136	176137	176140	176141	22	10.22
t DIV2		176142	176143	176144	176145	176146	23	10.23
t ADD3		176147	176150	176151	176152	176153	24	10.24
t SUB3		176154	176155	176156	176157	176160	25	10.25
t MUL3		176161	176162	176163	176164	176165	26	10.26
t DIV3		176166	176167	176170	176171	176172	27	10.27
tn MUL4		176040	176044	176050			28	10.28
tn DIV4		176054	176060	176174			29	10.29
tn UMUL				176200			30	10.30
tn UDIV				177110			31	10.31
tn ADDC				177100			32	10.32
tn SUBC				177104			33	10.33
tn CLR	204	204	204	204	210	214	34	10.34
t STZ	176205	110	111	112	113	114	35	10.35
t SET1	176206	176207	176210	115	107	176211	36	10.36
t INCR		176212	116	117	120	176213	37	10.37
t DECR		176214	176215	121	176216	176217	38	10.38
tn AND	176714	176220	176224	344			39	10.39
tn OR	176770	176230	176234	240			40	10.40
tn XOR	176774	176240	176244	244			41	10.41
t SHL		176250	176251	176252			42	10.42
t SHA		176253	176254	176255			43	10.43
t SHR		176256	176257	176260			44	10.44
tn GETBI		176264	176270	176720			45	10.45
tn PUTBI		176724	176730	176734			46	10.46
t CLEBI		177175	177176	177177			47	10.47
t SETBI		177200	177201	177202			48	10.48
tn GETBF		176740	176744	176750			49	10.49
tn PUTBF		176754	176760	176764			50	10.50
tn AXI					176300	176304	51	10.51
tn IXI		176310	176314	176320			52	10.52
tn SQRT					176324	176330	53	10.53
tn POLY					176340	176344	54	10.54
tn REM					177130	177134	55	10.55
tn INT					177140	177144	56	10.56
tn INTR					177150	177154	57	10.57
tn MULAD		176350	176354	250	176360	176364	58	10.58
tn PSUM		176370	176374	176400	176404	176410	59	10.59
tn LIND		176414	176420	254			60	10.60
tn CIND		176424	176430	260			61	10.61
:B GO				300			62	11.1
:H GO				301			63	11.1
:W GO				302			64	11.1
JUMPG				264			65	11.2
:B IF = GO				304			66	11.3

APPENDIX G Instruction code table

	BI	BY	H	W	F	D	REF.	MANUAL
:H IF = GO				305			67	11.3
:B IF >< GO				306			68	11.3
:H IF >< GO				307			69	11.3
:B IF > GO				310			70	11.3
:H IF > GO				311			71	11.3
:B IF < GO				312			72	11.3
:H IF < GO				313			73	11.3
:B IF >= GO				314			74	11.3
:H IF >= GO				315			75	11.3
:B IF <= GO				316			76	11.3
:H IF <= GO				317			77	11.3
:B IF K GO				320			78	11.3
:H IF K GO				321			79	11.3
:B IF -K GO				322			80	11.3
:H IF -K GO				323			81	11.3
:B IF >> GO				324			82	11.3
:H IF >> GO				325			83	11.3
:B IF >>= GO				326			84	11.3
:H IF >>= GO				327			85	11.3
:B IF << GO				330			86	11.3
:H IF << GO				331			87	11.3
:B IF <<= GO				332			88	11.3
:H IF <<= GO				333			89	11.3
:B IF ST GO				176173			90	11.3
:H IF ST GO				176544			91	11.3
:B IF -ST GO				176545			92	11.3
:H IF -ST GO				176204			93	11.3
:B t LOOPI		176336	176337	277	176434	176435	94	11.4
:H t LOOPI		176436	176437	341	176441	176442	95	11.4
:B t LOOPD		176443	176444	176445	176446	176447	96	11.5
:H t LOOPD		176450	176451	176452	176453	176454	97	11.5
:B t LOOP		176455	176456	176457	176460	176461	98	11.6
:H t LOOP		176462	176463	176464	176465	176466	99	11.6
CALL				303			100	11.7
CALLG				265			101	11.8
INIT				334			102	11.9
ENTM				337			103	11.10
ENTD				234			104	11.10
ENTS				270			105	11.10
ENTF				335			106	11.10
ENTSN				272			107	11.10
ENTFN				336			108	11.10
ENTT				274			109	11.10
ENTB				275			110	11.10
RET				200			111	11.11
RETK				201			112	11.11
RETB				177034			113	11.11
RETBK				177035			114	11.11
RETD				202			115	11.11
RETT				203			116	11.11
IF K RET				235			117	11.11
t SMOVE	176546	176547	176550	176551	176552	176553	118	12.2
t SMVWH		176562					119	12.3
t SMVUN		176563					120	12.4
t SMVTR		176564					121	12.5
t SMVTU		176565					122	12.6
t SMOVN	176566	176567	176570	176571	176572	176573	123	12.7
tn SFILL	176574	176600	176604	176610	176614	176620	124	12.8
tn SFILLN	176624	176630	176634	176640	176644	176650	125	14.9
t SCOMP		176654					126	12.10
t SCOTR		176655					127	12.11
t SCOPA		176676					128	12.12
t SCOPT		176677					129	12.13
t SSKIP		176656					130	12.14
t SLOCA	176657	176660					131	12.15
t SSCAN		176661					132	12.16

