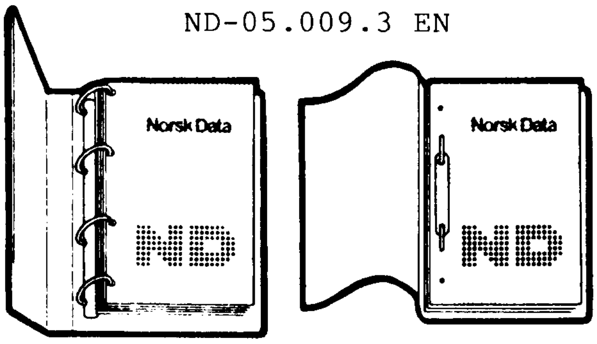
ND-500 Reference Manual

ND-05.009.3 EN

PRINTING RECORD

PRINTING	NOTES
10/80	Version 1
07/82	Version 2
06/87	Version 3

ND-500 Reference Manual
ND-05.009.3 EN



UPDATING

Manuals can be updated in two ways, new versions and revisions. New versions consist of a completely new manual which replaces the old one, and 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 Customer Support Information and can be ordered from the address below.

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

RING BINDER OR PLASTIC COVER

The manual can be placed in a ring binder for greater protection and convenience of use. Ring binders may be ordered in two sizes: B5 and A-4.

The manual may also be placed in a plastic cover. This cover is more suitable for manuals in A4 size of less than 100 pages than for larger manuals.
Please send your order, as well as all types of inquiries and requests for documentation to the local ND office, or (in Norway) to:

Norsk Data A.S
Graphic Centre
P.O.Box 25 BOGERUD
N-0621 OSLO 6 - Norway

I would like to order

.... Ring binders, B5, at NOK 35.- per binder

.... Ring binders, A4, at NOK 45.- per binder

.... Plastic Covers, A4, at NOK 10.- per cover

Name: ....

Company: ....

Address: ....

Preface:

PREFACE

THE PRODUCT

This manual describes the instruction set, the trap-handling system and the memory management system of the central processing unit of the ND-500 series computer systems and the ND-5000 series computer systems.

The ND-5000 CPU has a completely new and unique physical implementation, but is based on the ND-500 systems architecture. The ND-5000 uses the same instructions as the ND-500 .

THE READER

The ND-500 CPU reference manual is intended for anybody using the ND-500 assembler and for system programmers needing to know the exact format of the generated code.

Programmers making advanced use of the memory management system for segmenting, or writing their own trap-handling routines will find detailed information in this manual.

PREREQUISITE KNOWLEDGE

No previous knowledge of the ND-500 or the ND-5000 is required, but assembly programming experience is desirable. Understanding the memory management system, making programs that handle communication between the I/O processor and the ND-500 or the ND-5000 and the inner kernel of the operating system requires a more detailed description of both ND-500 , or ND-5000 , and ND-100 hardware. This can be found in

ND-5000 Hardware Description	- ND-05.020
ND-500/2 Hardware Description	- ND-05.015
ND-100 Functional description	- ND-06.026

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

ND-500 Assembler Reference manual	- ND-60.113
ND-500 Loader Monitor	- 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 hardware details relevant to software. Each chapter is independent and can be understood without reading previous chapters.

This manual is valid for both the ND-500 and the ND-5000 computer systems. When the manual uses the name ND-5000 this is also valid fore the ND-500 .

The chapters are organized as follows:

Part II is organized in a logical way. You find related instructions when leafing through the neighbouring pages to a specific lookup.

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

NEW INSTRUCTIONS

A number of new instructions are introduced with the ND-5000 . These instructions also run on computer systems with the ND-500/1 and the ND- 500/2 CPUs. The instructions are labelled: **('87 extension)**.

CPU - I/O PROCESSOR

The term 'CPU' is used for the ND-500/ND-5000 processor throughout this manual. Whenever the I/O processor is mentioned, this means the ND-100/ND-110 processor.

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' and 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 OFFH). In this manual hexadecimal numbers are always preceded 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.

The term WORD always refers to 32-bit words. 16-bit data items (ND-100 words) are referred to as HALFWORDS. The term BYTE refers to 8-bit bytes.

In the figures, address values increase downwards.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 CPU Architecture and CPU Implementation

By introducing the ND-5000 systems, Norsk Data also introduces the ND-5000 CPU. This is the third generation of implementations of the ND-500 CPU architecture.

The CPU software architecture is still named ND-500, while the new systems, with the ND-5000 CPU implementation, are named the ND-5000 series computer systems. The concepts software architecture and implementation are outlined in table 1.

CPU-		Name	Systems
software architecture →	→ instruction set → register set → addressing modes → trap system	ND-500	All
physical implementation		ND-500/1	ND-520/540/560
		ND-500/2	ND-510/530/550/560/570/580
		ND-5000	ND-5X00

Table 1. CPU Architecture and CPU Implementation

The ND-5000 CPU runs the same instruction set, uses the same register set and the same addressing modes as the ND-500/1 and the ND-500/2 CPUs.

1.2 System configuration

The ND-5000 central processing unit is part of the ND-5000 computer system. This system is a combination of an I/O processor, an ND-5000 CPU and a shared memory, see figure 1. Until now the I/O processor has been an ND-100, but when the DOMINO I/O system is introduced, other types of I/O processors will be possible.

THE I/O PROCESSOR:

- Supervises the CPU

- Runs the I/O system, file system, operating system and job scheduling

- Runs local I/O-processor jobs

THE ND-500 type CPU:

- 32-bit logical address
- 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 efficient execution 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 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 multiplication and division
- Optional BCD hardware for operations on packed binary-coded decimal numbers.

MEMORY:

- Multi Function Bus main memory with direct access for the ND-5000 CPU, the I/O processor CPU and DMA transfer devices
- Physical main memory up to 32 Mbytes
- Virtual memory management system
- Memory fully or partially shared between the I/O processor and ND-500 type CPU.

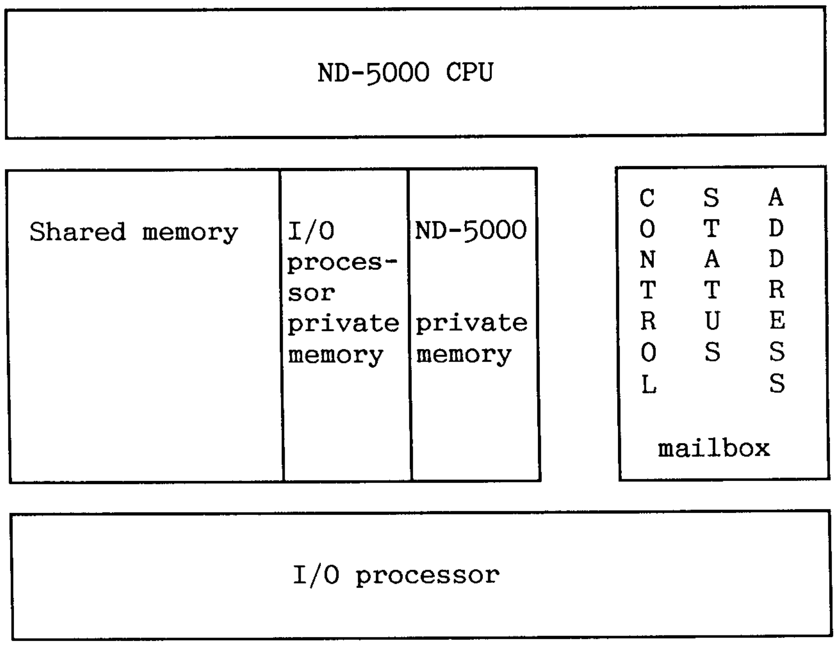


Figure 1. The ND-5000 computer system

1.3 Communication between the I/O Processor and the CPUs

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

The communication between the I/O processor and the CPU is set up as a mailbox and DMA transfer system. The mailbox contains 3 registers:

- Control register: For the I/O processor to give the CPU a command
- Status register: For the CPU to give the I/O processor status
- Address register: A pointer to where in the I/O-processor memory a chain of message buffers will be found. Message buffers may contain commands or data from the I/O processor to the CPU or may be used by the CPU for storing extended status information

Some examples of commands to the CPU are context switch, reset, wait or data transfer.

The status information returned to the I/O processor reports that a job is finished, the reason for the CPU termination and the type of possible CPU malfunctions.

The CPU microprogram initiates and controls the DMA access channel to the I/O-processor memory. The communication channel is also used extensively for diagnostic and test program information. The I/O-processor is used as a diagnostic vehicle for the CPU.

1.4 Domains, segments and processes

The memory in an ND-500 type system is logically structured into DOMAINS. A domain has one 32-bit address area (4 gigabytes) for executable code (the program domain) and another one for data (the data domain).

Each domain is divided into SEGMENTS, with up to 32 per domain. A segment can be up to 128 Mbytes, which is equivalent to 27 address bits. The smallest unit for access protection (write and parameter access protection) is a segment. An instruction segment may access any data segment in the domain.

Two (or more) domains may have segments in common in order to share code or data.

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, in effect, run concurrently. The processes, however, are "time-sliced".

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, and 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 domain, routines are called directly by address. Routines in other domains are called through their routine number, not by address.

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

2 THE REGISTER BLOCK

The ND-500 type 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 ND-5000 also has several special purpose registers:

ST	Status register
OTE	Own trap enable register
CTE	Child trap enable register
MTE	Mother trap enable register
TEMM	Trap enable modification mask

Table 2. 64-bit Special Purpose Registers

TOS	Top of stack register
LL	Low limit trap register
HL	High limit trap register
THA	Trap handler address register

Table 3. 32-bit Special Purpose Registers

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:

CED	Current executing domain register
CAD	Current alternative domain register
PS	Process segment register
PSTP	Physical segment table pointer

Table 4. Memory Management Utilized Registers

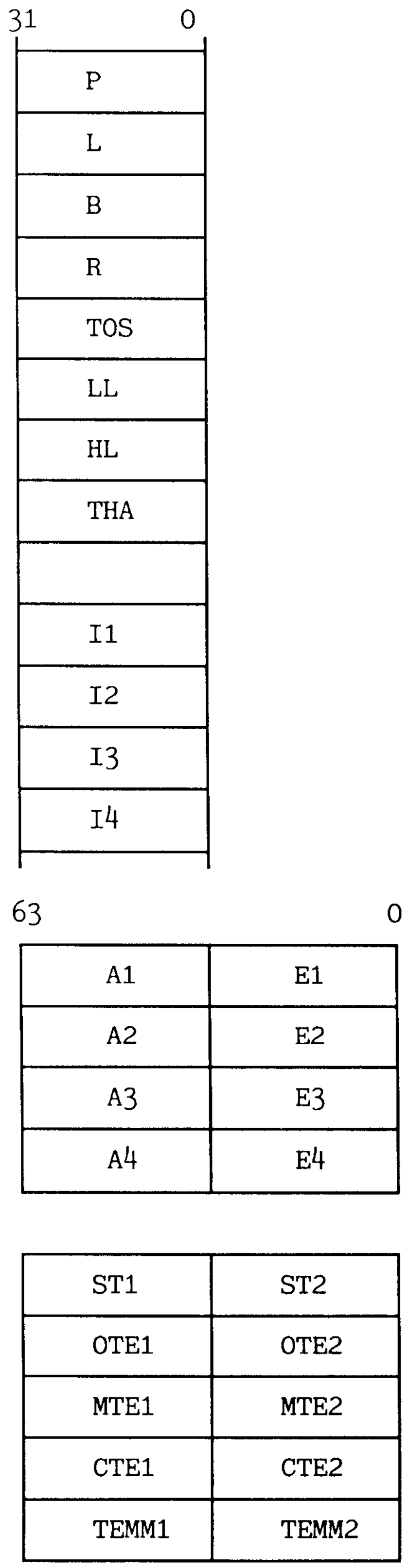
Each process in the system has its own copy of the CED, CAD and PS registers. PSTP is one global register for the whole system.

The context block is made up from these registers except from PSTP. In addition, it contains scratch registers named 'mic'. These are registers accessable from microprogram only, for use in macroinstructions that may be interupted while operating on more data than are handled by the general registers.

The registers are numbered according to the table below. Note that 64-bit registers are given consecutive numbers.

arg1	:	Trapping P	arg17	:	E4	arg33	:	CTE1
2	:	P	18	:	ST1	34	:	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	41-50	:	copy of program memory
10	:	A1	26	:	CAD			
11	:	A2	27	:	mic			
12	:	A3	28	:	mic			
13	:	A4	29	:	mic			
14	:	E1	30	:	mic			
15	:	E2	31	:	OTE1			
16	:	E3	32	:	OTE2			

Table 5. Register Numbers



Program counter

Link (subroutine return address)

local variable Base

Record base

Top Of Stack register

Low Limit trap register

High Limit trap register

Trap Handler Address register

Integer accumulators
or Index registers

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

Floating point accumulators
and Extension registers
A=E= 32 bits, D=A+E= 64 bits

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

STatus register

Own Trap Enable register

Mother Trap Enable register

Child Trap Enable register

Trap Enable Modification Mask

Figure 2. The Register Block

3 STATIC DATA, STACK AND HEAP

When a subroutine is called, space is required to store return information and local variables. This space may be allocated

- in a fixed location in memory, referenced relative to the B register or by absolute address (static allocation)
- on a stack growing from low to high memory, referenced relative to the B register
- in a block released from a freelist. The block may be anywhere in otherwise unused memory, referenced relative to the B register.

Static or dynamic allocation of the local data area of a routine 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.

The initialization of the header of the local data area is in most respects equivalent for static, stack and heap allocation. Usually, 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. Statically allocated data are not released during program execution for other use, and local variables in routines keep their values from one call to the next.

Routines with static data areas are entered through an ENTF or ENTFN instruction. Such routines are by definition non-reentrant and cannot be called recursively , but in other respects they behave like other routines. The fixed local data area is initialized as shown in figure 3. 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 can declare 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 beyond the stack's maximum extent. TOS serves to prevent the stack from growing too large, and as a pointer to the variables describing the heap. The first free location beyond the current extent of the stack is pointed to by the B.SP location.

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

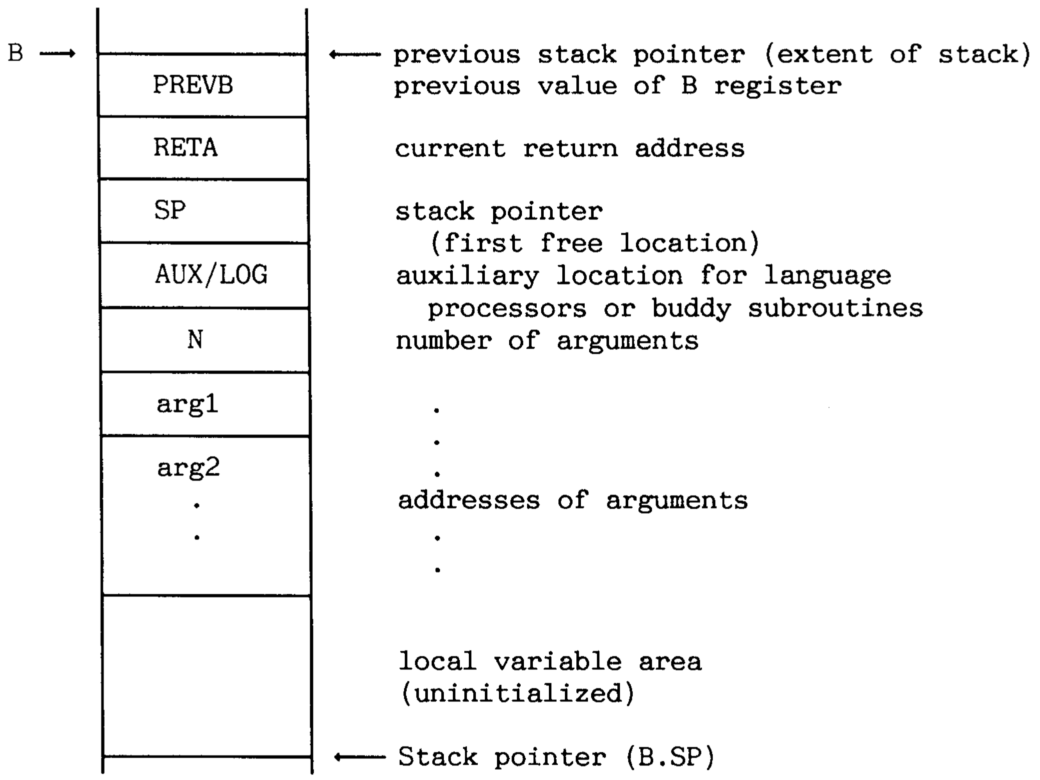

B
PREVB
RETA
SP
AUX/LOG
N
arg1
arg2
previous stack pointer (extent of stack)
previous value of B register
current return address
stack pointer (first free location)
auxiliary location for language processors or buddy subroutines
number of arguments
addresses of arguments
local variable area (uninitialized)
Stack pointer (B.SP)

Figure 3. Local Data Area Layout

If the number of arguments supplied exceeds the maximum allowed by the ENTSN entry point instruction, only the maximum allowed number of argument addresses will be put on the stack and the N location will contain the value of the "maximum number of arguments" operand. (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 or Stack Underflow trap.

The ENTM instruction initializes a new stack starting from a specified address, giving the TOS register a new value. If the module called is within the current domain, the old TOS value is saved on the current top of the old stack, pointed to by B.SP. Initialization of the new stack is the same as for a routine entry; the base address of the previous stack block is saved in PREVB. If the module is in another domain, TOS, PREVB and RETA are stored in the domain information table and restored on return.

The ENTM is typically used for initializing a stack for the routines on a segment, being called from other segments in the same domain or from other domains. Executing the same ENTM instruction twice will overwrite the old initial values, possibly destroying the 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) in the current domain, the TOS register is not updated; this must be done explicitly. After a domain call, TOS is restored from the domain information table.