APPENDIX G Instruction code table

		BI	BY	H	W	F	D	REF.	MANUAL
t	SSPAN		176662					133	12.17
t	SMATCH		176663					134	12.18
t	SSPAR		176664					135	12.19
t	SCHPAR		176665					136	12.20
t	BMOVE		176440	177170	177171	177172	177173	137	13.1
t	BICONV		176511	176516	176523	176530	176535	138	13.2
t	BYCONV	176504		176517	176524	176531	176536	139	13.2
t	HCONV	176505	176512		176525	176532	176537	140	13.2
t	WCONV	176506	176513	176520		176533	176540	141	13.2
t	FCONV	176507	176514	176521	176526		176541	142	13.2
t	DCONV	176510	176515	176522	176527	176534		143	13.2
t	BYCONR					177160	177161	144	13.3
t	HCONR					177162	177163	145	13.3
t	WCONR					177164	177165	146	13.3
t	FCONR				177203		177204	147	13.3
tn	LADDR	177040	177044	177050	176474	176474	177054	148	13.4
t	RLADDR	176125	176132	176261	276	276	176262	149	13.5
t	BLADDR	176263	176274	176467	176543	176543	176470	150	13.6
tn	CHAIN				176554			151	13.7
	NOOP				003			152	13.8
	SETK				177002			153	13.9
	CLRK				177003			154	13.10
Wn	GETB				177114			155	13.11
	FREEB				176666			156	13.12
	SOLO				177000			157	14.1
	TUTTI				177001			158	9.2
	SETE				176471			159	14.3
	CLTE				176472			160	14.4
	BP				002			161	14.5
t	TSET				176500			162	14.6
L	:=				176473			163	14.7
HL	:=				176667			164	14.7
LL	:=				176670			165	14.7
ST1	:=				176671			166	14.7
OTE1	:=				176673			167	14.7
OTE2	:=				176674			168	14.7
TOS	:=				176675			169	14.7
THA	:=				176712			170	14.7
L	=:				176700			171	14.8
HL	=:				176701			172	14.8
LL	=:				176702			173	14.8
ST1	=:				176703			174	14.8
OTE1	=:				176705			175	14.8
OTE2	=:				176706			176	14.8
MTE1	=:				176560			177	14.8
MTE2	=:				176561			178	14.8
CTE1	=:				177120			179	14.8
CTE2	=:				177121			180	14.8
TEMM1	=:				177122			181	14.8
TEMM2	=:				177123			182	14.8
CED	=:				177124			183	14.8
CAD	=:				177125			184	14.8
CES	=:				177126			185	14.8
CAS	=:				177127			186	14.8
PS	=:				177174			187	14.8
TOS	=:				176711			188	14.8
THA	=:				176713			189	14.8
P	=:				176542			190	14.8
An	:=				177060			191	14.9
En	:=				177064			192	14.9
An	=:				177070			193	14.9
En	=:				177074			194	14.9
	illeg.1				000			195	-
	illeg.2				001			196	-
n	exten.				374			197	-

APPENDIX H Instruction code cross reference table

This table lists all octal instruction code values. For each value a reference number to the instruction code table in appendix G is given. The instruction name can then be found by consulting appendix G.

	0	1	2	3	4	5	6	7
000000	0	196W	161W	152W	1BY	1BY	1BY	1BY
000010	1H	1H	1H	1H	1W	1W	1W	1W
000020	1F	1F	1F	1F	1D	1D	1D	1D
000030	3W	7BY	7W	7F	4BY	4BY	4BY	4BY
000040	4W	4W	4W	4W	4F	4F	4F	4F
000050	4D	4D	4D	4D	7D	10BY	10W	10F
000060	9BY	9BY	9BY	9BY	9W	9W	9W	9W
000070	9F	9F	9F	9F	9D	9D	9D	9D
000100	10D	11BI	11BY	11H	11W	11F	11D	36F
000110	35BY	35H	35W	35F	35D	36W	37H	37W
000120	37F	38W	8W	20W	16W	16W	16W	16W
000130	16F	16F	16F	16F	16D	16D	16D	16D
000140	17W	17W	17W	17W	17F	17F	17F	17F
000150	17D	17D	17D	17D	18W	18W	18W	18W
000160	18F	18F	18F	18F	18D	18D	18D	18D
000170	19W	19W	19W	19W	19F	19F	19F	19F
000200	111W	112W	115W	116W	34W *	34W *	34W *	34W *
000210	34F	34F	34F	34F	34D	34D	34D	34D
000220	12W	12W	12W	12W	12D *	12D *	12D *	12D *
000230	13W	13W	13W	13W	104W	117W	0	0
000240	40W	40W	40W	40W	41W	41W	41W	41W
000250	58W	58W	58W	58W	60W	60W	60W	60W
000260	61W	61W	61W	61W	65W	101W	0	0
000270	105W	0	107W	0	109W	110W	149F *	94W
000300	62W	63W	64W	100W	66W	67W	68W	69W
000310	70W	71W	72W	73W	74W	75W	76W	77W
000320	78W	79W	80W	81W	82W	83W	84W	85W
000330	86W	87W	88W	89W	102W	106W	108W	103W
000340	21W	95W	0	0	39W	39W	39W	39W
000350	19D	19D	19D	19D	0	0	0	0
000360	0	0	0	0	0	0	0	0
000370	0	0	0	0	197W	197W	197W	197W
176000	0	0	0	0	1BI	1BI	1BI	1BI
176010	2W	6W	5W	7BI	4BI	4BI	4BI	4BI
176020	4H	4H	4H	4H	7H	10BI	10H	20BY
176030	9BI	9BI	9BI	9BI	9H	9H	9H	9H
176040	28BY	28BY	28BY	28BY	28H	28H	28H	28H
176050	28W	28W	28W	28W	29BY	29BY	29BY	29BY
176060	29H	29H	29H	29H	16BY	16BY	16BY	16BY
176070	16H	16H	16H	16H	17BY	17BY	17BY	17BY
176100	17H	17H	17H	17H	18BY	18BY	18BY	18BY
176110	18H	18H	18H	18H	19BY	19BY	19BY	19BY
176120	19H	19H	19H	19H	20H	149BI	20F	20D
176130	21BY	21H	149BY	21F	21D	22BY	22H	22W
176140	22F	22D	23BY	23H	23W	23F	23D	24BY
176150	24H	24W	24F	24D	25BY	25H	25W	25F
176160	25D	26BY	26H	26W	26F	26D	27BY	27H
176170	27W	27F	27D	90W	29W	29W	29W	29W
176200	30W	30W	30W	30W	93W	35BI	36BI	36BY
176210	36H	36D	37BY	37D	38BY	38H	38F	38D
176220	39BY	39BY	39BY	39BY	39H	39H	39H	39H
176230	40BY	40BY	40BY	40BY	40H	40H	40H	40H
176240	41BY	41BY	41BY	41BY	41H	41H	41H	41H
176250	42BY	42H	42W	43BY	43H	43W	44BY	44H
176260	44W	149H	149D	150BI	45BY	45BY	45BY	45BY
176270	45H	45H	45H	45H	150BY	8BI	8BY	8H
176300	51F	51F	51F	51F	51D	51D	51D	51D
176310	52BY	52BY	52BY	52BY	52H	52H	52H	52H
176320	52W	52W	52W	52W	53F	53F	53F	53F
176330	53D	53D	53D	53D	8F	8D	94BY	94H
176340	54F	54F	54F	54F	54D	54D	54D	54D
176350	58BY	58BY	58BY	58BY	58H	58H	58H	58H
176360	58F	58F	58F	58F	58D	58D	58D	58D
176370	59BY	59BY	59BY	59BY	59H	59H	59H	59H
176400	59W	59W	59W	59W	59F	59F	59F	59F
176410	59D	59D	59D	59D	60BY	60BY	60BY	60BY