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

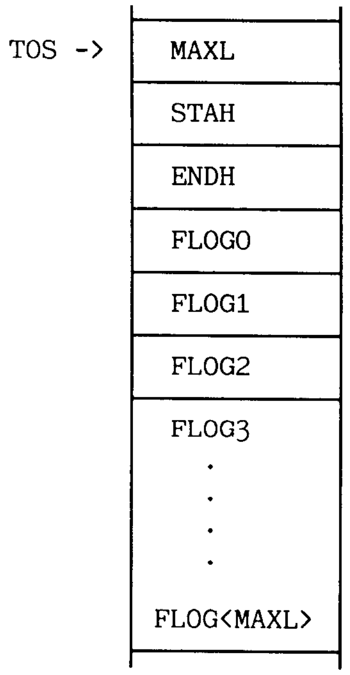
3.3 Heap allocation

When running several routines "concurrently" (see section 1.4), 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 a set of heap variables pointed to by the TOS register. The heap variables are the MAXL, STAH and ENDH locations and an array of pointers to linked lists of free elements, each block size has its own free list. The first word of an element contains the address of the next element in the list, zero indicating the end of the list. 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.

MAXL, the first location beyond the stack, is pointed to by the TOS register and contains the maximum size of elements to be allocated. The next two locations, STAH and ENDH, are reserved for the lower and upper address limits of the pool respectively. Beyond these two locations is the array of pointers, FLOG0 to FLOG<MAXL>.



Max log size of elements allowed

Start of heap

End of heap

Head pointers for freelists of elements of the different log sizes. The freelist pointers have the value 0 if no element of the log size is available.

Figure 4. 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.

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, which specifies 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 is left in the appropriate freelist. The block may have to be split several times before an element of the requested size can be given 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 acquired 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 trap condition.

The stack overflow trap is used to signal that all lists containing blocks of wanted size or larger are empty.

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.

If ENTB is used to allocate space for co-routines, care should be exercised if the called routines make further calls to stack routines. When co-routines use a common stack and a second co-routine is activated before the return, the stack areas will overlap because B.SP is the same in both routines. No problems will occur if all routines in the system are entered through ENTB or if the stack routine is certain to terminate before another co-routine is activated. (Standard library routines may be used freely; they will not cause activation of other co-routines.)

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

4 MEMORY MANAGEMENT SYSTEM

4.1 Introduction

A process is a sequential computation requiring no parallel execution. A process may refer to up to 256 domains. Each domain is a full 32-bit address area for program instructions and another one for data. A process may easily access two such data domains, the so-called 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 with 27 address bits each. A 27-bit logical segment address is translated by the memory management system so that it addresses a location in a so-called physical segment. Physical segments contain the data and programs for the CPU. A physical segment is divided into blocks of 2k bytes called pages, and may have any size from 2**11 to 2**27 bytes in units of 2k bytes (1 page). Pages can be moved (swapped) between main memory and secondary storage as the need arises.

All physical segments in the system are described in the Physical Segment Table (PST). The PST always resides in the 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 may use up to 256*32 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 CPU. 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 the I/O processor. Each domain used by a process has one entry in the PS.

One part of the process segment is called the domain information table. A domain information table contains 32 pointers for data (the data capability table) and 32 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. 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. The domain information table also contains the necessary 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 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.

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. By doing this, 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 a process identification. The 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 identification.

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 inherent property of the architecture prohibits other implementation strategies.

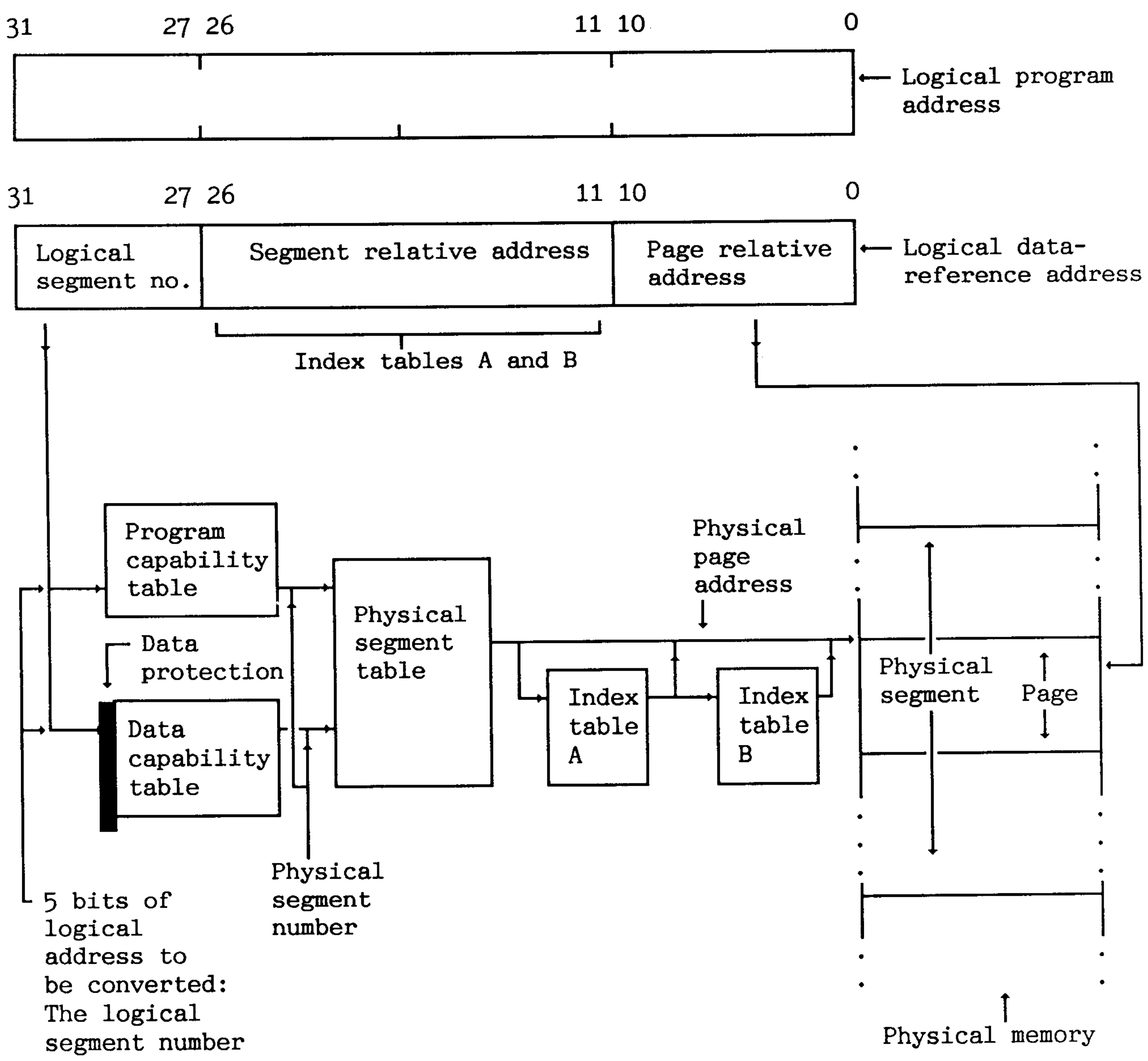


Figure 5. Logical addressing scheme

4.2 Memory management architecture

4.2.1 Address domain

An address has 32 bits, i.e. is in the range 0 to (2**32)-1. Instruction fetches and data references refer to different areas of the memory. If the memory request is an instruction fetch, the address value range is called a program domain. If the memory request is a data reference, the address value range is called a data domain.

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

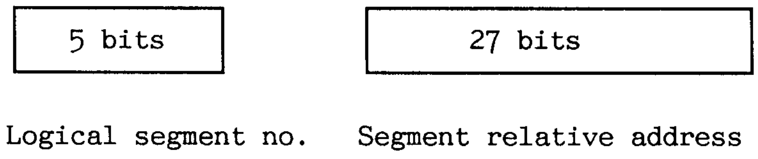


Figure 6. Logical Address

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

The division of domains into segments makes different protection and cache setup possible for each segment (see figure 9).

The scheme does not, however, forbid accesses to data structures crossing segment borders as long as the access capabilities are the same for both segments.

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 to execute the 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.

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

The domains of a process are hiearchically structured in a tree. The closest domain above a domain D is called the mother domain of D; D is called the child. In figure 7, D and E are both child domains of B; B is their mother. A is the mother of B and C. The hierarchical structure is reflected in the process description.

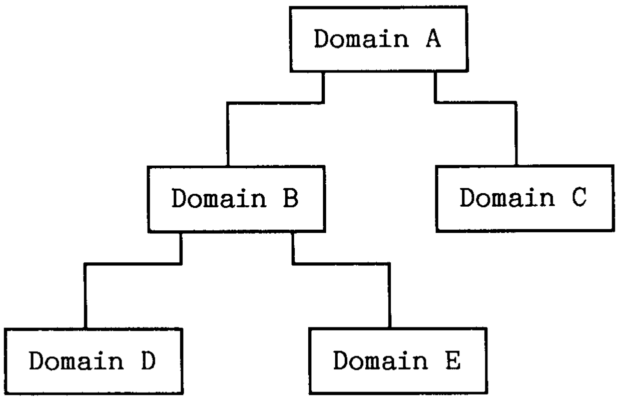


Figure 7. Hierarchy of Domains

Transfer of control between domains may take place by routine calls (domain calls) or enabled traps. Routine calls may transfer control to any of the domains of the process. The child-to-mother links are followed when a trap occurs in a child domain and no trap handler is defined locally in the child domain.

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 microcode routines and by supervising mechanisms, which may be other processes, other domains or the I/O processor. 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
PS	Process Segment

Figure 8. 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 three registers CED, CAD and PS are part of the process description of the running process, i.e. the registers' contents are saved and loaded when the process is changed.

The Current Executing Domain register holds the current domain number of the currently executing process. When a domain call is performed, or when a trap condition is not <u>own</u> but <u>mother</u> enabled, the domain number of the calling domain is stored in the Current Alternative Domain register. CAD is used with the alternative addressing mode.

4.2.3.2 Capability tables

Each domain has two capability tables, one for instructions and one for data. Each table has 32 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.

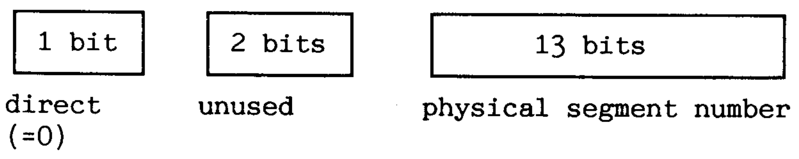
In a program capability, bit 15 indicates whether the segment is in the current domain or not. If the bit is zero, the segment is in the current domain. A segment not in the current domain, called an indirect segment, has bit 14 set if the physical segment resides in another machine, otherwise it is 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 other's updating of a data item.

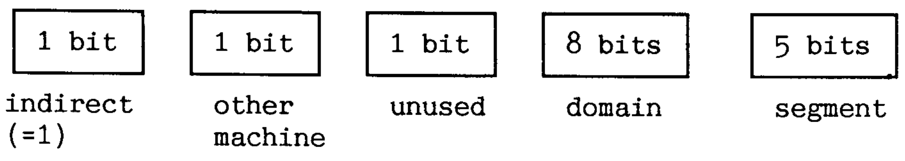
Direct program segments and data segments contain the physical segment number in the lower 13 bits.

Program segment capability:

a) Direct segment



b) Indirect segment



Data segment capability:

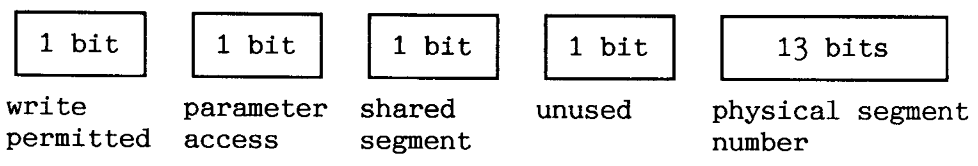


Figure 9. Capability Layout

4.2.3.3 Domain information

When performing domain calls and trap handling, some extra table space is needed for each domain. The first part of a domain information is made up of 2 capability tables. The next part has two save areas; one used when performing domain calls, and one used during trap handling. The last part holds the domain characteristics.

All the above constitute one domain information table. This table is followed by an unused area to a total size of 256 bytes.

The "category" column below uses the following abbreviations:

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

The domain information table layout is shown on the next page.

		Relative address	No.of bytes	Cate-gory
a. **Program capability table**		0B	64	0
b. **Data capability table**		100B	64	0
c. **Domain call information**				
Calling domain		200B	1	M
Alternative of calling domain		201B	1	M
P of calling domain	P	203B	4	M
B of calling domain	B	207B	4	M
d. **Trap handling information**				
Trapped domain		213B	1	T
Alternative of trapped domain		214B	1	T
Status register save area	ST1	216B	4	T
	ST2	222B	4	T
Inside trap handler flag		273B	1	T
e. **Domain characteristics**				
Own trap enable	OTE1	226B	4	0/M
	OTE2	232B	4	0/M
Child trap enable	CTE1	236B	4	0
	CTE2	242B	4	0
Mother trap enable	MTE1	246B	4	0
	MTE2	252B	4	0
Trap enable modification mask	TEMM1	256B	4	0
	TEMM2	262B	4	0
Trap handler address	THA	266B	4	0/M
Mother domain		272B	1	0
Top of stack register	TOS	274B	4	0/M
Low limit register	LL	300B	4	0/M
High limit register	HL	304B	4	0/M
Domain status (PiA = bit 0)		310B	1	0

Table 6. Domain Information Table

4.2.4 Logical addressing

A logical address consists of the logical 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.

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

4.2.5 Domain communication

Within the domain hierarchy of the 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 final 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 calls and monitor calls

From one domain, a routine on any other domain of the process may be called through the CALL and CALLG instructions. This is only possible if an indirect capability to that domain has been set up. This is indicated by bit 15 being set in the capability of the segment. An indirect capability is set up through monitor calls. An indirect segment resides in another domain than the current one. A call to a routine on such a segment implies a change of domain, and is referred to as a domain call.

Domain calls to supervising domain routines performing specific functions are called monitor calls. Service requests to the operating system are implemented as monitor calls.

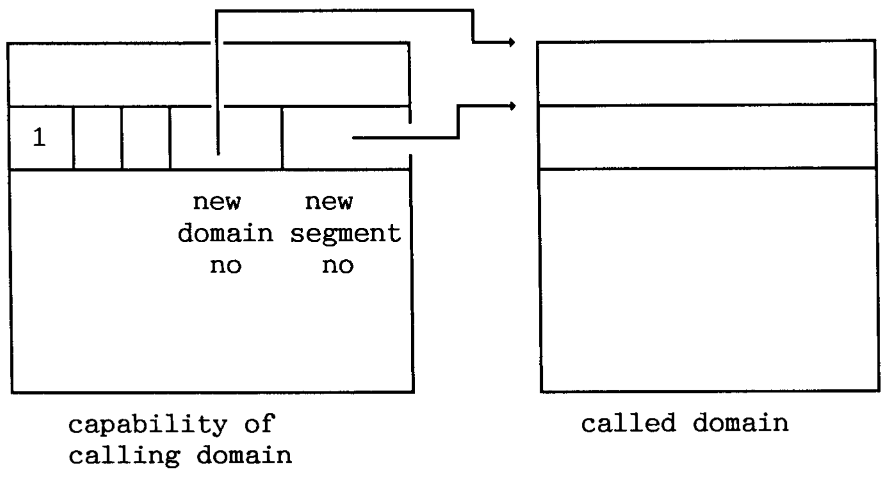


Figure 10. Indirect segment

The new domain and segment number are taken from the capability of the calling segment. The P and B registers, domain number and alternative domain number of the calling domain 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 ENTM instructions.) The return address and old base register field of the local data field of the new routine are filled with zeroes.

The new domain number is loaded into the Current Executing Domain register and the number of the calling domain is loaded into the Current Alternative Domain register.

The lower 27 bits of the routine address are not interpreted as within the segment an address. Instead they are taken as an index in the start address vector at segment address zero on the new segment. The first word is the length of the vector, which is the number of routines on the segment. If the index is less than this word, the indexed element in the vector contains the address of the routine entry point. Otherwise the call is illegal and causes an instruction sequence error trap condition. The routines on the segment are numbered starting from zero.

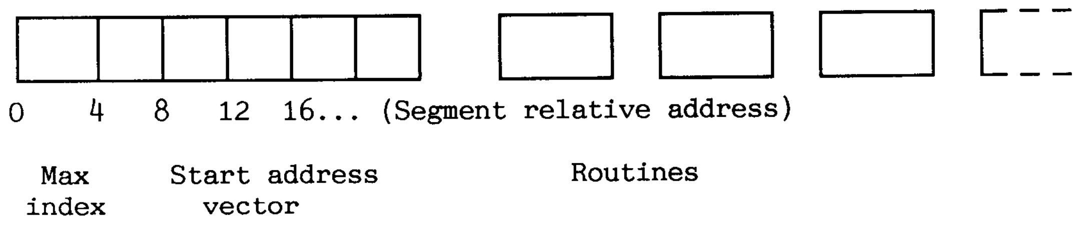


Figure 11. Program segment layout

On jumps 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 instruction. When an ENTM is entered from another domain, B.PREVB and B.RETA will be cleared. Other entry point types will not properly initialize the stack.

When the new domain is entered, TOS is not saved on top of the old stack. The TOS, THA, LL and HL registers will be saved in the old domain information table and the new contents of these registers are loaded from the new domain information table.

Control reverts to the calling domain when either the return address, the old base register, or both is zero when a return instruction is executed. On return from a domain call, the registers CED, CAD, P and B are loaded from the old domain information table. The registers TOS, THA, LL, HL and TE are loaded from the new domain information table.

Note that return information is not stacked in the domain information table. Calling the same domain twice without return in between, will cause an instruction sequence error trap condition. The memory management system will zeroize the return address and B register value in the domain information table at a domain call return to indicate that a call to the domain may be done. If it is non-zero a domain call is in progress.

A return instruction with 0 in PREVB or RETA will only change domains 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 in the current domain. This may cause 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 and CAD are saved in 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 and fatal bits and bits 0 to 8 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

Physical main memory size may be up to 2**41 bytes, divided into 2048-byte pages. The page size of 2048=2**11 implies 2**30 pages, or a 30-bit page number.

The memory management system has a bit map with two bits per physical page, set if the page is or has been written to. 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**30) bits, and it is accessible to microcode and privileged processes 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 4-byte entry for each physical segment, giving the page number of a data page or an index page.

If the Physical Segment Table entry is 0, this means that no mapping exists for the logical address that needs translation. This is a page fault trap condition.