APPENDIX H Instruction code cross reference table

	0	1	2	3	4	5	6	7
176420	60H	60H	60H	60H	61BY	61BY	61BY	61BY
176430	61H	61H	61H	61H	94F	94D	95BY	95H
176440	137BY	95F	95D	96BY	96H	96W	96F	96D
176450	97BY	97H	97W	97F	97D	98BY	98H	98W
176460	98F	98D	99BY	99H	99W	99F	99D	150H
176470	150D	159W	160W	163W	148F *	148F *	148F *	148F *
176500	162W	0	0	0	139BI	140BI	141BI	142BI
176510	143BI	138BY	140BY	141BY	142BY	143BY	138H	139H
176520	141H	142H	143H	138W	139W	140W	142W	143W
176530	138F	139F	140F	141F	143F	138D	139D	140D
176540	141D	142D	190W	150F *	91W	92W	118BI	118BY
176550	118H	118W	118F	118D	151W	151W	151W	151W
176560	177W	178W	119BY	120BY	121BY	122BY	123BI	123BY
176570	123H	123W	123F	123D	124BI	124BI	124BI	124BI
176600	124BY	124BY	124BY	124BY	124H	124H	124H	124H
176610	124W	124W	124W	124W	124F	124F	124F	124F
176620	124D	124D	124D	124D	125BI	125BI	125BI	125BI
176630	125BY	125BY	125BY	125BY	125H	125H	125H	125H
176640	125W	125W	125W	125W	125F	125F	125F	125F
176650	125D	125D	125D	125D	126BY	127BY	130BY	131BI
176660	131BY	132BY	133BY	134BY	135BY	136BY	156W	164W
176670	165W	166W	0	167W	168W	169W	128BY	129BY
176700	171W	172W	173W	174W	0	175W	176W	0
176710	0	188W	170W	189W	39BI	39BI	39BI	39BI
176720	45W	45W	45W	45W	46BY	46BY	46BY	46BY
176730	46H	46H	46H	46H	46W	46W	46W	46W
176740	49BY	49BY	49BY	49BY	49H	49H	49H	49H
176750	49W	49W	49W	49W	50BY	50BY	50BY	50BY
176760	50H	50H	50H	50H	50W	50W	50W	50W
176770	40BI	40BI	40BI	40BI	41BI	41BI	41BI	41BI
177000	157W	158W	153W	154W	0	0	0	0
177010	12BY	12BY	12BY	12BY	12H	12H	12H	12H
177020	13BI	13BI	13BI	13BI	13BY	13BY	13BY	13BY
177030	13H	13H	13H	13H	113W	114W	0	0
177040	148BI	148BI	148BI	148BI	148BY	148BY	148BY	148BY
177050	148H	148H	148H	148H	148D	148D	148D	148D
177060	191W	191W	191W	191W	192W	192W	192W	192W
177070	193W	193W	193W	193W	194W	194W	194W	194W
177100	32W	32W	32W	32W	33W	33W	33W	33W
177110	31W	31W	31W	31W	155W	155W	155W	155W
177120	179W	180W	181W	182W	183W	184W	185W	186W
177130	55F	55F	55F	55F	55D	55D	55D	55D
177140	56F	56F	56F	56F	56D	56D	56D	56D
177150	57F	57F	57F	57F	57D	57D	57D	57D
177160	144F	144D	145F	145D	146F	146D	0	0
177170	137H	137W	137F	137D	187W	47BY	47H	47W
177200	48BY	48H	48W	147W	147D	0	0	0
177210	0	0	0	0	0	0	0	0
177220	0	0	0	0	0	0	0	0
177230	0	0	0	0	0	0	0	0
177240	0	0	0	0	0	0	0	0
177250	0	0	0	0	0	0	0	0
177260	0	0	0	0	0	0	0	0
177270	0	0	0	0	0	0	0	0
177300	0	0	0	0	0	0	0	0
177310	0	0	0	0	0	0	0	0
177320	0	0	0	0	0	0	0	0
177330	0	0	0	0	0	0	0	0
177340	0	0	0	0	0	0	0	0
177350	0	0	0	0	0	0	0	0
177360	0	0	0	0	0	0	0	0
177370	0	0	0	0	0	0	0	0
177400	15BY	15BY	15BY	15BY	15H	15H	15H	15H
177410	15W	15W	15W	15W	15D *	15D *	15D *	15D *
177420	14W	14W	14W	14W	0	0	0	0
177430	0	0	0	0	0	0	0	0

APPENDIX I Setting of status bits

This table indicates which status bits may be modified and which trap conditions may be invoked by the various instructions. Be aware that some instructions will unconditionally set or clear various status bits, regardless of the result or operand value.

	PS D	Z	C	S.K	O	IV O	D.F Z.U	F O	B O	I.S O.I V.T	B	C	B.A P.T T.F	A T R	A T W	A A.D Z.R	I X	S T O	S.P T.R U.T	D T	D E	X.I S.I E.C	I O S	I S E	I P V
*		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
+		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
-		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
/		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
:=		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
=:		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
ABS		*	*	*		*	*			*			*						*	*	*				
ADD2		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
ADD3		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
ADDC		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
AND		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
AXI		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
An :=		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
An =:		*	*	*		*	*	*		*			*		*	*	*		*	*	*	*	*		*
B :=		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
B =:		*	*	*		*	*	*		*			*		*	*	*		*	*	*	*	*		*
BLADDR		*	*	*		*	*	*		*			*	*		*	*		*	*	*				*
BMOVE		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
BP										*			*						*	*	*		*		
BYCONR		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
BYCONV		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
CALL										*	*	*	*						*	*	*			*	*
CALLG										*	*	*	*	*		*	*		*	*	*	*	*	*	*
CHAIN		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
CIND		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
CLEBI		*	*	*		*	*	*	*	*			*	*	*	*	*		*	*	*	*	*		*
CLR		*	*	*		*	*	*		*			*						*	*	*				
CLRK					*					*			*						*	*	*	*	*		
CLTE									*	*			*	*		*	*		*	*	*	*	*		*
COMP		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
COMP2		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
CTE1 :=		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
CTE1 =:		*	*	*		*	*	*		*			*		*	*	*		*	*	*	*	*		*
CTE2 :=		*	*	*		*	*	*		*			*	*		*	*		*	*	*	*	*		*
CTE2 =:		*	*	*		*	*	*		*			*		*	*	*		*	*	*	*	*		*
DCONV		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
DECR		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
DIV2		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
DIV3		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*
DIV4		*	*	*		*	*	*		*			*	*	*	*	*		*	*	*	*	*		*

APPENDIX I Setting of status bits

....	PSD	Z	C	S	K	O	IVO	DZ	FU	FO	BO	IOV	SIT	BT	CT	BPT	ATF	ATR	ATW	AZ	DR	IX	STO	STU	PRT	DT	DE	XSE	IIC	IOS	ISE	PV
ENTB													*	*			*	*		*			*		*	*	*	*		*	*	*
ENTD													*	*			*	*							*	*	*				*	*
ENTF													*	*			*	*		*					*	*	*	*		*	*	*
ENTFN													*	*			*	*		*					*	*	*	*		*	*	*
ENTM													*	*			*	*		*			*		*	*	*	*		*	*	*
ENTS													*	*			*	*		*			*		*	*	*	*		*	*	*
ENTSN													*	*			*	*		*			*		*	*	*	*		*	*	*
ENTT													*	*			*	*		*			*		*	*	*	*		*	*	*
En :=		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
En =:		*	*	*		*		*	*	*	*		*				*		*	*	*	*			*	*	*	*		*		*
FCONR		*	*	*		*		*	*	*	*		*				*	*	*	*	*	*			*	*	*	*		*		*
FCONV		*	*	*		*		*	*	*	*		*				*	*	*	*	*	*			*	*	*	*		*		*
FREEB													*				*		*	*	*	*			*	*	*	*		*		*
GETB													*				*	*		*	*	*	*		*	*	*	*		*		*
GETBF		*	*	*		*		*	*	*	*	*	*				*	*		*	*	*			*	*	*	*		*		*
GETBI		*	*	*		*		*	*	*	*	*	*				*	*		*	*	*			*	*	*	*		*		*
GO:B													*	*			*			*					*	*	*				*	*
GO:H													*	*			*			*					*	*	*				*	*
GO:W													*	*			*			*					*	*	*				*	*
HCONR		*	*	*		*		*	*	*	*		*				*	*	*	*	*	*			*	*	*	*		*		*
HCONV		*	*	*		*		*	*	*	*		*				*	*	*	*	*	*			*	*	*	*		*		*
HL :=		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
HL =:		*	*	*		*		*	*	*	*		*				*		*	*	*	*			*	*	*	*		*		*
IF C GO													*	*			*			*					*	*	*				*	
IF -C GO													*	*			*			*					*	*	*				*	
IF -K GO													*	*			*			*					*	*	*				*	
IF -S GO													*	*			*			*					*	*	*				*	
IF -ST GO													*	*			*			*					*	*	*				*	
IF -Z GO													*	*			*			*					*	*	*				*	
IF <rel> GO													*	*			*			*					*	*	*				*	
IF K GO													*	*			*			*					*	*	*				*	
IF K RET													*	*			*			*					*	*	*				*	
IF S GO													*	*			*			*					*	*	*				*	
IF ST GO													*	*			*			*					*	*	*				*	
IF Z GO													*	*			*			*					*	*	*				*	
INCR		*	*	*		*		*	*	*	*		*				*	*	*	*	*	*			*	*	*	*		*		*
INIT													*				*	*		*	*	*			*	*	*	*		*		*
INT		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
INTR		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
INV		*	*	*		*		*	*	*	*		*				*								*	*	*					
INVC		*	*	*		*		*	*	*	*		*				*								*	*	*					
IXI		*	*	*		*		*	*	*	*	*	*				*	*		*	*	*			*	*	*	*		*		*
JUMPG													*	*			*	*		*	*	*			*	*	*	*		*	*	*
L :=		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
L =:		*	*	*		*		*	*	*	*		*				*		*	*	*	*			*	*	*	*		*		*
LADDR		*	*	*		*		*	*	*	*		*				*								*	*	*	*		*		*
LIND		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
LL :=		*	*	*		*		*	*	*	*		*				*	*		*	*	*			*	*	*	*		*		*
LL =:		*	*	*		*		*	*	*	*		*				*		*	*	*	*			*	*	*	*		*		*
LOOP		*	*	*		*		*	*	*	*		*	*			*	*	*	*	*	*			*	*	*	*		*	*	*
LOOPD		*	*	*		*		*	*	*	*		*	*			*	*	*	*	*	*			*	*	*	*		*	*	*
LOOPI		*	*	*		*		*	*	*	*		*	*			*	*	*	*	*	*			*	*	*	*		*	*	*