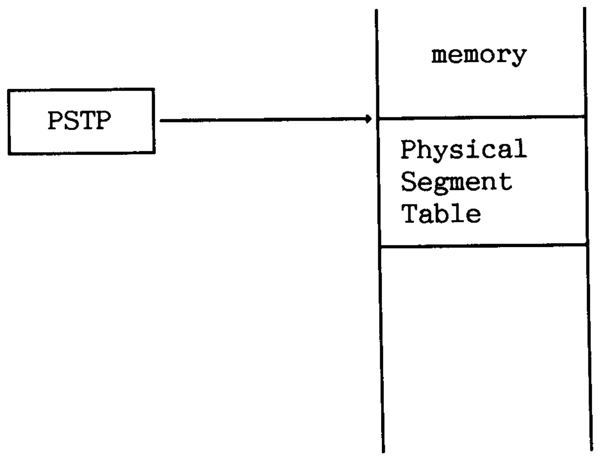


Figure 12. Physical segment table

The access method, directly by physical page number, or indexed once or twice, depends on the size of the segment. Bits 30-31 of an element in the physical segment table hold information about access method.

Direct access restricts the segment size to 2 k bytes. Single indexing allows 512 pages, or 1 megabytes maximum segment size. Larger segments use double indexing, the maximum size of which (2**31 bytes) exceeds the maximum segment size.

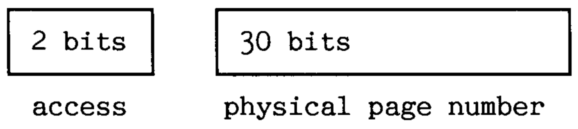


Figure 13. Physical segment table entry

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

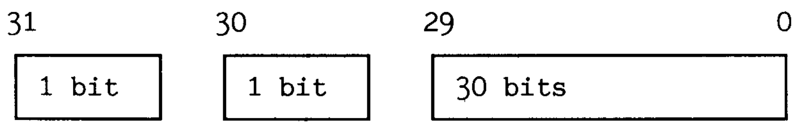


Figure 14. Index page table entry

An index page entry has a layout similar to a PST entry. Bit 30 is unused. Bit 31 in an index page table entry is unused except on the last indexing level, that is, when the page number part of the entry specifies a data page, when bit 31 is used for data page write protection. The physical address is calculated from the physical segment number and segment relative address as shown in figure 15.

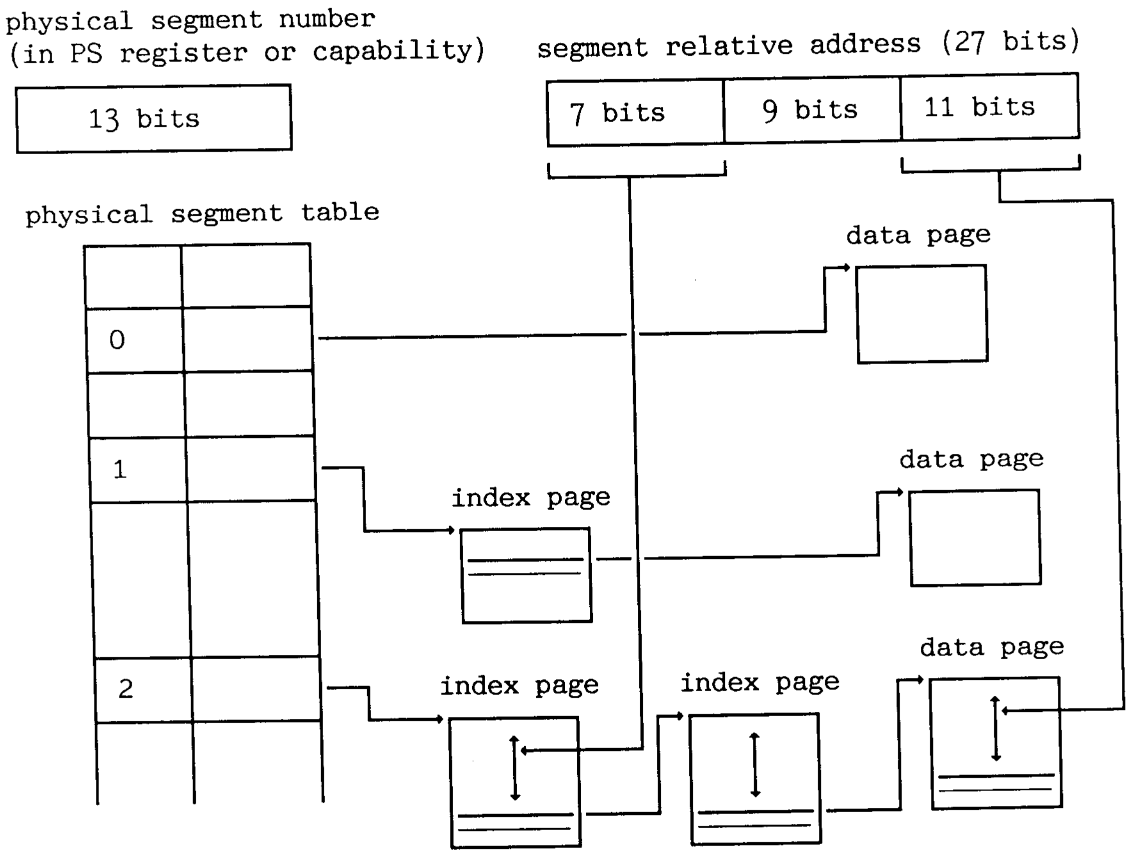


Figure 15. Physical memory

As for pointers in PST, pointers in index tables will have zero value to indicate a page fault.

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 current logical segment number selects an entry in the capability table. This table entry contains the physical segment number of the referenced segment.

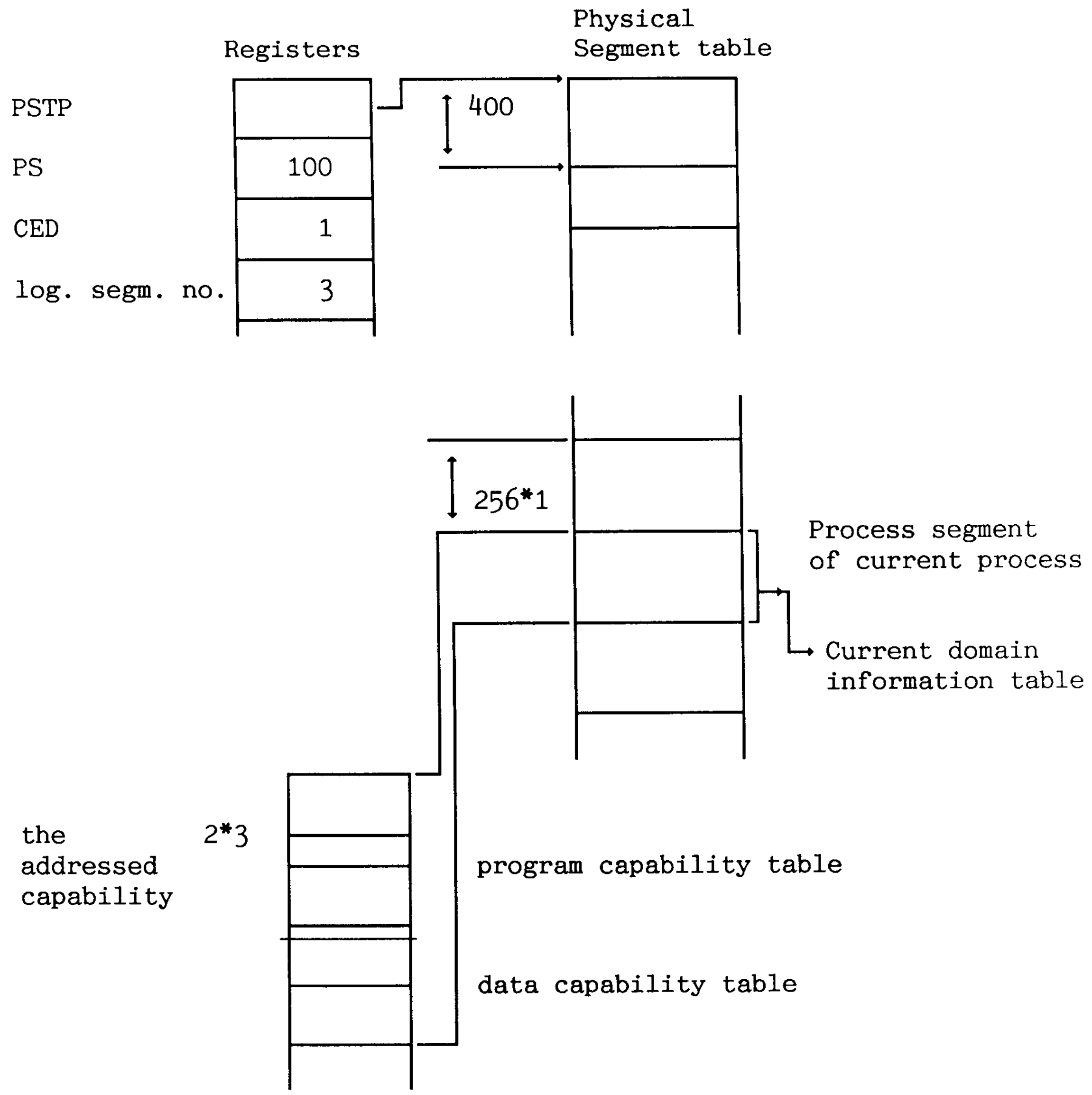


Figure 16. Addressing a program capability

4.4 Buffering

Translation from logical to physical address is complicated and requires several memory accesses. To reduce the number of accesses, the most recently used logical page number (the upper 21 address bits), domain number and a part of the process number are 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 (TSB). The domain and process numbers are also stored. Therefore it is not necessary to clear the buffer when changing domain or process.

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 necessary 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 necessary information has been found.

Further details on the translation speedup buffer are found in the manual ND-5000 Hardware Description (ND-05.020).

5 CACHE MEMORY SYSTEM

The ND-500 CPU and the ND-5000 CPU have different cache memory implementation. Consult the manuals ND-500/2 Hardware Description (ND-05.015) and ND-5000 Hardware Description (ND-05.020) for details.

The speed of the CPU is considerably 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 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 a cache. For economic reasons the cache is comparatively small, and sophisticated circuitry is employed to determine which data elements should be allotted space in the cache.

When data residing in the cache is updated without updating the corresponding memory location, the cache item is marked 'dirty'. Thus, such items should be dumped when the cache is cleared in order to maintain data consitency.

The effective memory access time as seen from the CPU is a function 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").

To prevent instructions and data located at the same cache address from constantly displacing each other when a loop is executed, instructions and data have separate cache systems.

6 THE TRAP SYSTEM

6.1 General

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. The 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. 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 (i.e. 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.

- Ignorable trap conditions
- Non-ignorable trap conditions
- Fatal trap conditions

Ignorable trap conditions do not require any handling; they may be disabled and will have no effect on program execution. Non-ignorable trap conditions require some kind of handling. If the current domain does not have a handler for it, the trap is propagated to the mother domain. After handling, program execution may continue.

Fatal trap conditions make it impossible to continue execution of the process. The CPU will report to the I/O processor, which will take appropriate action depending on the kind of trap.

The 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.

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.

6.2 Trap handler routines

Most traps may be handled by a routine in the CPU. 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 I/O processor 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. For example, 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 I/O processor.

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 calling sequence prior to the trap situation may be totally unrelated to the mother/child links.

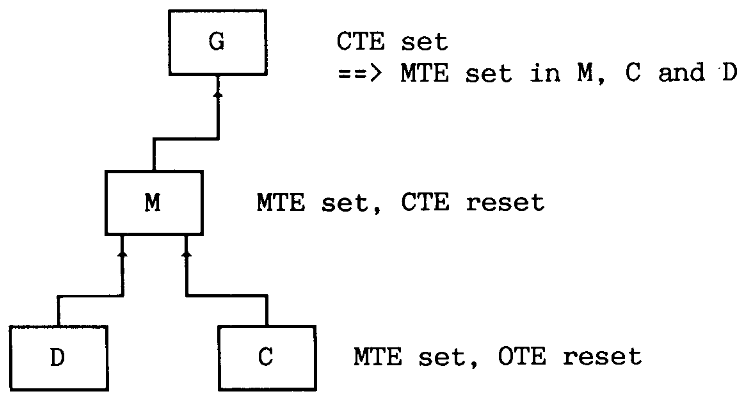
6.3 Searching for a trap handler

Three registers in the CPU are used for trap enabling: The Own Trap Enable (OTE), the Mother Trap Enable (MTE) and the Child Trap Enable (CTE) 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 occurring 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.

MTE is 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.



```
Trap in C : OTE reset,MTE set=> trap propagated to M
     in M : CTE reset       => trap propagated further
     in G : CTE set         => trap handled in G
```

Figure 17. Trap propagation

The I/O processor will always be the mother of the upper domain. Trap conditions are always enabled in the I/O processor. Non-ignorable trap conditions may be enabled in the CPU and handled by some program in the CPU. If they are not, they will be reported to the I/O processor. Fatal trap conditions are always reported directly to the I/O processor.

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 by either setting or resetting it. An attempt to change a bit in OTE, that is to reset in TEMM, will be ignored, while a change in an OTE bit that is set in the TEMM will have the desired effect.

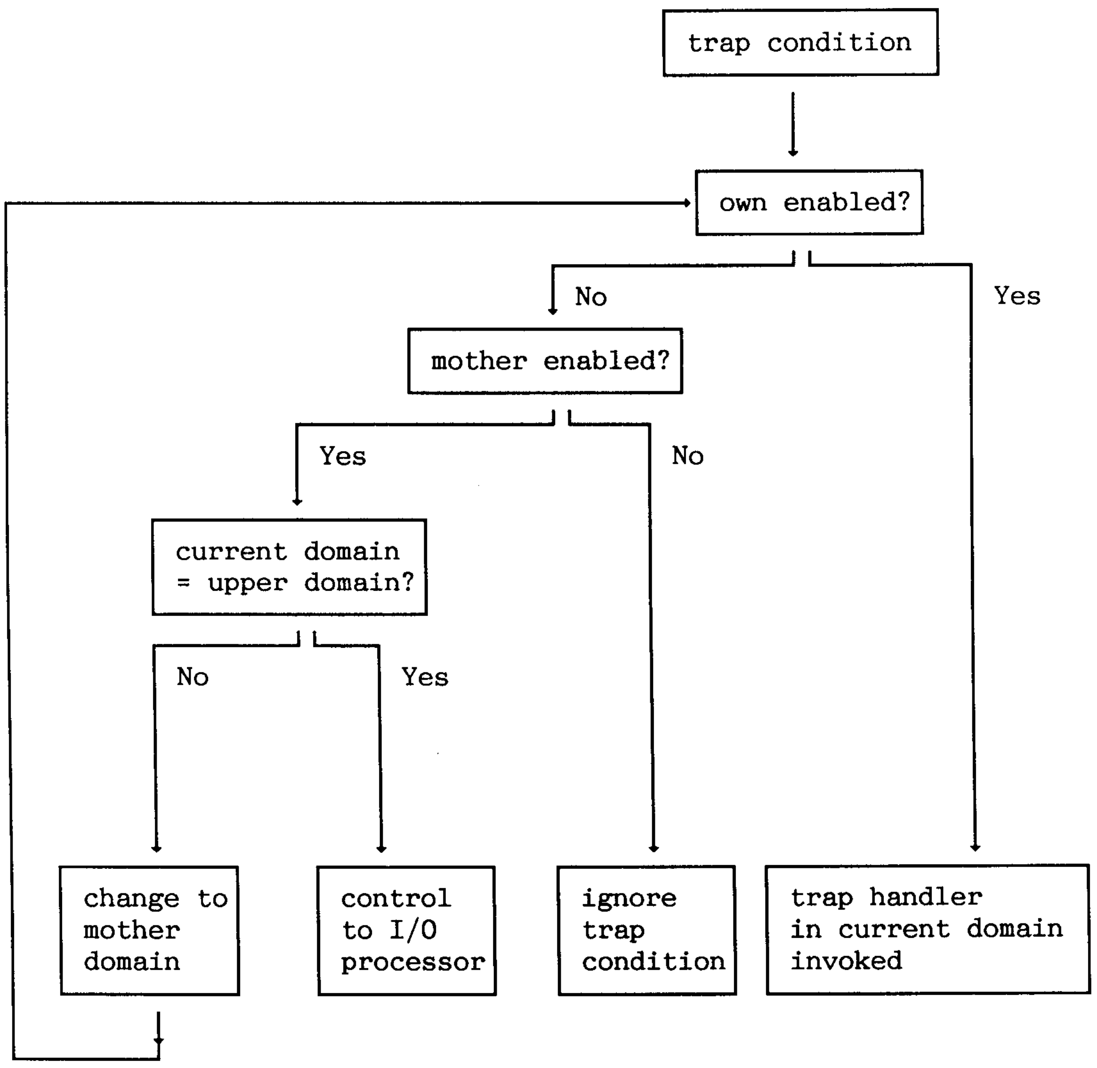


Figure 18. Treatment of non-fatal trap conditions

6.4 Trap handler data field

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 Nth element of this array must hold the start address of the routine to handle the Nth 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 section 13.10).

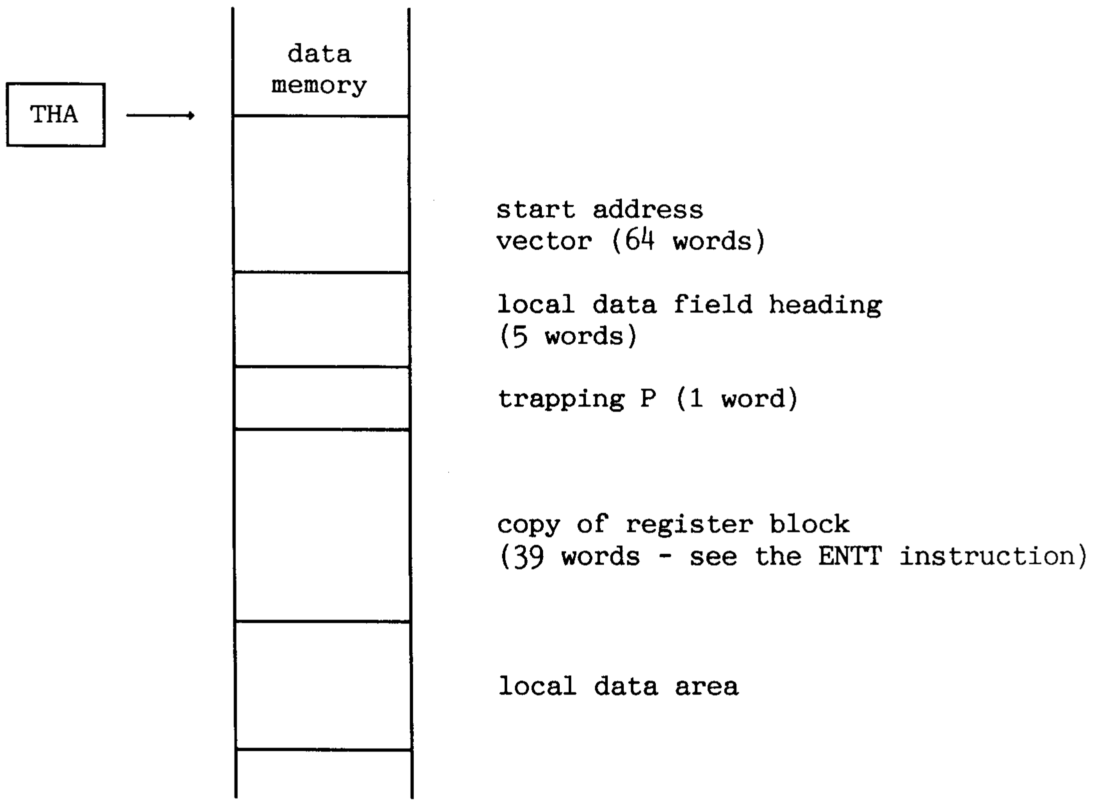

THA
data memory
start address vector (64 words)
local data field heading (5 words)
trapping P (1 word)
copy of register block (39 words - see the ENTT instruction)
local data area

Figure 19. Trap handler start address and local data field

When a trap handler is invoked, trapping P (the address of the instruction that caused the trap condition), the register block, and information about the trap are saved in the local data area of the trap handler.

The P register saved in B.ARG2 holds the address of the instruction to be executed when the trap condition has been taken care of. Trapping P and the saved P register will be equal if the trap is handled before the instruction is executed. The instruction causing the trap will then be re-executed. If the trap is handled after the instruction is executed, the saved P register will point to the next instruction.

The trap handler data area is not re-entrant, due to the fixed location. As long as a trap is being handled, another trap condition should not arise in the same domain. The Own Trap Enable register (OTE) is therefore cleared, forcing propagation to the mother domain of any trap condition occurring during trap handler execution. The OTE register is reloaded from the domain information table on return from the trap handler.

A mother domain which itself is inside a trap handler will not be entered to handle a trap for one of its child domains. A trap in that case not handled locally in the child domain will be propagated to its grandmother.

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).

Modification of status bits is done by changing 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.

If several traps to be handled before or during instruction execution occur together, only the highest numbered one is handled. All other enabled traps that are of the type before and during, are cleared on trap handler return, before the instruction is re-executed. The re-execution may cause these traps again, and they will be handled normally. A trap handled after instruction execution will cause all enabled before traps and all enabled during traps to be cleared when the status register is loaded. Traps not enabled will be not be cleared in either case.

6.5 The status register

There are 64 bits in the status register. 40 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

System error status bits

6.5.1 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 are 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 only used 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. Floating underflow is an exception; then the Z-bit in all cases, except in the POLY and IXI instructions.

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 borrowing into the most significant bit occurs. The contents of the carry bit are 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. One example of this is executing a square root instruction with a negative argument. It 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, with the sign of the result as it would appear when calculated in unlimited format. An underflow trap in a long instruction, like POLY, will occur at the completion of instruction execution, even if the underflow occurred at an intermediate step.

FO : Floating Overflow will occur in floating arithmetic if the result of an operation is too large to be represented in the floating point format, i.e. a signed exponent requiring more than 9 bits. The largest possible floating point value will be stored in the destination, with the sign of the result as it would appear when calculated in unlimited format. An overflow trap in a long instruction, like POLY, will occur at the completion of instruction execution, even if the overflow occurred at an intermediate step.

BO : BCD Overflow. The destination field in a packed decimal instruction was not wide enough to hold the result of an operation. (BCD arithmetic is a hardware option.)

6.5.2 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 program one instruction at a time.

BT : Branch Trap condition occurs when the next instruction to be executed is other than the one immediately following the last executed instruction; e.g. after a GO, JUMPG, RET, LOOP or conditional jump instruction. The trap condition does not occur 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 (BP) is executed. If BPT is not enabled, a BP instruction will cause an IIC trap condition.

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.

6.5.3 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
THM	trap handler missing	37
PGF	page fault	38

These status bits are all trap conditions. Most are ignorable, but XSE, IIC, IOS, ISE and PV are considered so serious that they are defined as non-ignorable. THM and PGF are defined as fatal. 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).

6.5.3.1 Ignorable trap conditions

IOV : Illegal Operand Value. Operand values exceeding the legal range, e.g. 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 a limit is given for the operand values.

On the IOV trap condition the destination field is not changed.

If the IOV trap condition is ignored the instruction will be terminated (act as a NOOP instruction).

The CPU 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 unsigned greater than the LL register and less than or equal to 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.)

The memory is accessed in 1,2,3, or 4-byte units starting on any byte address. It is the starting address of the access that is checked against LL and HL. Bytes inside the area defined for address trapping by the LL and HL registers will therefore be accessed without causing a trap condition if: 1. the access starts at LL-1 and is 2,3, or 4 bytes long, 2. the access starts at LL-2 and is 3 or 4 bytes long, or 3. the access starts at LL-3 and is 4 bytes long.

These registers are used during program development and debugging for tracing access to a specific location/data block or execution of a routine or instruction sequence. The LL and HL registers are properties of the domain. If a routine call causes transfer to another domain the local LL and HL values will be in effect for the duration of the call.

If enabled, program tracing takes precedence over data tracing; if both ATF and ATR/ATW traps are enabled ATF will be trapped, and ATR/ATW trap conditions are ignored. If ATF is enabled, ATR and ATW bits in the status register are cleared when memory is accessed, even if data accesses are within the guarded area. If ATF is disabled, ATR and ATW bits are set in the status register and may cause a trap if ATR or ATW is enabled.

If LL=HL no traps will occur. If HL<LL access from 0 to HL or greater than LL will be trapped; access to addresses from HL+1 to LL will not be trapped. In a multi-operand instruction, any of the operands may cause a trap. The specified address determines its legality; a multi-byte operand value (halfword, word, float, doublefloat or descriptor) may extend into the protected area without being trapped.

The trap conditions are handled after instruction execution; data are loaded or stored before the trap handler is invoked.

ATF : A program reference within the 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 with data memory reference.

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 with data memory reference. The store is performed.

AZ : An address equal to zero will cause an Address Zero trap condition. INIT will set B.PREVB 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 address zero will also cause an AZ trap condition.
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 or equal to the maximum number of elements (length) described by the descriptor length word. A Descriptor Range trap condition will also occur if an empty string (length zero) is used in a string or BCD (packed decimal) instruction.

The DR bit is set/reset at the end of all string instructions or instructions with descriptor addressing (see section 8.15) with memory access. The index register is incremented even if a trap condition occurs.

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.

STO : When the contents of a new stack pointer (B.SP) in a stack subroutine call are greater than or equal to 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. INIT and ENTM cause stack overflow if main program stack demand is greater than system stack demand. The STO status bit is set/reset for each ENTS, ENTSN, ENTB, INIT, ENTM and GETB 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 occurred).

6.5.3.2 Non-ignorable trap conditions

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

IIC : Illegal Instruction Code. Undefined code, privileged instruction with the PIA status bit reset or execution of a BP instruction with the BPT trap disabled.

IOS : Illegal Operand Specifier. Constant operands as destination, ALT prefix on routine argument, type conflict between instruction and operands or non-constant number of arguments to call and polynomial instructions. Also, some special instructions (TSET, RDUS) does not allow register or constant operands.

ISE : Instruction Sequence Error. Illegal subroutine entry point, illegal domain call nesting or execution of an entry point instruction without comming directly from a subroutine call instruction.

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

6.5.3.3 Fatal trap conditions

THM : Trap Handler Missing. The location pointed to by the trap handler vector does not contain an ENTT instruction, or the ENTT operands contain values causing non-ignorable traps.

PGF : PaGe Fault. This trap may be caused by all instructions, and is a signal to the I/O processor that another page has to be swapped in from backing storage. If a page fault arises with the process switch disabled, it will cause a disable process switch error trap. Page fault is also caused if a memory management table lookup gives zero as result.

6.5.4 Signalling, synchronization and miscellaneous 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 section 8.15) to indicate that the last element in the array is accessed, in the LIND and CIND instructions an illegal index, to indicate and in string instructions to indicate termination conditions. CIND, LIND and string instructions will always leave a status in K regardless of its previous value, while descriptor addressing may set but never clear the K flag.

PRT : PRogrammed Trap. A process in the CPU 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 switch 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. It is defined in the domain information table.

PD : Part Done. This bit is used by the microprogram in long interruptable instructions to indicate if the instruction is to be restarted, e.g. 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 CPU has protection against bad synchronization procedures. Synchronization procedures can execute with the process switch disable status bit set. If this bit is set for more than 256 microcycles (including the 2 spent in the SOLO instruction), a process switch timeout trap condition occurs. Most simple instructions, like load, store, and simple arithmetic, execute in one microcycle per operand specifier. When executing with the 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.

Ignorable trap conditions are ignored in SOLO-TUTTI sequences regardless of enabling of these traps.

PSD	:	Process Switch Disabled. The process switch disable bit is only modifiable by the SOLO and TUTTI instructions.
DT	:	Disable process switch Timeout. Timeout occurs if the process switch has been diabled for more than 256 microcycles.
DE	:	Disable process switch Error. Occurs if a non-ignorable process switch (such as Page Fault) occurs while the process switch is disabled.

6.5.5 System error status bits

Code	Name	Bit no.
PWF	power failure	39

The system error status bits are all fatal CPU traps. On detection, they are reported directly to the I/O processor.

PWF : Power failure.

6.5.6 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	

6.5.7 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, i.e., the sequential execution of the program is interrupted and control is passed to a trap handler
- F - fatal CPU error, i.e., another processor in the system must solve the trap condition

A special case exsists for the 'trap handler missing' trap. This trap is nonignorable if a trap handler for this exception exists somewhere in the hierarchy of domains running in this processor. The condition is fatal if no such handler exists.

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 Trapping P 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 <u>should not modify</u> the saved register block.

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 Trapping P location in the trap handler local data area contains the address of the instruction causing the trap.

Bit no.	Name	Code	Trap type	Modifiable(M)	Trap handled before(B), during(D), or after(A)
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	I	M	D
28	stack underflow	STU	I	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	trap handler missing	THM	F		B
38	page fault	PGF	F		D
39	power fail	PWF	F		A

7 DATA TYPES

7.1 Introduction

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:

- Register operands

- Variable operands residing in data memory

- Constants residing in program memory,
  as a part of the instruction using them

7.2 Data types

The ND-500 instruction set handles several basic data types: Bit, byte, halfword, word, float, doublefloat and packed decimal (BCD), abbreviated as BI, BY, H, W, F, D and P respectively. (Packed decimal is a hardware option.) 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 ND-500 is byte addressable, 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 either as a signed or as 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

31	30 22	21 0

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 +/-8.6*(10**(-78)) to +/-5.8*(10**76) and exactly 0, with an accuracy of approximately 7 decimal digits. An operand with exponent = 0 is treated as exactly zero, with no respect to the sign nor the mantissa. Minus zero (all but bit 31 zero) will only be returned from an operation generating floating underflow.

The smallest ΔX to be added to 1.0 is 1.192093180*10**-6.

7.2.6 Double precision floating point

63	62 54	53 0

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 +/-8.6*(10**(-78)) to +/-5.8*(10**76) and exactly 0, with an accuracy of approximately 16 digits. An operand with exponent = 0 is treated as exactly zero, with no respect to the sign nor the mantissa. Minus zero (all but bit 63 zero) will only be returned from an operation generating floating underflow.

The smallest ΔX to be added to 1.0 is 2.775557562*10**-17.

Floating point numbers are always normalized, - i.e. 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.