APPENDIX I Setting of status bits

	PSD	Z	C	S.K	O	IVO	D.FZ.U	F	F	B	I.SO.IV.T	B	C	T	B.AP.TT.F	A	T	T	A.DZ.R	I	S TX O	S.PT.RU.T	D	D	X.IS.IE.C	I	I	I	P
MTE1 :=		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
MTE1 =:		*	*	*:	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
MTE2 :=		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
MTE2 =:		*	*	*:	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
MUL2		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
MUL3		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
MUL4		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
MULAD		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
NEG		*	*	*:	*		*:*	*	*		:*				:*				:			:*	*	*	:				
NOOP				:			:				:*				:*				:			:*	*	*	:				
OR		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
OTE1 :=		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
OTE1 =:		*	*	*:	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
OTE2 :=		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
OTE2 =:		*	*	*:	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
P =:		*	*	*:	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
POLY		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
PSUM		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
PUTBF		*	*	*:	*		*:*	*	*	*	*:				:*		*		*:*	*		:*	*	*	*:		*		*
PUTBI		*	*	*:	*		*:*	*	*	*	*:				:*		*		*:*	*		:*	*	*	*:		*		*
R :=		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
R =:		*	*	*:	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
REM		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
RET				:			:				:*	*			:*				*:			*:*	*	*	:				
RETB				:			:				:*	*			:*				*:			:*	*	*	:				
RETBK				:*			:				:*	*			:*				*:			:*	*	*	:				
RETD				:			:				:*	*			:*				*:			:*	*	*	:				
RETK				:*			:				:*	*			:*				*:			*:*	*	*	:				
RETT				:			:				:*	*			:*				*:			:*	*	*	:				
RLADDR		*	*	*:	*		*:*	*	*		:*				:*	*			*:*	*		:*	*	*	*:		*		*
SCHPAR		*	*	*:	*		*:*	*	*	*	*:				:*	*			*:			:*	*	*	:		*		*
SCOMP		*	*	*:	*		*:*	*	*		:*				:*	*			*:			:*	*	*	:		*		*
SCOPA		*	*	*:*	*		*:*	*	*		:*				:*	*			*:			:*	*	*	:		*		*
SCOPT		*	*	*:*	*		*:*	*	*		:*				:*	*			*:			:*	*	*	:		*		*
SCOTR		*	*	*:*	*		*:*	*	*		:*				:*	*			*:			:*	*	*	:		*		*
SET1		*	*	*:*	*		*:*	*	*		:*				:*		*		*:*	*		:*	*	*	*:		*		*
SETBI		*	*	*:*	*		*:*	*	*	*	*:				:*		*		*:*	*		:*	*	*	*:		*		*
SETE				:			:				*:				:*	*			*:*	*		:*	*	*	*:		*		*
SETK				:*			:				:*				:*				:			:*	*	*	:				
SFILL		*	*	*:	*		*:*	*	*		:*				:*		*		*:			:*	*	*	:		*		*
SFILLN		*	*	*:*	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
SHA		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
SHL		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*
SHR		*	*	*:	*		*:*	*	*		:*				:*	*	*	*	*:*	*		:*	*	*	*:		*		*