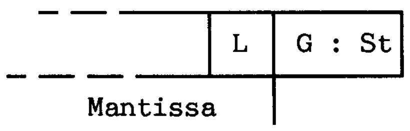
Examples:

```
                   1 (implicit bit)
                   v

-1.0   = 1 100000001 00000000000000000000000 = -1*2**(257-256)*0.5

12.75  = 0 100000010 10011000000000000000000 =  1*2**(260-256)*0.796875

0.5    = 0 100000001 00000000000000000000000 =  1*2**(257-256)*0.5

0.375  = 0 011111111 10000000000000000000000 =  1*2**(255-256)*0.75

-5.0   = 1 100000011 01000000000000000000000 = -1*2**(259-256)*0.625

0.0    = 0 000000000 00000000000000000000000     (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 labelled as follows:

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



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

Figure 20. 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 <u>Descriptor</u>

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	Number of elements (N)
bytes 4 to 7	Address of element 0 (A)

Figure 21. A descriptor

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 ND-500 memory is a 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.

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) <u>byte</u>. Addressing with short address codes is either B or R relative and has <u>word</u> as the displacement unit. The memory must then be looked on as if the basic unit is a 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.

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 they 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.

Figure 22. Data formats in main memory

7.4 Data in registers

Data may be loaded to the registers in the ND-500 CPU register block. Integer data types, i.e. BI, BY, H and W data, may be loaded to the four Integer registers (In, n=1,2,3,4). Floating point data types, i.e. F and D data, may be loaded to 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 loaded to 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.

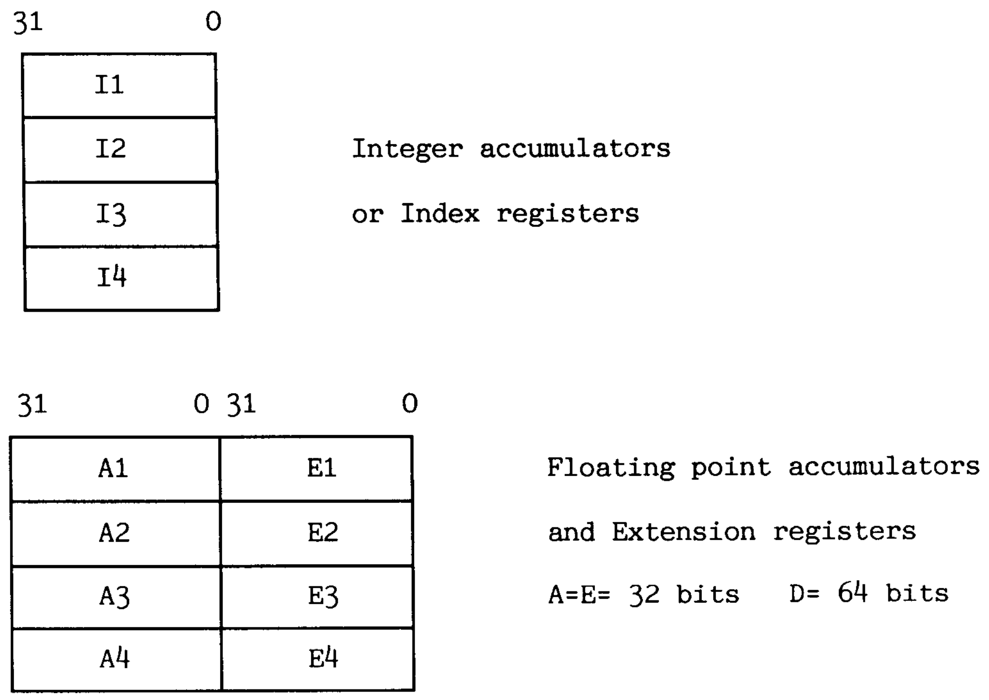


Figure 23. Arithmetic registers

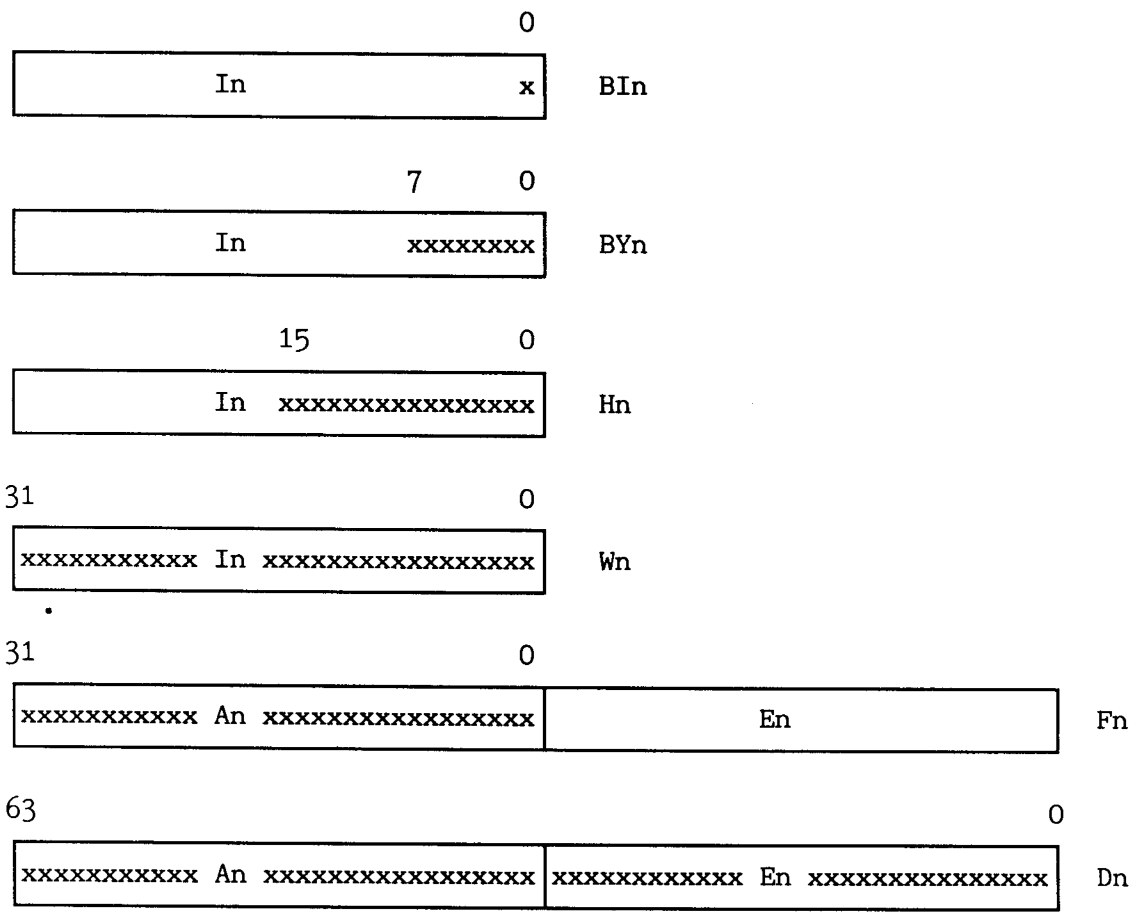

0
In
x
BIn
7
0
In
xxxxxxxx
BYn
15
0
In
xxxxxxxxxxxxxxxx
Hn
31
0
xxxxxxxxxxx In xxxxxxxxxxxxxxxxx
Wn
31
0
xxxxxxxxxxx An xxxxxxxxxxxxxxxxx
En
Fn
63
0
xxxxxxxxxxx An xxxxxxxxxxxxxxxxx
xxxxxxxxxxxx En xxxxxxxxxxxxxxx
Dn

Figure 24. 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 to the register.

When single float data are loaded to one of the Fn registers, i.e. 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:

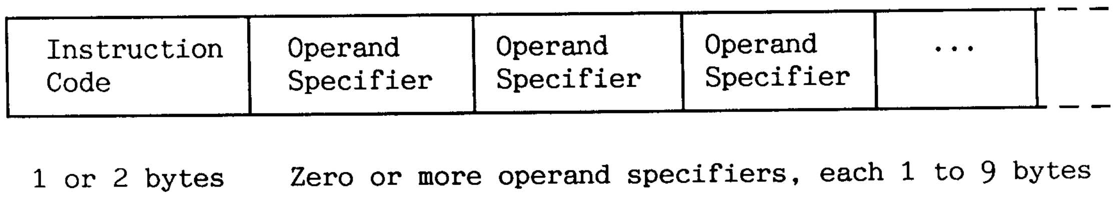


Figure 25. Instruction format

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

8.2 General and direct operands

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

Figure 26. 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 unambiguously as the contents of the following 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, e.g. <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 must have an address code.

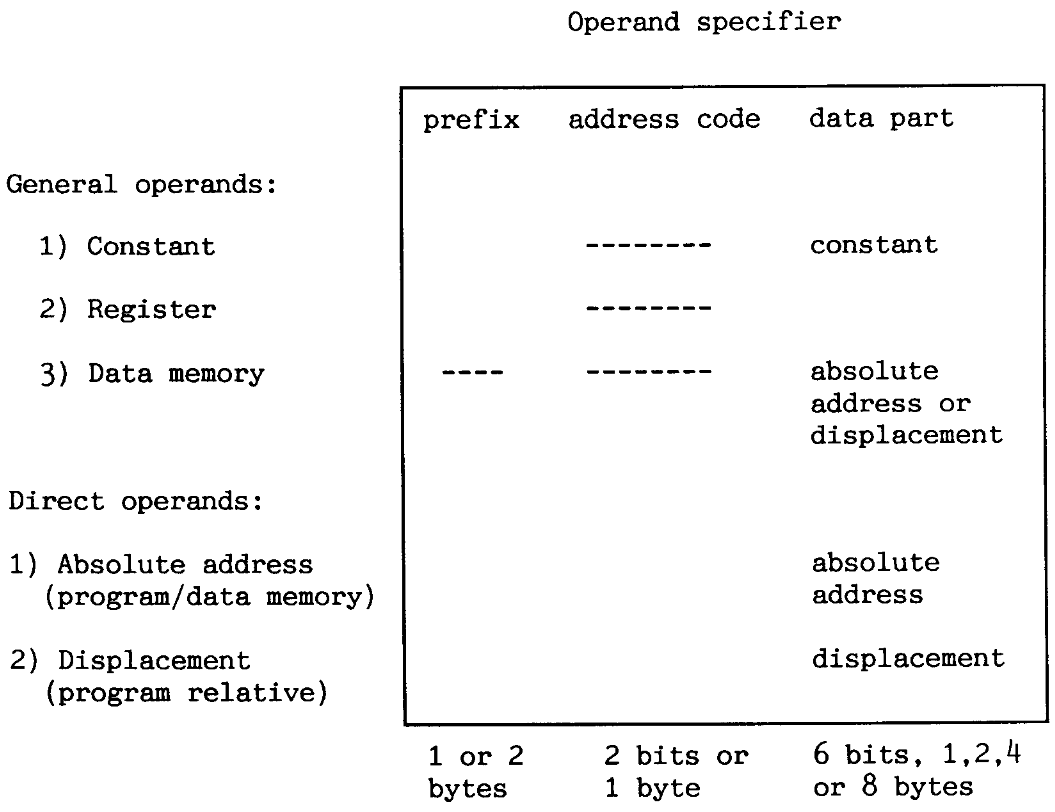


Figure 27. Operand specifier structures

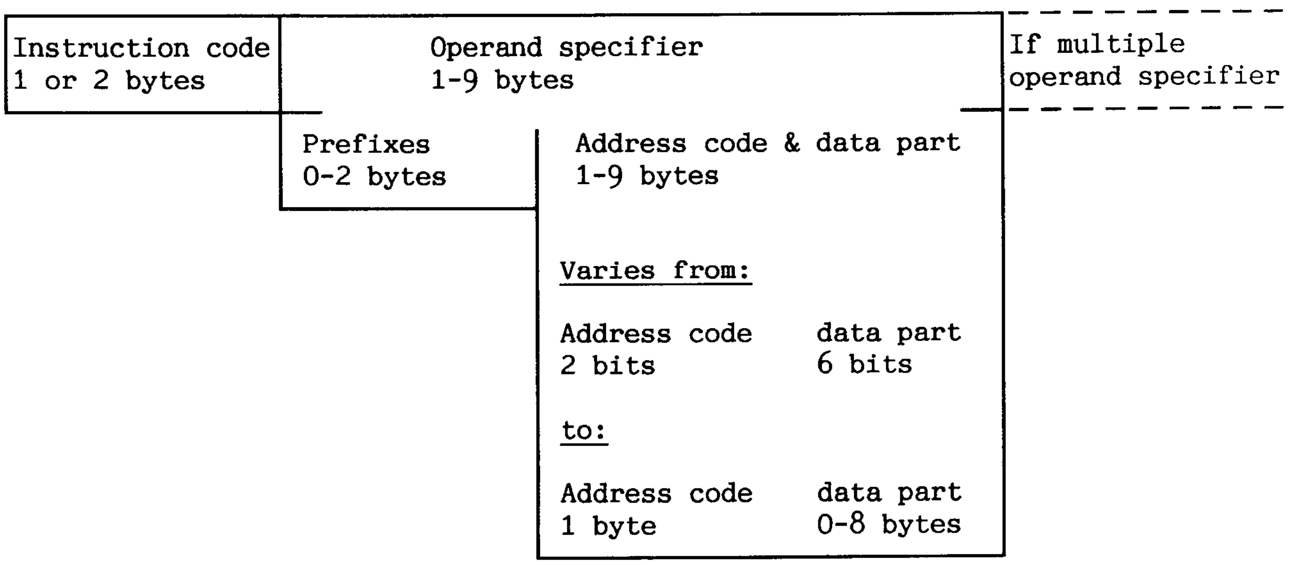


Figure 28. Operand Specifier Layout

8.2.1 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 data part length specifiers (in the ND-500 assembler notation), names and sizes are as follows (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

The table below shows the 10 address modes and the 6 data part length specifiers. Legal combinations are marked with ●. Post-index is abbreviated as P.I.

Address mode	Data part length specifier						No data part
^x,data part length specifier;	: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							●

Figure 29. ND-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.

Addresses always occupy four bytes. 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.

Normally the ND-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 apply to constants only.) In examples shown, a data part length specifier is used only when forcing a non-default data part length.

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 then followed by the operand specifiers.

The only two prefix combination allowed 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.

8.2.2 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. The index is signed, and 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.

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> :B (Rn) byte displacement	ea=(B)+d+p*(Rn)	0D4H+y	324B+y
B. <displ> :H (Rn) halfword displacement		0D8H+y	330B+y
B. <displ> :W (Rn) 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
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

```
if (Rn)+1 >> descriptor.length then
   descriptor range trap condition
endif
if (Rn)+1 >>= descriptor.length then
   1=:status.K
endif
if not descriptor range trap then
   perform addressing with Rn as post-index
   if data access then
       (Rn)+1=:Rn
   endif
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.

8.4 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
ea = (B)+d*4   (B.<displ>:S)
```

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 contents of the base register.

A 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.

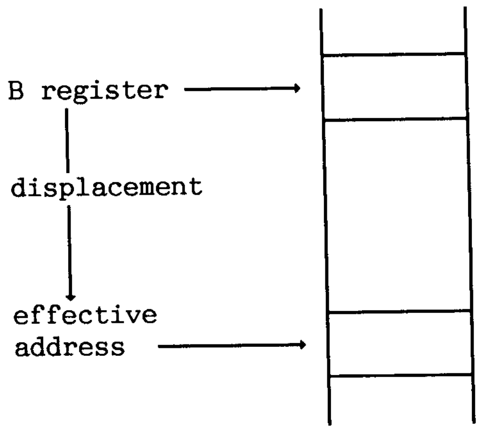
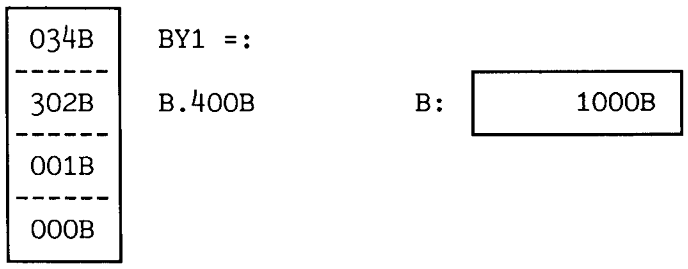


Figure 30. Local addressing

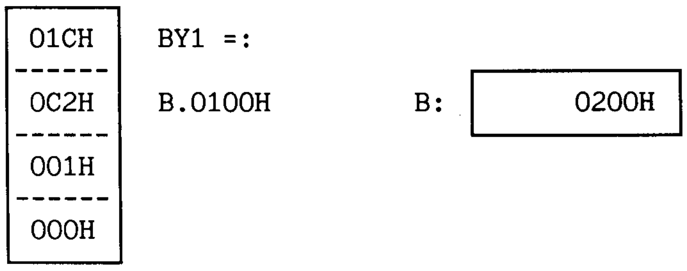
Example:



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

Octal

Hexadecimal



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

8.5 Local, post-indexed addressing

Assembly notation	Name	Hex code	Octal code
B.<displ>(Rn)	local, post-indexed		
B.<displ> :B (Rn)	local, post-indexed, byte displacement	OD4H+y	324B+y
B.<displ> :H (Rn)	local, post-indexed, halfword displacement	OD8H+y	330B+y
B.<displ> :W (Rn)	local, post-indexed, word displacement	ODCH+y	334B+y

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

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

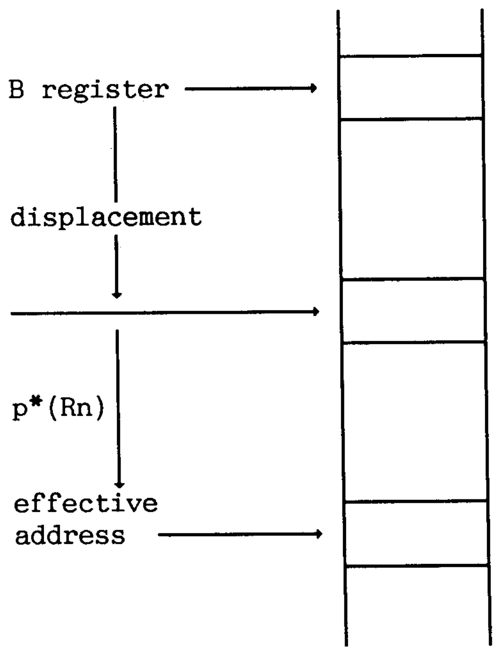


Figure 31. Local, post-indexed addressing

Example:

```
 176005B   BI2 :=
 -------
   332B    B.170:H(R3)         B:   10000B
 -------
   000B
 -------
   170B                        R3:    400B
```

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

Octal

Hexadecimal

```
  011H     BI2 :=
 ------
  0DAH     B.078H:H(R3)        B:   01000H
 ------
  000H
 ------
  078H                         R3:   0100H
```

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

8.6 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.

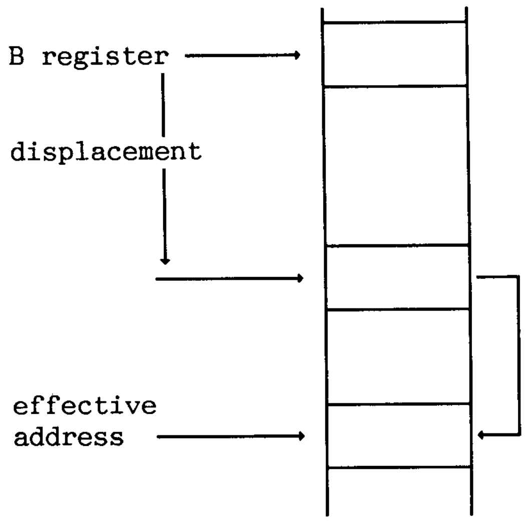


Figure 32. Local indirect addressing

<u>Example:</u>

```
+------+
| 133B |   F4 +
|------|                          +------------+
| 305B |   IND(B.120B:B)       B:  |       400B |
|------|                          |------------|
| 120B |                    520B:  |      1000B |
+------+                          +------------+
```

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

```
Octal
--------------
Hexadecimal
```

```
+------+
| 05BH |   F4 +
|------|                          +------------+
| 0C5H |   IND(B.050H:B)       B:  |      0100H |
|------|                          |------------|
| 050H |                   0150H:  |      0200H |
+------+                          +------------+
```

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

8.7 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 contents of the base register. This sum is interpreted as an address. The contents of the word with this address are added to the contents 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.

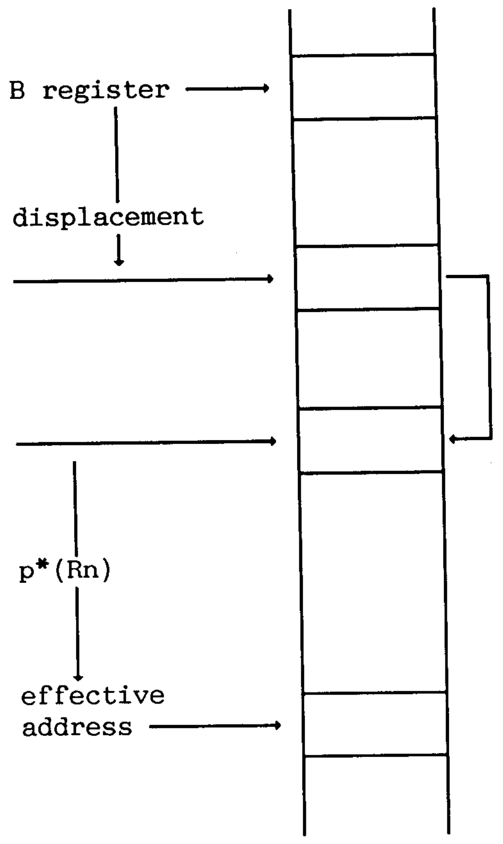


Figure 33. Local indirect, post-indexed addressing

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)        01B0H:   |      0400H |
|------|                                  |------------|
| 030H |                             R4:  |       068H |
+------+                                  +------------+

ea = ((B)+d)+p*(Rn) = (01B0H)+2*068H = 0400H+0D0H = 04D0H
```

8.8 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
ea = (R)+d*4   (R.<displ>:S)

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

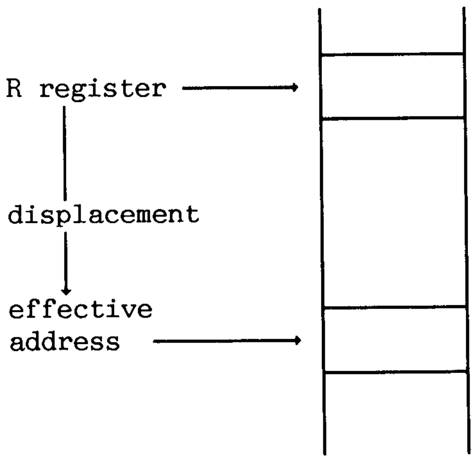
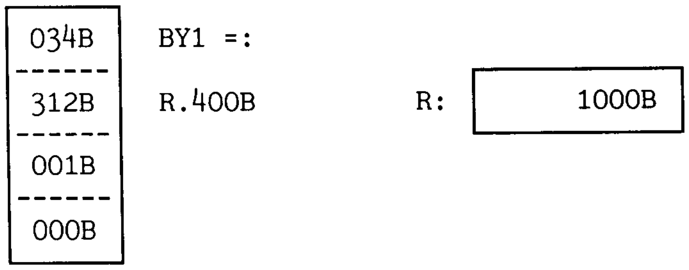


Figure 34. Record addressing

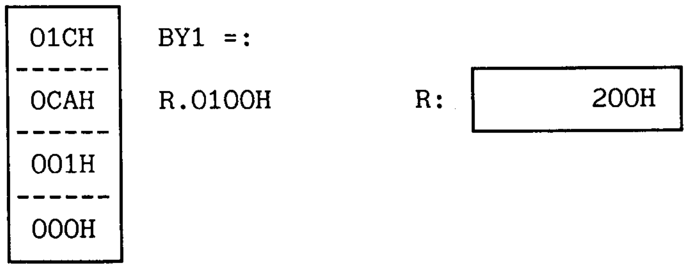
Example:



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

Octal

Hexadecimal



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

8.9 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 are added to the unsigned displacement of the address code. This sum is taken as the address of the operand.

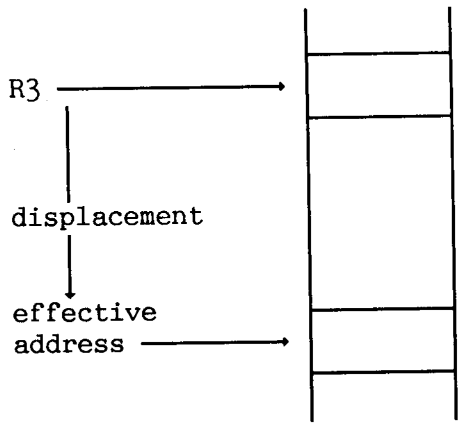


Figure 35. Pre-indexed addressing

Example:

```
+------+
| 165B |  D2 *
|------|
| 372B |  R3.400B              R3: |  10000B  |
|------|
| 001B |
|------|
| 000B |
+------+
```

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

Octal

Hexadecimal

```
+------+
| 075H |  D2 *
|------|
| 0FAH |  R3.0100H             R3: |  01000H  |
|------|
| 001H |
|------|
| 000H |
+------+
```

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

8.10 Absolute addressing

Assembly notation	Name	Hex code	Octal code
<label>	absolute addressing	OC4H	304B

ea = a

When the address code is equal to 304B, OC4H, the four bytes following the address code are taken as the address of the operand.

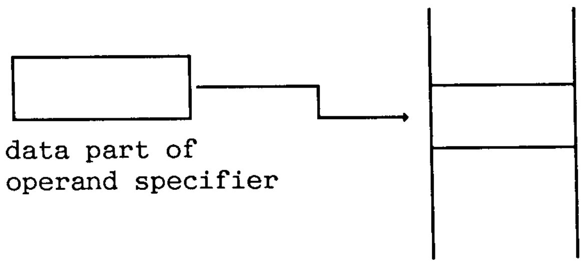


Figure 36. Absolute addressing

Example:

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

ea = 2002044522B

Octal
--------------
Hexadecimal

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

ea = 010084952H
```

8.11 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 contents 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.

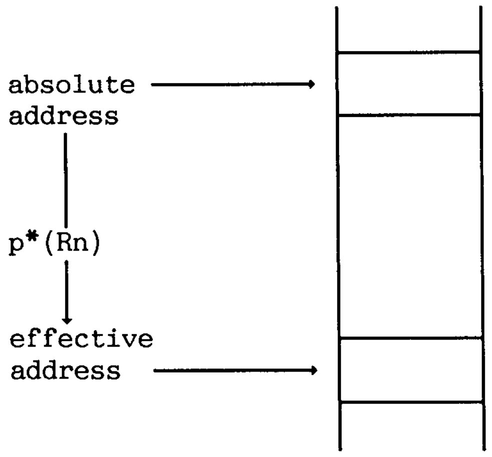


Figure 37. Absolute, post-indexed addressing

Example:

```
 ------
| 020B |  W1 :=
|------|
| 341B |  2000B(R2)      R2:  |        200B |
|------|
| 000B |
|------|
| 000B |
|------|
| 004B |
|------|
| 000B |
 ------
```

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

Octal

Hexadecimal

```
 ------
| 010H |  W1 :=
|------|
| 0E1H |  0400H(R2)      R2:  |        080H |
|------|
| 000H |
|------|
| 000H |
|------|
| 004H |
|------|
| 000H |
 ------
```

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

8.12 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 part of the operand specifier. It resides in the program memory and cannot be modified by any instruction. The value of the operand may have a length of six bits or one, two, four or eight bytes.

Constant operands are illegal for all write instructions, e.g. store, swap, or shift instructions. They are also illegal as destination operand(s) for multi-operand instructions, and in certain special instructions like TSET and RDUS. 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		

Table 7. 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 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

	Constant operand type					
Instruction 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

Table 8. Treatment of constants as operands

8.13 Register addressing

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

One of the registers may be the operand of an instruction. If the data type of an instruction is an 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, as a destination in the BMOVE instruction or as an argument to certain special instructions (such as TSET and RDUS).

8.14 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.

Alternative addressing is illegal for register addressing and constant operand addressing.

8.15 Descriptor addressing

Assembly notation	Name	Hex code	Octal code
DESC(<operand>)(Rn)	descriptor	OFOH+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 except ALT, constant or register. <operand> may be post-indexed, selecting an element in an array of descriptors, in which case the post-index scaling factor 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 number of elements in 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. Elements are indexed from zero; the legal index range is 0 to descriptor.length-1.

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 is set and a descriptor range trap condition occurs.

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
if not descriptor range trap then
    perform addressing with Rn as post-index
    if data access then
        (Rn)+1 =: Rn
    endif
endif
```

Note that an access outside the string as defined by the descriptor is carried out if the descriptor range trap is not enabled.

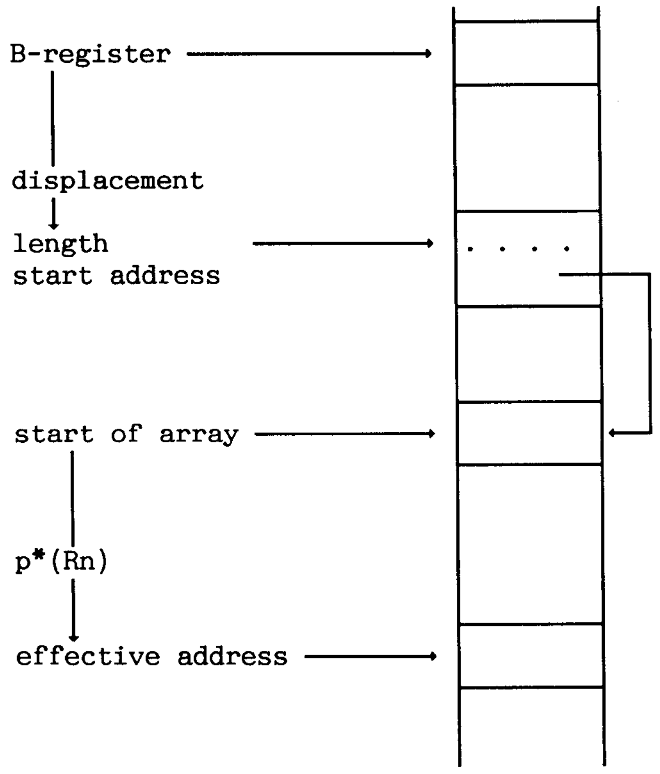


Figure 40. Addressing with a descriptor

Example:

011B	H2 . :=	B:	400B
362B	DESC(B.100B)(R3)	500B:	100B
301B		504B:	2000B
100B		R3:	50B

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

Octal

Hexadecimal

00DH	H2 :=	B:	0100H
0F2H	DESC(B.040H)(R3)	0140H:	040H
0C1H		0144H:	0400H
040H		R3:	028H

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

8.16 Direct operands

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. They are 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.

8.16.1 Displacement addressing

The ND-500 instructions LOOP, LOOPI, LOOPD, GO and IF <rel> GO have displacement (program relative) addressing. Each instruction has two instruction codes, one for the byte displacement part and one for the halfword displacement part. GO is also available with the word displacement part. The displacement is signed, and is the distance from the first byte of the current instruction to the first byte of the addressed instruction.

```
        (P) + d -> (P)
```

8.16.2 Absolute program addressing

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

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

8.16.3 Absolute data addressing

The INIT and ENTM instructions are followed by the 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 ND-500 INSTRUCTION SET

The ND-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:

Instruction Code	Operand Specifier	Operand Specifier	Operand Specifier	...
1 or 2 bytes	Zero or more operand specifiers, each 1 to 9 bytes			

Figure 41. 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 computer. 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 ND-500 instructions one of the general registers or one of the floating-point registers is used as an 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, and the general register (Rn) when the data type is integer.

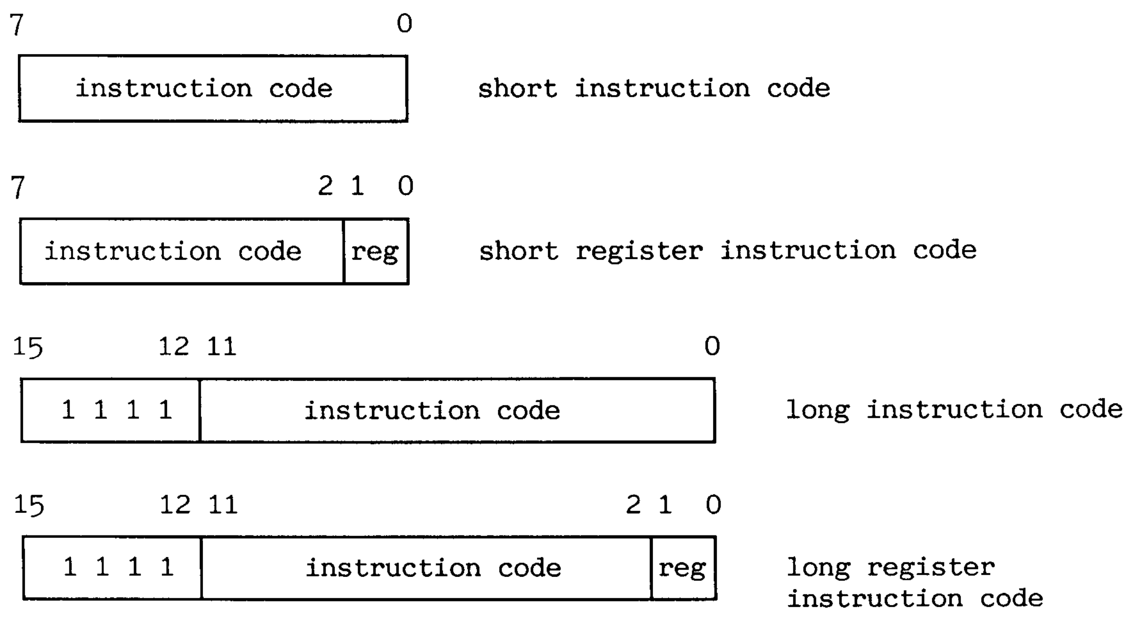


Figure 42. Instruction Code Formats

All the upper 4 bits of a long (two byte) instruction code are set, which means that such codes are in the range 170000B to 177777B, 0F000H to 0FFFFH.

The instruction set is described using the syntax explained below. Optional syntax elements enclosed are in brackets, [ ]. Brackets followed by an "n" mean 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 character string used as a descriptive term. For example, 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	-	read, write and 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.

ACTUAL OPERAND VALUE

The actual operand value used may be the value found in the instruction or the value found at the address specified by the instruction, determined by the addressing mode. In the descriptions of the operation performed in the following chapters, dereferencing of source operands is implicit if the operand is an address. For example,

```
tn  ADD3 <a/r/t>, <br/t>, <c/w/t>

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

In the instruction

```
W3  ADD3 SOU, 5, DES
```

SOU is an address (a label); the value found at this address is the <a> operand value. The <b> operand is the value 5 rather than the value found at address 5; the operand specifier is CONSTANT type. DES is the address of the <c> operand.

If the actual source operand value is the address, rather than the value found at that address, the description of the operation indicates this by the notation addr(<operand>). Take, for example, the LADDR instruction:

```
tn LADDR <operand/aa/t>

Operation: addr(<operand>) -> Rn
```

DATA STATUS BITS

Data status bits not mentioned in the instruction description are always cleared after the instruction has been executed. If the status bit is conditionally set a TRUE condition causes the bit to be set (1), a FALSE condition causes it to be reset (0).

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 on the next page):

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

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

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

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

```
         Program memory                                Data memory

                |   .   |                                    |   .   |
                |   .   |                 B ──→              | 000B  | 10000B
  P ──→         | 374B  | 150300B                            |   .   |
                | 005B  |                 displacement       |   .   |
                | 332B  |                                    | 000B  | 10170B
                | 000B  |                 p * Rn             |   .   |
                | 170B  |                                    |   .   |
                |   .   |                 effective ──→      | 020B  | 10230B
                |   .   | 150305B         address            |   .   |
                |   .   |                                    |   .   |

                               Registers

   P :   ┌──────────┐                ┌──────────┐
   B :   │  150300B │                │  150305B │
   R2:   │   10000B │                │   10000B │
   R3:   │  770140B │                │        1 │
         │     403B │                │     403B │
         └──────────┘                └──────────┘

        Before execution            After execution
```

Octal

Hexadecimal

```
         Program memory                                Data memory

                |   .   |                 B ──→              | 000H  | 01000H
  P ──→         | 0FCH  | 0D0C0H                             |   .   |
                | 005H  |                 displacement       |   .   |
                | 0DAH  |                                    | 000H  | 01078H
                | 000H  |                 p * Rn             |   .   |
                | 078H  |                                    |   .   |
                |   .   |                 effective ──→      | 010H  | 01098H
                |   .   | 0D0C5H          address            |   .   |
                |   .   |                                    |   .   |

                               Registers

   P :   ┌──────────┐                ┌──────────┐
   B :   │   0D0C0H │                │   0D0C5H │
   R2:   │   01000H │                │   01000H │
   R3:   │  03F060H │                │        1 │
         │     103H │                │     103H │
         └──────────┘                └──────────┘

        Before execution            After execution
```

10 DATA TRANSFER 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 goes in the 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> are 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 datatype-dependent part of the contents of the specified register is stored in the memory location or register specified in the operand specifier. The datatype-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, forcing word displacement part

```
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	OFCOAH	176012B

Operation: B -> <operand>

Description:

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

Trap conditions: Addressing traps

Data status bits:

```
    B register = 0     -> Z
    B register.signbit -> S
```

Example:

Store B in local variable CURRB indexed by R1

```
    B =: B.CURRB(I1)
```

10.6 Store record register

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

Assembly notation	Name	Hex code	Octal code
R =:	store record register	OFC09H	176011B

Operation: R -> <operand>

Description:

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

Trap conditions: Addressing traps

Data status bits:

```
    R register = 0     -> Z
    R 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 are stored in the second, and the original contents of the second operand are stored in the first. The operands are assumed to have the same data type (see section 7.3 on page 75).

Trap conditions: Addressing traps

Data status bits:

```
original contents of <op1> = 0      -> Z
original contents of <op1>.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 instruction subtracts the operand from the contents of 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 Overflow, Floating Underflow

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 <op1/r/t>,<op2/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: <op1> - <op2>

Description:

The instruction 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 are 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	OFE10H+(n-1)	177020B+(n-1)
BYn	INV	byte invert register	OFE14H+(n-1)	177024B+(n-1)
Hn	INV	halfword invert register	OFE18H+(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 contents 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	OFF1OH+(n-1)	177420B+(n-1)

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

Description:

The one's complement of the contents 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 contents 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
0           -> S
overflow    -> O  (integer)
```

Example:

Take the absolute value of double precision register D1

```
    D1  ABS
```

10.16 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 zeroes. For all integer data types, the entire register is cleared.

Trap conditions: None

Data status bits: 1 -> Z

Example:

Clear double register D3

```
    D3 CLR
```

10.17 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 are replaced by zero.

Trap conditions: Addressing traps

Data status bits: 1 -> Z

Example:

Clear the byte FLAGS

```
    BY  STZ FLAGS
```

10.18 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 are replaced by one.

Trap conditions: Addressing traps

Data status bits: All cleared

Example:

Set float argument START to one

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

10.19 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 when and only when integer -1 is incremented.

Trap conditions: Addressing traps, Integer Overflow

Data status bits:

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

Example:

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

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

10.20 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                        -> O
carry from most significant bit -> C
```

Example:

Decrement the halfword record variable STEP on the alternative domain

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

10.21 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 contents of 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)
```

10.22 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 contents of 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
```

10.23 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 contents of 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
```

10.24 Logical shift

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

Assembly notation	Name	Hex code	Octal code
BY SHL	byte shift logically	OFCA8H	176250B
H SHL	halfword shift logically	OFCA9H	176251B
W SHL	word shift logically	OFCAAH	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
```

10.25 Arithmetical shift

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

Assembly notation	Name	Hex code	Octal code
BY SHA	byte shift arithmetically	OFCABH	176253B
H SHA	halfword shift arithmetically	OFCACH	176254B
W SHA	word shift arithmetically	OFCADH	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 to the right

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

10.26 Rotational shift

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

Assembly notation	Name	Hex code	Octal code
BY SHR	byte shift rotationally	OFCAEH	176256B
H SHR	halfword shift rotationally	OFCAFH	176257B
W SHR	word shift rotationally	OFCBOH	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. 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:

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

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

10.27 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
```

10.28 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> are 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
```

10.29 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	177175B
H	CLEBI	halfword clear bit	0FE7EH	177176B
W	CLEBI	word clear bit	0FE7FH	177177B

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
```

10.30 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: All cleared

Example:

Set bit FAILURE in word argument EXCEPTIONS on the alternative domain

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

10.31 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 as many higher numbered bits as necessary to obtain 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
```

10.32 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	OFDECH+(n-1)	176754B+(n-1)
Hn	PUTBF	halfword put bit field	OFDFOH+(n-1)	176760B+(n-1)
Wn	PUTBF	word put bit field	OFDF4H+(n-1)	176764B+(n-1)

Operation: Rn -> specified bit field

Description:

The contents of bit 0 to <field size> - 1 of the specified register are stored in the specified bit field of the operand. In the <operand>, the bit field is composed of the <bit No.> bit and as many higher numbered bits as necessary to obtain 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
```

10.33 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:

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

Description:

The value of the <x> operand is divided by the value of the <y> operand 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
```

10.34 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
```

10.35 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 l

10.36 AMODB - Integer modulo ('87 extension)

Format: tn AMODB <opernad1/r/t>,<operand2/r/t>

Assembly notation	Name	Hex code	Octal code
BYn AMODB :	byte integer modulo	FFBCH	177674B+n-1
Hn AMODB :	halfword integer modulo	FFC0H	177700B+n-1
Wn AMODB :	word integer modulo	FFC4H	177704B+n-1

Operation:

```
<operand1> - (<operand1> div <operand2>) * <operand2>  -> Res
if
  res = 0 then 0 -> result
elseif
  sign(res) >< sign(<operand2>) then res+<operand2> -> result
else
  res -> result
endif
```

Description:

The specified register is loaded corresponding to the SIMULA IMOD definition. The function applies to integer operands only.

Trap Condition: Divide by zero

Data Status Bits:

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

10.37 ENTIER - SIMULA Entier function ('87 extension)

Format: t ENTIER <source/r/t1>,<destination/w/w>

Assembly notation	Name	Hex code	Octal code
F ENTIER	float entier	FDC7H	176707B
D ENTIER	double float entier	FDC8H	176710B

Operation:

```
if int(source) > source then
   int(source) - 1 -> destination
else
   int(source) -> destination
endif
```

Description:

The function calculates the integer part of the source in accordance to the SIMULA Entier definition and stores it as a 32 bit integer in the destination.

Data Status Bits:

result = 0	-> Z
<result>.signbit	-> S
integer overflow	-> O

11 ARITHMETICAL INSTRUCTIONS

11.1 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
carry from most significant bit -> C  (integer)
floating underflow              -> FU
floating overflow               -> FO
```

Example:

Add byte argument FIFTHARG to R3

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

11.2 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 contents of 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
    carry from the most significant bit -> C  (integer)
    floating underflow                -> FU
    floating overflow                 -> FO
```

Example:

Subtract the contents of register F1 from the contents of register F4

```
    F4 - F1
```

11.3 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 contents of the specified register and the product is 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
floating underflow	-> FU
floating overflow	-> FO

Example:

Multiply halfword register R2 by 5

```
H2 * 5
```

11.4 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 are 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, i.e. 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           -> O
floating underflow-> FU
floating overflow -> FO
divisor = 0        -> DZ
```

Example:

Divide float register A3 by the R4th element of argument ARR

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

11.5 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 section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow

Data status bits:

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

11.6 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 section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow

Data status bits:

difference = 0	-> Z	
difference.signbit	-> S	
overflow	-> O	
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
```

11.7 Multiply two operands

Format: t MUL2 <a/r/t>,<b/r/t>,<c/w/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
    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
```

11.10 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 section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow

Data status bits:

```
difference = 0                   -> Z
difference.signbit               -> S
overflow                         -> O
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
```

11.9 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 section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow

Data status bits:

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

Example:

Add R1 and R2 and leave the result in R3

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

11.8 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	OFC62H	176142B
H DIV2	halfword divide two operands	OFC63H	176143B
W DIV2	word divide two operands	OFC64H	176144B
F DIV2	float divide two operands	OFC65H	176145B
D DIV2	double float divide two operands	OFC66H	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, i.e. 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
floating underflow	-> FU
floating overflow	-> FO
<b> = 0	-> DZ

Example:

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

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

11.11 Multiply three operands

Format: t MUL3 <a/r/t>,<b/r/t>rw/t>,<b/r/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 section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow

Data status bits:

```
product = 0         -> Z
product.signbit     -> S
overflow            -> O
floating underflow -> FU
floating overflow  -> FO
```

Example:

Store the product of the second and third element of the word array pointed to by R2 in the first element of the word array pointed to by R2

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

11.12 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, i.e. 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 section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow, Divide by Zero

Data status bits:

```
quotient> = 0        -> Z
quotient>.signbit    -> S
overflow             -> O
floating underflow   -> FU
floating overflow    -> FO
<b> = 0              -> DZ
```

Example:

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

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

11.13 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 section 7.3 on page 75).

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
```

11.14 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 and the quotient is stored in the <c> operand. The remainder is stored in the specified register.

Note that the register content is in compliance with ADA and SIMULA remainder. Separate testing must be done to obtain status. The operands are assumed to have the same data type (see section 7.3 on page 75).

Trap conditions: Addressing traps, Integer Overflow, Divide by Zero

Data status bits:

```
quotient = 0      -> Z
quotient.signbit -> S
overflow          -> O
<b> = 0           -> DZ
```

Example:

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

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

11.15 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 and the product is 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 zero.

Trap conditions: Addressing traps, Integer Overflow

Data status bits:

```
product = 0      -> Z
product.signbit -> S
overflow         -> O
```

Example:

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

```
W1 UMUL B.LEASTX, B.LEASTY, R2
```

11.16 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 as unsigned.

Trap conditions: Addressing traps, Divide by Zero

Data status bits:

```
quotient = 0      -> Z
quotient.signbit -> S
<b> = 0           -> DZ
```

Example:

Divide the arguments LONG and FACT on the alternative domain (LONG/FACT) and leave the quotient in the address on the alternative domain contained in RES, and put the remainder in R3

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

11.17 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:

The <addend> operand, the Carry bit in the status register (treated as 0 or 1) and the contents of the specified register are added and the result is stored in the specified 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	-> O
carry from most significant bit	-> C

Example:

Add variable MOST to R2 with carry

```
W2 ADDC MOST
```

11.18 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:

The Carry bit in the status register (treated as 0 or 1) and the one's complement of <subtrahend> are added to the contents of 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 with carry

```
W2 SUBC  0400H
```

11.19 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 contents of 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 from most significant bit -> C  (integer)
overflow                         -> O
floating underflow               -> FU
floating overflow                -> FO
```

Example:

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

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

11.20 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 is added to the contents of the specified register.

Trap conditions: Addressing traps, Integer Overflow, Floating Overflow, Floating Underflow

Data status bits:

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

Example:

Add local floats UNITCOST times UNITS to F4

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

12 MATHEMATICAL FUNCTIONS

12.1 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:

The value of the <a> operand is raised to the power of the <i> operand. The result is loaded into the specified float or double float register. The <a> operand can be float or double float. The <i> operand is word integer. A negative value of <i> and the value of <a> equal to zero causes an illegal operand value trap condition and the result is set to the largest possible floating point number (approximately 5.8E+76). 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'th power into F3

```
F3 AXI 2.0, STATE
```

12.2 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:

The value of the <i> operand is raised to the power of the <j> operand. 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. A negative value of <j> and a value of <i> different from 1 or -1 will give zero. A negative value of <j> and a value of <i> equal to zero cause an illegal operand value trap condition and a zero result.

When an overflow occurs, the specified register will be loaded with the least significant part of the result from the calculation. The rest of the result is lost, while the status register flags an overflow.

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
```

12.3 Polynomial

Format: tn POLY <x/r/t>,<m/s/BY>,
<cm/r/t>,...,<c1/r/t>,<c0/r/t>

Assembly notation	Name	Hex code	Octal code
Fn POLY	floating polynomial	OFCEOH+(n-1)	176340B+(n-1)
Dn POLY	double float polynomial	OFCE4H+(n-1)	176344B+(n-1)

Operation:

$$<cm>*<x>^{m} + ... + <c2>*<x>^{2} + <c1>*<x> + <c0> \rightarrow Rn$$

Description:

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

If floating overflow or underflow occurs, the trap will not have any effect until the instruction has completed execution, even if the trap condition occurred at an intermediate step. The Z and S bits reflect the final result.

Trap conditions: Addressing traps, Floating Overflow, Floating Underflow, Illegal Operand Specifier

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 constants

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

12.4 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 cause an invalid operation trap condition.

Trap conditions: Addressing traps, InValid Operation

Data status bits: result = 0 -> Z

Example:

Load double float register D1 with the square root of AREA

```
    D1 SQRT AREA
```

12.5 Sine

Format: tn SIN <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn SIN	float sine	0FF58H+(n-1)	177530B+(n-1)
Dn SIN	double float sine	0FF84H+(n-1)	177604B+(n-1)

Operation: sine(<argument>) -> Rn

Description:

The trigonometric sine of <argument> is loaded into the specified float or double float register. The maximum absolute value of <argument> is 65536.0 radians; a larger value will cause an invalid operation trap condition and the specified register will be set to zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Calculate the sine of 2 radians and load into F2

```
F2 SIN 2.0
```

12.6 Arc sine

Format: tn ASIN <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ASIN	float arcsine	0FF5CH+(n-1)	177534B+(n-1)
Dn ASIN	double float arcsine	0FF88H+(n-1)	177610B+(n-1)

Operation: arcsine(<argument>) -> Rn

Description:

The trigonometric arcsine of <argument> is loaded into the specified float or double float register. The result value gives the angle in radians, in the range -pi/2 to pi/2. <argument> should be in the range -1 to +1, otherwise an invalid operation trap condition will occur and the specified register will be set to zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Replace the number in F2 with its arcsine

```
F2 ASIN F2
```

12.7 Cosine

Format: tn COS <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn COS	float cosine	OFF60H+(n-1)	177540B+(n-1)
Dn COS	double float cosine	OFF8CH+(n-1)	177614B+(n-1)

Operation: cosine(<argument>) -> Rn

Description:

The trigonometric cosine of <argument> is loaded into the specified float or double float register. The maximum absolute value of <argument> is 65536.0 radians; a larger value will cause an invalid operation trap condition and the specified register will be set to zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Calculate the cosine of double-precision ANGLE and load into D2

```
D2 COS ANGLE
```

12.8 Arc cosine

Format: tn ACOS <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ACOS	float arc cosine	0FF64H+(n-1)	177544B+(n-1)
Dn ACOS	double float arc cosine	0FF90H+(n-1)	177620B+(n-1)

Operation: arccosine(<argument>) -> Rn

Description:

The trigonometric arccosine of <argument> is loaded into the specified float or double float register. The result value gives the angle in radians in the range 0 to pi. < argument> should be in the range -1 to +1, otherwise an invalid operation trap condition will occur and the specified register is set to zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Load into F4 the arc cosine of the field FOO in the record pointed to by the R register

```
F4 ACOS R.FOO
```

12.9 Tangent

Format: tn TAN <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn TAN	float tangent	OFF68H+(n-1)	177550B+(n-1)
Dn TAN	double float tangent	OFF94H+(n-1)	177624B+(n-1)

Operation: tangent(<argument>) -> Rn

Description:

The trigonometric tangent of <argument> is loaded into the specified float or double float register. The maximum absolute value of <argument> is 65536.0 radians; a larger value will cause an invalid operation trap condition and the specified register is set to zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Calculate the tangent of argument SPREAD and load into F4

```
F4 TAN SPREAD
```

12.10 Arc tangent

Format: tn ATAN <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ATAN	float arc tangent	0FF6CH+(n-1)	177554B+(n-1)
Dn ATAN	double float arc tangent	0FF98H+(n-1)	177630B+(n-1)

Operation: arctangent(<argument>) -> Rn

Description:

The trigonometric arctangent of <argument> is loaded into the specified float or double float register. The result value gives the angle in radians in the range -pi/2 to pi/2.

Trap conditions: Addressing traps

Data status bits:

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

Example:

Load into F4 the arctangent of RAY

```
F4 ATAN  RAY
```

12.11 Arc tangent two argument

Format: tn ATAN2 <num/r/t>, <den/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ATAN2	float arctangent2	0FF70H+(n-1)	177560B+(n-1)
Dn ATAN2	double float arctangent2	0FF9CH+(n-1)	177634B+(n-1)

Operation: arctangent(<num>/<den>) -> Rn

Description:

The trigonometric arctangent of <num>/<den> is loaded into the specified float or double float register. The result value gives the angle in radians in the correct quadrant in the range -pi to pi. A zero value of both <num> and <den> will cause an invalid operation trap condition and the specified register will be set to zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Load into D3 the arctangent of WIDTH divided by DIST

```
D3 ATAN2 WIDTH, DIST
```

12.12 Exponential

Format: tn EXP <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn EXP	float exponential	OFF74H+(n-1)	177564B+(n-1)
Dn EXP	double float exponential	OFFAOH+(n-1)	177640B+(n-1)

Operation: e ** <argument> -> Rn

Description:

The exponential of <argument> is loaded into the specified float or double float register. (e = 2.718281828459045...)

The maximum value of <argument> is 255*ln(2) (approximately 176.75). A larger argument will cause an invalid operation trap and the specified register will be set to the largest possible floating point number (approximately 5.8E+76). An <argument> value less than -255*ln(2) will give a result value of zero.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Load the antilogarithm of NATLOG into D1

```
D1 EXP NATLOG
```

12.13 Natural logarithm

Format: tn ALOG <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ALOG	float natural logarithm	OFF78H+(n-1)	177570B+(n-1)
Dn ALOG	double float nat. logarithm	OFFA4H+(n-1)	177644B+(n-1)

Operation: ln(<argument>) -> Rn

Description:

The natural logarithm (base e = 2.718281828459045...) of <argument> is loaded into the specified float or double float register. <argument> should be positive; zero or negative values cause an invalid operation trap condition and a result of -5.8*10**76.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Load the natural logarithm of the R1th element of global array COEFF into D1

```
D1 ALOG COEFF(R1)
```

12.14 Binary logarithm

Format: tn ALOG2 <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ALOG2	float binary logarithm	0FF7CH+(n-1)	177574B+(n-1)
Dn ALOG2	double float bin. logarithm	0FFA8H+(n-1)	177650B+(n-1)

Operation: log2(<argument>) -> Rn

Description:

The base 2 logarithm of <argument> is loaded into the specified float or double float register. <argument> should be positive; zero or negative values cause an invalid operation trap condition and a result of -5.8*10**76.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Load the binary logarithm of local variable RANGE into F1

```
F1 ALOG2 B.RANGE
```

12.15 Common logarithm

Format: tn ALOG10 <argument/r/t>

Assembly notation	Name	Hex code	Octal code
Fn ALOG10	float common logarithm	0FF80H+(n-1)	177600B+(n-1)
Dn ALOG10	double float common log.	0FFACH+(n-1)	177654B+(n-1)

Operation: log(<argument>) -> Rn

Description:

The base 10 logarithm of <argument> is loaded into the specified float or double float register. <argument> should be positive; zero or negative values will cause an invalid operation trap condition and a result of -5.8*10**76.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

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

Example:

Load the common logarithm of BIGNUMB into F4

```
    F4 ALOG10 BIGNUMB
```

13 CONTROL INSTRUCTIONS

13.1 Unconditional relative jump

Format: GO <<displacement>>

Assembly notation	Name	Hex code	Octal code
GO:B	jump byte	OCOH	300B
GO:H	jump halfword	OC1H	301B
GO:W	jump word	OC2H	302B

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

Description:

Perform a jump relative to the current program counter value. GO uses a direct operand and has three formats, with a byte, halfword, or word displacement part. The displacement is signed and is found in the 1, 2 or 4 bytes following the instruction code.

Trap conditions: Addressing traps, Branch Trap

Data status bits: Unaffected

Example:

Jump to BACK (Assembler will calculate displacement)

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

13.2 Unconditional absolute jump

Format: JUMPG <address/r/W>

Assembly notation	Name	Hex code	Octal code
JUMPG	jump general	OB4H	264B

Operation: <address> -> P

Description:

Perform a jump to the absolute address given by the operand. JUMPG requires a general operand. The <address> operand may not be prefixed by the operand specifier prefix ALT.

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, Branch Trap, Illegal Operand Specifier

Data status bits: Unaffected

Example:

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

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

13.3 Conditional jump

Formats:

```
        IF <rel> GO  <<displacement>>

        IF <rel> GO  <bit No./r/BY>, <<displacement>>
```

Operation:

```
    if  <rel> then
        (P)+<<displacement>> -> P
    endif
```

Description:

A conditional jump will cause transfer of control if and only if a specified condition is true.

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 byte or halfword <<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; no jump is performed if <rel> is ST, the jump is performed if <rel> is -ST.

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, Branch Trap, 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.

Assembly notation	Condition	Name	Hex code	Octal code
IF = GO	Z=1	equal		
IF Z GO		(alt. assembly notation)		
IF = GO:B			0C4H	304B
IF = GO:H			0C5H	305B
IF >< GO	Z=0	unequal		
IF -Z GO		(alt. assembly notation)		
IF >< GO:B			0C6H	306B
IF >< GO:H			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 set		
IF K GO:B			0D0H	320B
IF K GO:H			0D1H	321B
IF -K GO	K=0	flag reset		
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

13.4 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> - <limit> > 0 then
   address of next instruction -> P
else
   P+<<displacement>> -> P
endif
<index> + 1 -> <index>
```

Description:

The <index> operand is incremented by one and compared with <limit>. If it is less than or equal to <limit>, the signed <<displacement>> is added to the program counter; otherwise 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.

The <index> and <limit> operands are of 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.

Trap conditions: Addressing traps, Branch Trap

Data status bits:

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

Example:

Repeat the instructions from AGAIN until local byte COUNTER reaches 100

```
AGAIN: .
       .
       .
      BY LOOPI B.COUNTER, 100, AGAIN
```

13.5 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	OFD23H	176443B
BY LOOPD:H	byte loop decrement	OFD28H	176450B
H LOOPD:B	halfword loop decrement	OFD24H	176444B
H LOOPD:H	halfword loop decrement	OFD29H	176451B
W LOOPD:B	word loop decrement	OFD25H	176445B
W LOOPD:H	word loop decrement	OFD2AH	176452B
F LOOPD:B	float loop decrement	OFD26H	176446B
F LOOPD:H	float loop decrement	OFD2BH	176453B
D LOOPD:B	double float loop decrement	OFD27H	176447B
D LOOPD:H	double float loop decrement	OFD2CH	176454B

Operation:

```
<index> - 1  -> <index>
if <index> - <limit> < 0 then
   address of next instruction -> P
else
   P+ <<displacement>> -> P
endif
```

Description:

The <index> operand is decremented by one and compared with <limit>. If it is greater than or equal to <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>>. <<displacement>> is the number of bytes from the first byte of the loop to the first byte of the LOOPD instruction.

The <index> and <limit> operands are of 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.

Trap conditions: Addressing traps, Branch Trap

Data status bits:

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

Example:

Repeat from TOP until word register R3 is decremented to zero

```
TOP:  .
      .
      .
      W LOOPD R3, 0:W, TOP
```

13.6 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:

```
<index>+<step> -> <index>
if  <step> > 0  and  <index> - <limit> > 0
or  <step> < 0  and  <index> - <limit> < 0  then
   address of next instruction -> P
else
   P + <<displacement>> -> P
endif
if  <step> = 0 then
    illegal operand value trap condition
endif
```

Description:

The value of the <step> operand is added to the <index> operand . If the sign of <index> - <limit> is equal to the sign of the <step> operand, 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, and given a negative <<displacement>>. The <<displacement>> is the number of bytes from the first byte of the loop to the first byte of the LOOP instruction.

The <index>, <step> and <limit> operands are of 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> value of zero will cause an illegal operand value trap condition and execution continues at the next instruction.