APPENDIX I Setting of status bits

	PS D	Z	C	S.K	O	IV O	D.F Z.U	F O	B O	I.S O.I V.T	B T	C T	B.A P.T T.F	A T R	A T W	A.D Z.R	I X	ST O	S.P T.R U.T	D T	D E	X.I S.I E.C	I O	I S	I E	P V
SLOCA		*	*	*:*	*		*:*	*	*	:*			:*	*		*:*	*		:*	*	*	:	*		*
SMATCH		*	*	*:*	*		*:*	*	*	:*			:*	*		*:			:*	*	*	:	*		*
SMOVE		*	*	*:*	*		*:*	*	*	:*			:*	*	*	*:			:*	*	*	:	*		*
SMOVN		*	*	*:*	*		*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	*:	*		*
SMVTR		*	*	*:*	*		*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	:	*		*
SMVTU		*	*	*:*	*		*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	:	*		*
SMVUN		*	*	*:*	*		*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	:	*		*
SMVWH		*	*	*:*	*		*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	:	*		*
SOLO	*			:			:			:*			:*			:			:*	*	*	:			
SQRT		*	*	*:	*	*	*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
SSCAN		*	*	*:	*	*	*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
SSKIP		*	*	*:	*	*	*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
SSPAN		*	*	*:	*	*	*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
SSPAR		*	*	*:	*	*	*:*	*	*	*:*			:*	*		*:*	*		:*	*	*	*:	*		*
ST1 :=				:			:			:*			:*		*	*:*	*		:*	*	*	*:	*		*
ST1 =:				:			:			:*			:*		*	*:*	*		:*	*	*	*:	*		*
STZ		*	*	*:	*	*	*:*	*	*	:*			:*		*	*:*	*		:*	*	*	*:	*		*
SUB2		*	*	*:	*	*	*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	*:	*		*
SUB3		*	*	*:	*	*	*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	*:	*		*
SUBC		*	*	*:	*	*	*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
SWAP		*	*	*:	*	*	*:*	*	*	:*			:*	*	*	*:*	*		:*	*	*	*:	*		*
TEMM1 :=		*	*	*:	*		*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
TEMM1 =:		*	*	*:	*		*:*	*	*	:*			:*		*	*:*	*		:*	*	*	*:	*		*
TEMM2 :=		*	*	*:	*		*:*	*	*	:*			:*	*		*:*	*		:*	*	*	*:	*		*
TEMM2 =:		*	*	*:	*		*:*	*	*	:*			:*		*	*:*	*		:*	*	*	*:	*		*
TEST		*	*	*:	*		*:*	*	*	:*			:*	*		:			:*	*	*	*:	*		*
THA :=		*	*	*:	*		*:*	*	*	:*			:*	*		:			:*	*	*	*:	*		*
THA =:		*	*	*:	*		*:*	*	*	:*			:*		*	:			:*	*	*	*:	*		*
TOS :=		*	*	*:	*		*:*	*	*	:*			:*	*		:			:*	*	*	*:	*		*
TSET		*	*	*:	*		*:*	*	*	:*			:*	*	*	:			:*	*	*	*:	*		*
TUTTI	*			:			:			:*			:*			:			:*	*	*	:			*
UDIV		*	*	*:	*		*:*	*	*	:*			:*	*		:			:*	*	*	*:	*		*
UMUL		*	*	*:	*		*:*	*	*	:*			:*	*		:			:*	*	*	*:	*		*
WCONR		*	*	*:	*		*:*	*	*	:*			:*	*	*	:			:*	*	*	*:	*		*
WCONV		*	*	*:	*		*:*	*	*	:*			:*	*	*	:			:*	*	*	*:	*		*
XOR		*	*	*:	*		*:*	*	*	:*			:*	*	*	:			:*	*	*	*:	*		*

I N D E X

A to the I'th power 144
abbreviations . . . . . App.C
absolute addressing 77
absolute post indexed addressing 79
absolute program addressing 88,157,168
absolute value 108
access code 91,App.C
access protection 4,13,18,84
add 109
add three operands 117
add two operands 113
add with carry 124
address access 91
address code table 60,App.A,B
address codes 57,59,62
address domain 15
address mode survey 62
address register 3,226
address translation 25
address trap fetch (ATF) 40
address trap read (ATR) 41
address trap write (ATW) 41
address vector 21
address zero trap (AZ) 41
address, word 52
addressing modes 62
addressing traps 46
allocation strategy 7
alphabetical instruction table . . . . . App.F
ALT prefix 12,16,18,22,60,84
alternative addressing 21,84
alternative domain 4,12,16,22,61
An register 5-6,54,224
and 132
arithmetical instructions 94
array addressing 43,52,71,85
array arguments 71
AUX/LOG location 8,11,178
B register 5-6,7,65,67,69,71
BCD overflow (BO) 38
bias 50
bit data type 48
bit field 48,142,143
bit number within word 52
bit, implicit 50
block move and fill 83,203
branch trap (BT) 39
break point instruction 220
break point trap (BPT) 39
buddy allocation 10,178,180,214,215

buffering 28
byte 49
byte address 52
byte data type 48
byte number within word 52
cache 1,30,45
cache parity error (CPE) 45
cache partitioning 30
cache word 30
CAD register 5-6,17,21,22,181,223
calculate index 154
call subroutine absolute 168
call subroutine general 166
call trap (CT) 39
capability tables 17
carry (C) 38
CAS register 17,21,23,24,223
CED register 5-6,12,17,23,24,180,223
CES register 5-6,17,21,23,223
check parity in string 202
child domain 4
child trap enable register (CTE) 5-6,33,223
clear bit in own trap enable register . . . 219
clear flag 213
clear register 127
communication NORD-100/NORD-500 3,226
compare 102
compare two operands 103
concurrent procedures 7,10
conditional jump 40,158
constant operands 81
control instructions 156
control register 3,226
CPU 2
CTE register 5-6,33,223
current alternative domain 12,17
current alternative segment 12,17
current executing domain 12,17
current executing segment 12,17
data domain 4,15
data field, local 8,10,35
data part length specifier 59
data segment capability 12
data status bits 37,46
data type converson 82,204
data type conversion with rounding 206
data types in memory 52
data types in registers 54
decrement 131
DESC prefix 52,60,85
descriptor 52,61
descriptor addressing 43,52,85
descriptor range trap (DR) 41,52,85
descriptor, implicit 182
destination string 182
direct operands 88
disable process switch 216

disable process switch error (DE) 44
disable process switch timeout (DT) 44
displacement 60
displacement addressing 60,88
displacement, optimal size 60,92
divide 112
divide by zero trap (DZ) 38
divide three operands 120
divide two oerands 116
divide with remainder to register 122
DMA 3,226
Dn register 5-6,54,224
domain 4,12,15
domain call 4,13,16,22,42
domain communication 16,22
domain information 16,22
domain return 23,181
domain tree 4,16
domain, mother 4,24,32
double precision float 50
dynamic allocation 7,10,178,214
dynamic structures 7
En register 5-6,54,224
enable process switch 217
ENDH 10
enter module 8,23,172,180
enter stack subroutine 8,174,180,181
enter subroutine directly 8,173,180
enter trap handler 24,32,176
exponent 50
extension registers 5-6,54,224
fatal trap conditions 32,46
figures, table . . . . . App.D
flag (K) 43
float data type 50
floating point accumulator 5-6,54
floating point double precision 50
floating point overflow (FO) 38
floating point remainder 122
floating point rounding 51
floating point single precision 50
floating point underflow (FU) 38
Fn register 5-6,54,224
forward fetch 30
free buddy 215
freelist 10,178,215
general operands 57,62
general registers 5
get bit 138
get bit field 142
get buddy 214
halfword data type 49
heap management 10,214,215
heap variables 10,214,215
hexadecimal . . . . . Preface
hidden bit 50
high limit register (HL) 40

highlights 1
hit rate 30
I to the J'th power 145
I/O system 1
ignorable trap conditions 32,40
illegal index trap (IX) 41
illegal instruction code (IIC) 42
illegal operand specifier (IOS) 42
illegal operand value (IOV) 40
implicit bit 50
implicit descriptor 182
In register 5-6,54
increment 130
index register 5-6,54
index scaling error (ISE) 42
index, logical 61
index, physical 12,25,61,67,71,79
indirect segment 18,22
information tables 17,18
initial values 11
initialize stack 170
instruction code 89
instruction code cross reference table . . 259
instruction code table 256
instruction example 92
instruction formats 89
instruction lengths 90
instruction operands 56
instruction reference (IR) 43
instruction sequence error (ISE) 42
integer accumulators 5-6,54
integer data type 48
integer float register communication . . . 224
integer part 149
integer part with rounding 150
interprocess communication 4
invalid operation trap (IVO) 38
invert 106
invert with carry add 107
job scheduling 1
jump, conditional 158
jump, unconditional 156,157
K flag 43,85,179,180,182
link register 5
LL register 5-6,40
load 94
load address of multilevel link 210
load address to base register 209
load address to record register 208
load base register 95
load index 153
load record register 96
load special register 222
local addressing 65
local data field 8,10,35
local indirect addressing 69
local indirect post indexed addressing . . 71

local post indexed addressing 67
locked swap access 91,App.C
log size 10
logical address 21
logical instructions 94
logical page number 25,29
loop general 164
loop with decrement 162
loop with increment 160
low limit register (LL) 5-6,40
mailbox 3,226
mantissa 50
mass storage 25
memeory, physical 1,25
memory address out of range (MOR) 45
memory management 12
memory management system error (MME) . . . 45
memory size 25,29
memory system error (MSE) 45
microcode 226
microprogram 226
miscellaneous instructions 203
monitor call 4,22,32,43
mother trap enable register (MTE) 5-6,33
move 100
multioperand instructions 53
multiply 111
multiply and add 151
multiply three operands 119
multiply two oerands 115
multiply with overflow to register 121
N stack location 8,11,17,171-178
negate 105
newB 170-178
newB.ARGn 170-178
no operation 211
non-ignorable trap conditions 32,47
non-reentrant routines 7,35,173
NORD-100 1,32,226
numeric formats 48
octal . . . . . Preface
oldB 171-178
operand 56
operand reference status bits 40
operand specifier address code 60
operand specifier data part 59
operand specifier format 58
operand specifier prefix 57,84,85
operand specifier structure 58
operands, constant 81
operands, direct 56,88
operands, general 57
operands, register 83
operating system 1,32
or 132
overflow trap (O) 38
own trap enable register (OTE) 5-6,24,33,35,222,223

I N D E X

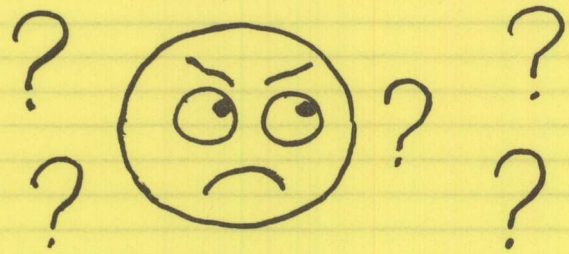
P register 5-6,223
P relative addressing 88,156,158-164
page fault (PGF) 42
paging 13,25,42
parameter access 22,84
parity 30,45,201,202
part done (PD) 43
physical implementation 25
physical page number 25-27
physical segment table pointer 5-6,25
physigal segment table 25-27
pipelining 2,30
pointer register 182
polyminal 147
pool 10
post indexing 52,59,61,67,71,79,85
power failure (PWF) 45
pre indexed addressing 75
prefetch 1,30
prefix combinations 61
prefix, alternative 84
prefix, descriptor 85
PREVB stack location 8,23,170-178,181
private memory 2
privileged instruction allowed (PIA) . . . 43
process 4,12,16
process description 13,16
process number 29
process registers 17
process segment 12,17-20
process switch disable (PSD) 42,44
processor fault (PRF) 45
program counter P 5-6,223
program domain 4,15
program memory 15
program segment capability 19
programmed trap (PRT) 43
protect violation (PV) 18,42
PS register 5-6,17,27
PSTP register 5-6,12,25
put bit 139
put bit field 143
read only memory 18,48,81
record addressing 73
record register 5-6,73,96,99
recursive routines 8
reentrant routines 7,35,173
register addressing 83
register block 5-6
register operands 56,83,89-90
registers, double precision 5-6,54-55
registers, extension 5-6,54-55,222,223
registers, floating point 5-6,54-55
registers, integer 5-6,54-55
registers, special 5-6,222,223
related manuals . . . . . Preface
RETA stack location 8,23,170-178,181

return from subroutine 179
Rn register 54
rounding, floating point 51,150,206
routine calls 9,23,42,166,168
routine number 4,21
routine vector 4,21,23
scaling factor 61,85
secondary storage 12
segment capability 19
segment number 7,15,21,166,168
segment relative address 21
segment size 4,25
segment, current 12,17,223
segment, indirect 18-19
set bit in own trap enable register 218
set flag 212
set parity in string 201
set to one 129
shared segment 4,7,13,16,18
shift arithmetical 136
shift logical 135
shift rotational 137
short address codes 52,59-60
short displacement part 52,59-60
sigle precision floating point 50
sign (S) 37
sign extension 55
signalling status bits 43
signed integer 48
single instruction trap (SIT) 39
source string 182
SP stack location 8,172,178
special instructions 216
special purpose registers 5-6,222,223
square root 146
stack displacement 9
stack initialization 8,23,170
stack management 7
stack overflow (STO) 42
stack pointer 8,172,178
stack underflow (STU) 42
STAH 10
static allocation 7
static link 210
status bits modification 36
status bits survey 47
status register (ST) 5-6,47,222,223
store 97
store local base register 98
store record register 99
store special registers 223
store zero 128
string compare 192
string compare translated 193
string compare translated with pad 195
string fill 190
string fill n elements 191

string instructions 182
string locate elements 197
string match 200
string move 184
string move n elements 189
string move translated 187
string move translated until 188
string move until 186
string move while 185
string scan 198
string skip elements 196
string span 199
subroutine arguments 69,81,83
subroutine entry points 171
subroutine return 179
subtract 110
subtract three operands 118
subtract two operands 114
subtract with carry 126
sum of products 152
swap 101
swapping 12,25
symbols and abbreviations . . . . . App.C
synchronization status bits 43
system configuration 1
system diagnosis 1,226
system error status bits 45
table of figures . . . . . App.D
tag register 226
termination conditions 182
test against zero 104
test and set 221
top of stack register (TOS) 5-6,8-11,170,172,177,223
TOS register 8-11,170,172,177
tracing status bits 39
translation speedup buffer (TSB) 28
translation table 182
trap conditions 32
trap enable modification mask (TEMM) . . . 5-6,33,223
trap handler address register (THA) 5-6,35,223
trap handler routine data field 35,176
trap handler routines 32
trap handling 24,32,176
trap information 29,176
trap priority 36,39
trap propagation 24,32
traps 24,32,176
two's coplement 49
type conflicts 53,81
unconditional absolute jump 157
unconditional relative jump 156
unsigned divide 124
unsigned multiply with overflow to register 123
word data type 49
write permitted 18
write through 31
xor 134

zero (Z) 37
zero fill 22-23,181

* * * * * * * * * * SEND US YOUR COMMENTS!!! * * * * * * * * * *

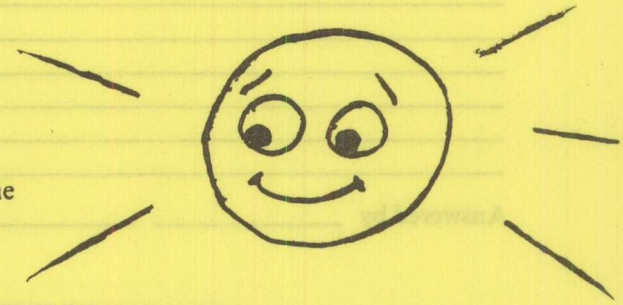


Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card - and an answer to your comments.

Please let us know if you

* find errors
* cannot understand information
* cannot find information
* find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!!



* * * * * * * * * HELP YOURSELF BY HELPING US!! * * * * * * * * *

Manual name: NORD-500 Reference Manual Manual number: ND-05.009.01

What problems do you have? (use extra pages if needed) _____

Do you have suggestions for improving this manual? _____

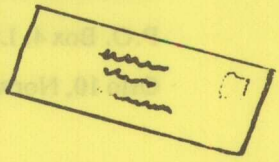
Your name: _____ _____ _____ Date: _____

Company: _____ _____ _____ Position: _____

Address: _____

What are you using this manual for? _____

Send to: Norsk Data A.S.
Documentation Department
P.O. Box 4, Lindeberg Gård
Oslo 10, Norway



Norsk Data's answer will be found on reverse side

Answer from Norsk Data _____

Answered by _____ _____ _____ _____ Date _____

I I

I I

I I

Norsk Data A.S.

Documentation Department

P.O. Box 4, Lindeberg Gård

Oslo 10, Norway

– we make bits for the future

NORSK DATA A.S BOX 4 LINDEBERG GARD OSLO 10 NORWAY PHONE: 30 90 30 TELEX: 18661