Trap conditions: Addressing traps, Branch Trap, Illegal Operand Value

Data status bits:

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

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 R.SIZE, 3.5, 35, LABELL
```

13.7 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	OB5H	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 specified by <subr. addr.>. This is a general operand and it must refer to an entry point instruction. Otherwise an instruction-sequence error-trap condition occurs.

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 effective addresses of the arguments in the instruction are calculated and stored for use by the entry point instruction. The arguments are always interpreted as word integers. 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 data types other 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.

Trap conditions: Addressing traps, Call Trap, Illegal Operand Specifier, Instruction Sequence Error

Data status bits: Unaffected

Example:

Call PRINT with arguments UNIT, FORMAT and the local variable VALUE

```
    CALLG PRINT, 3, UNIT, FORMAT, B.VALUE
```

13.8 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 specified by <<subr. addr.>>. The subroutine address is a direct operand in the four bytes following the instruction code. It must refer to an entry point instruction, otherwise an instruction sequence error trap condition occurs.

The <no of arg> operand must be a constant byte integer, i.e. less than 256. Other data types which are not constants will cause an illegal operand specifier trap condition.

The effective addresses of the arguments in the instruction are calculated and stored for use by the entry point instruction. 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 incorrect for data types other 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.

Trap conditions: Addressing traps, Call Trap, Illegal Operand Specifier, Instruction Sequence 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)
```

13.9 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	ODCH	334B

Operation:

```
<<bottom of stack>>              -> B

<<bottom of stack>> +
<total system stack demand>      -> TOS

<<bottom of stack>> +
<stack demand of main program> -> B.SP
0                                -> B.PREVB
0                                -> B.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> are added and the result is loaded into the top of stack register, TOS. PREVB and RETA are cleared. A value of <stack demand of main program> greater than or equal to <total system stack demand> will cause a stack overflow trap condition.

Trap conditions: Addressing traps, Stack Overflow

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
```

13.10 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 resulting from 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 or CALLG.

The parameters to the subroutine entry point instructions may not be prefixed by the operand specifier prefix ALT.

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. A value of <stack demand of main program> greater than or equal to <total system stack demand> will cause a stack overflow trap condition.

If ENTM is entered from another domain, TOS is not saved on the old stack, but is stored in the domain information table. Also THA, LL and HL are stored and new contents for these registers are fetched from the new domain information table.

ENTM is the only entry point that may be called from another domain.

Trap conditions: Addressing traps, Instruction Sequence Error, Stack Overflow

Initializations performed:

```
<<bottom of stack>>                 -> B
oldB                                -> B.PREVB
TOS                                 -> IND(oldB.SP)
<<bottom of stack>> +
<total system stack demand>         -> TOS
return address                      -> B.RETA -> L
<<bottom of stack>> +
<stack demand of main program>      -> B.SP
number of arguments                 -> B.N
addresses of arguments              -> B.arg
```

If change of domain:

```
0                                   -> B.PREVB
0                                   -> B.RETA
TOS, LL, HL, THA                    -> old domain information table
TOS, LL, HL, THA entries in
new domain information table        -> TOS, LL, HL, THA
```

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.

The call to ENTD must have zero parameters. A non-zero number of arguments will cause an instruction sequence error trap condition.

Trap conditions: Address Trap Fetch, Instruction Sequence Error

Initializations performed:

```
    return address -> L
```

ENTS - enter stack subroutine

Assembly notation	Hex code	Octal code
ENTS <stack demand/r/W>	0B8H	270B

Description:

The <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 B + <stack demand> is greater than or equal to TOS.

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 only the <max no. of arg.> are transferred to the stack, the remaining ones are ignored.

Trap conditions: Addressing traps, Stack Overflow, Instruction Sequence Error

Initializations performed:

B.SP	-> B
oldB	-> B.PREVB
return address	-> B.RETA -> L
newB + <stackdemand>	-> B.SP
number of arguments	-> B.N
addresses of arguments	-> B.ARG

ENTF - enter subroutine

Assembly notation	Hex code	Octal code
ENTF <<address of local data area/r/W>>	ODDH	335B

Description:

Enter subroutine with fixed data area. Variables will keep their values between calls.

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>	ODEH	336B

Description:

ENTFN is similar to ENTF, but only the <max no. of arg.> will be transferred to the stack, the remaining ones ignored.

Trap conditions: Addressing traps, Instruction Sequence Error

Initializations performed:

```
<<address of local data area>> -> B
oldB                           -> B.PREVB
return address                 -> B.RETA  -> L
oldB.SP                        -> B.SP
number of arguments            -> B.N
addresses of arguments         -> B.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>	OBCH	274B

Description:

ENTT is the trap handler entry point. A trap handler is called when a trap condition arises and the trap enable bit is set for the trap in question. When a trap handler routine is called, 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 a local data area for the trap handler routine called. It has a special layout illustrated in the chapter 6 on traps.

The register block is stacked as shown in table 5 on page 15.

The instruction may start at any byte in the first word. 'Trapping P', saved as arg1, is the address of the first byte of the instruction causing the trap.

Trap conditions: Addressing traps, Instruction Sequence Error

(No traps are handled locally.)

Figure 43 shows the layout of the data structure when entering ENTT.

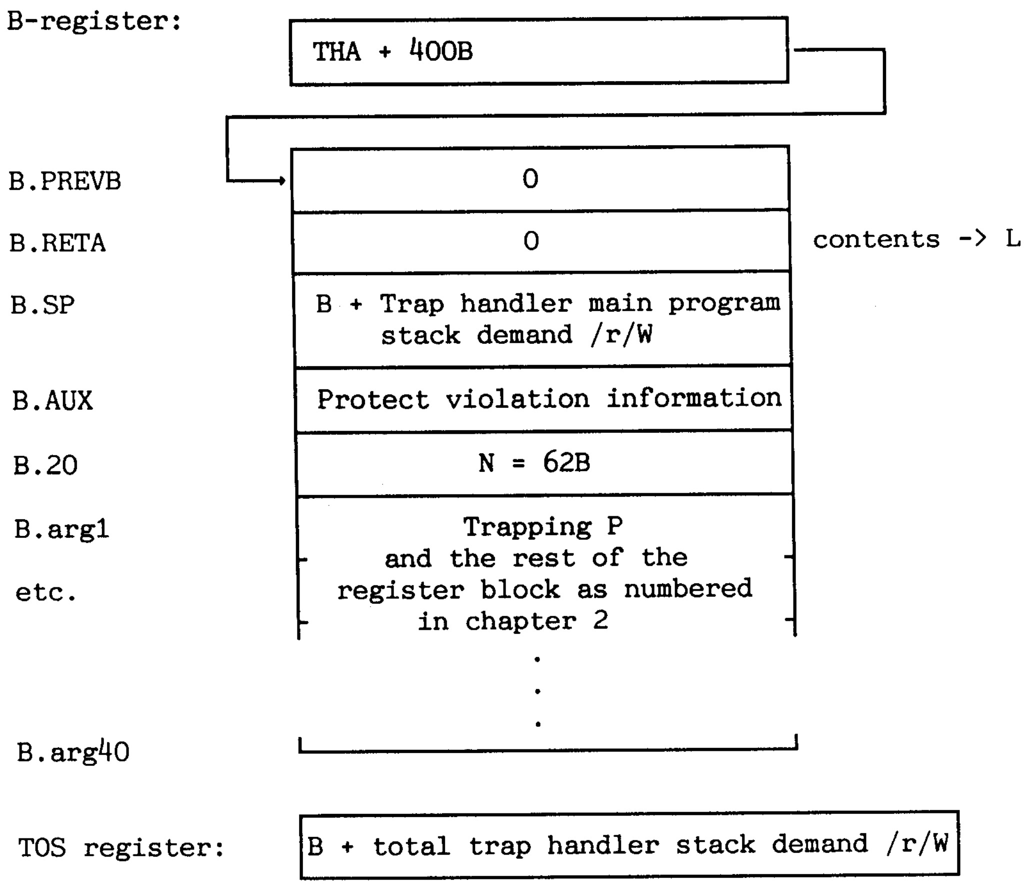

B-register:
THA + 400B
B.PREVB
0
B.RETA
0
contents -> L
B.SP
B + Trap handler main program stack demand /r/W
B.AUX
Protect violation information
B.20
N = 62B
B.arg1
etc.
Trapping P and the rest of the register block as numbered in chapter 2
B.arg40
TOS register:
B + total trap handler stack demand /r/W

Figure 43. Layout of Data Structure when entering ENTT

ENTB - enter subroutine with buddy allocation

Assembly notation	Hex code	Octal code
ENTB <log size/r/BY>	OBDH	275B

Description:

A local data area of size 2**<log size> 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 section 3.3 on buddy allocation for detailed description.)

In certain combinations of ENTB and ENTS there is a danger of allocating overlapping data areas.

Trap conditions: Addressing traps, Stack Overflow, Instruction Sequence Error

Initializations performed:

```
address of heap element -> B
oldB                    -> B.PREVB
oldB.SP                 -> B.SP
return address          -> B.RETA -> L
log size                -> B.LOG
number of arguments     -> B.N
addresses of arguments  -> B.ARG
```

13.11 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   B.RETA -> P -> L    B.PREVB -> B

RETK:       1 -> STATUS.K   B.RETA -> P -> L    B.PREVB -> B

RETD:       L -> P

RETT:       The register block is loaded from B.arg2..B.arg40. OTE,
            TEMM, CED and CAS are loaded from the domain information
            table. The status register is loaded partly from
            B.arg18..B.arg19 and partly from the domain information
table

IF K RET:   If  STATUS.K = 1  then
                B.RETA -> P -> L   B.PREVB -> B
            endif

RETB:       Local data area released to heap
            0 -> STATUS.K   B.RETA -> P -> L  B.PREVB -> B

RETBK:      Local data area released to heap
            1 -> STATUS.K   B.RETA -> P -> L   B.PREVB -> B
```

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 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 subroutine using a heap element as local data area. The local data area 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: Addressing traps, Stack Underflow, Branch 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 B.PREVB or B.RETA is zero, the RET, RETK and IF K RET instructions will compare CAD from DIT of calling domain to CED. If they are equal, a stack underflow trap condition occurs. If CAD from DIT of calling domain is not equal to CED, the current domain is changed back to CAD from DIT of calling domain, and the B, P, and CAD registers are loaded from the new domain information table. The TOS, HL, LL and THA values are loaded from the new domain information table.

RETT will compare the domain number of the trapped domain (saved in the domain information table) with the number of 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.

14 STRING INSTRUCTIONS

14.1 Introduction

The string handling instructions make special use of the I1 and I2 registers as pointers in the source and destination string respectively. I2 is also used for those instructions which have two source operands, as a pointer in the second source string.

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.

The operand in the instruction is the address of a string descriptor giving the length of the string and its start address. A DESC prefix is not allowed in the operand specifier; the descriptor addressing format is implicit in string instructions. If the ALT prefix is used, the descriptor is found in the current domain. Only the byte string is found in the alternative domain. Operands that are not strings are addressed directly and maybe prefixed by DESC.

Addressing traps may be caused by the addressing of the descriptor or by the address field in the descriptor.

CHARACTER TRANSLATION

Some instructions refer to a translation table. The table is 256 contiguous bytes and a translation is a reference in this table which uses 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 translation table is addressed directly, not via an implicit descriptor. If the translation table is addressed via an explicit descriptor operand, the index register is not incremented.

DATA STATUS BITS

The data status bits Z and 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 and overflow are always cleared.

The K flag always reflects the termination condition; the previous setting of the flag is lost. If a numeric argument (for example in the SFILLN instruction) is addressed via a descriptor, the descriptor addressing will not affect the K bit.

TERMINATION CONDITIONS

Execution of an instruction may terminate for various reasons and the termination condition sets the K, Z and/or S status bits.

If the destination pointer register (I2) is incremented beyond the last element of the destination string, the termination condition is called Destination full, implying that 1 -> K. Execution termination for reasons other than destination full implies that 0 -> K.

If the source pointer register (usually I1) is incremented beyond the last element of the source string, the termination condition is called
.

Each instruction gives different statuses to the Z and S bits.

After execution, I1 and I2 remain unmodified, and point to either the next element or to the element satisfying the specified condition, depending on the termination conditions. The next element is the first one not referred to by the instruction. It is the first character beyond the end of the string if the end of the string has been reached.

Source empty or Destination full implies that I1 and I2 point to the next element. conditions that terminate as a result of the condition being satisfied and instructions with will leave the I1 and I2 registers pointing to the element causing the termination.

When more than one termination condition is reached at the same time, the instruction terminates with the first one mentioned in the termination condition list of the instruction.

ADDRESSING OUTSIDE STRINGS

If the pointer register points outside the string when the instruction starts execution, a descriptor range trap condition arises. This may occur for source strings as well as for destination strings. Addressing a string of length zero will always be outside the string.

If any string operand is addressed outside its legal range, no string elements will be examined, moved, or compared. The I1 and I2 registers are then unmodified, and a descriptor-range trap condition occurs. If a <=source=> operand or both <=source=> and <=dest=> are addressed outside the strings, the instruction will terminate with K=0. Addressing outside the <=dest=> string, but within the <=source=> string, will cause termination with K=1.

OVERLAPPING STRINGS

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 an element in the source string of the old contents 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 necessary, by operating on the string elements in the reverse order.

For instructions containing a 'while' or 'until' condition, the length cannot be 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.

NOTATIONS

Instruction descriptions use the following notation:

<=operand=>	: Implicit descriptor operand, i.e. 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'st character in source string
D(I2)	: I2'nd character in destination or source-2 string
tr(char)	: char translated via the <trans table> operand

14.2 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 strings do
   S(I1) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

String elements are moved from the <=source=> operand to the <=dest=> operand until the end of <=source=> is reached or the <=dest=> is full.

Overlap is taken care of.

Terminating conditions:

outside source:	K=0	I1, I2 unmodified, DR trap condition
outside dest:	K=1	I1, I2 unmodified, DR trap condition
source empty:	K=0	I1, I2 :- next element
dest 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
```

14.3 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 not end of strings
and S(I1) AND <mask> = <test> do
  S(I1) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes are moved from the <=source=> operand to the <=dest=> operand. When the result of a logical AND between the moved byte and the <mask> operand is equal to the value of the <test> operand, the moving continues until the <=source=> operand is empty or the <=dest=> operand is full. Overlap is not taken care of.

Terminating conditions:

outside source:	K=0	Z=0	I1, I2 unmodified, DR trap condition
outside dest:	K=1	Z=0	I1, I2 unmodified, DR trap condition
different bytes:	K=0	Z=0	I1, I2 :- differing bytes
source empty:	K=0	Z=1	I1, I2 :- next element
dest full:	K=1	Z=1	I1, I2 :- next element

Example:

Copy characters from INPUT to BUFFER as long as the characters are in the range 100B to 200B, starting at current character positions in I1 and I2

```
    BY SMVWH INPUT, BUFFER, 300B, 100B
```

14.4 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 not end of strings
and S(I1) AND <mask> >< <test> do
   S(I1) -> D(I2),  I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes are moved from the <=source=> to the <=dest=> operand until the <=source=> is empty, the <=dest=> is full or the result of a logical AND between the next byte to be moved and the value of the <mask> operand is equal to the value of the <test> operand. Overlap is not taken care of.

The byte satisfying the until-condition is not moved.

Terminating conditions:

outside source:	K=0	Z=0	I1, I2 unmodified, DR trap condition
outside dest:	K=1	Z=0	I1, I2 unmodified, DR trap condition
byte found:	K=0	Z=1	I1, I2 :- found byte in source
source empty:	K=0	Z=0	I1, I2 :- next element
dest full:	K=1	Z=0	I1, I2 :- next element

Example:

Copy characters from argument ARG on the alternative domain to the global string LINE in the current domain. An apostrophe (ASCII 47B) is interpreted as the end of the source string.

```
W1 CLR; W2 CLR
BY SMVUN ALT(IND(B.ARG)), LINE, 177B, 47B
```

14.5 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	OFD74H	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=> operand are translated via a translation table found at the address specified in the operand <trans table>. Translated bytes are moved from the <=source=> to the <=dest=> operand until the <=source=> is empty or the <=dest=> is full. Overlap is taken care of.

Terminating conditions:

outside source:	K=0	I1, I2 unmodified, DR trap condition
outside dest:	K=1	I1, I2 unmodified, DR trap condition
source empty:	K=0	I1, I2 :- next element
dest 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
```

14.6 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 not end of strings
and tr(S(I1)) >< ASCII "escape" 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=> operand are translated via the translation table found at the address specified in the <trans table> operand. Translated bytes are moved from <=source=> to <=dest=> 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 operand is empty, or the <=dest=> operand full. Overlap is not taken care of.

The "escape" character is not moved.

Terminating conditions:

outside source:	K=0	Z=0	I1, I2 unmodified, DR trap condition
outside dest:	K=1	Z=0	I1, I2 unmodified, DR trap condition
"escape" found:	K=0	Z=1	I1, I2 :- position of "escape".
source empty:	K=0	Z=0	I1, I2 :- next element
dest full:	K=1	Z=0	I1, I2 :- next element

Example:

Remove ASCII NULs 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
```

14.7 String move m elements

Format: t SMOVN <=source/r/t/I1=>,<=dest/w/t/I2=>,<m/r/W>

Assembly notation	Name	Hex code	Octal code
BI SMOVN	string move m bits	OFD76H	176566B
BY SMOVN	string move m bytes	OFD77H	176567B
H SMOVN	string move m halfwords	OFD78H	176570B
W SMOVN	string move m words	OFD79H	176571B
F SMOVN	string move m floats	OFD7AH	176572B
D SMOVN	string move m double floats	OFD7BH	176573B

Operation:

```
0 -> i
while not end of strings and i < m do
   S(I1) -> D(I2)
   I1 + 1 -> I1, I2 + 1 -> I2
   i + 1 -> i
enddo
```

Description:

M items are moved from the <=source=> to the <=dest=> operand, unless the end of the <=source=> operand is reached or the <=dest=> operand full. Overlap is taken care of.

Terminating conditions:

outside source:	K=0	Z=0	I1, I2 unmodified, DR trap condition
outside dest:	K=1	Z=0	I1, I2 unmodified, DR trap condition
m items moved:	K=0	Z=1	I1, I2 :- next element
source empty:	K=0	Z=0	I1, I2 :- next element
dest full:	K=1	Z=0	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
```

14.8 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:

```
 while not end of string do
    tn -> D(I2)
    I2 + 1 -> I2
enddo
```

Description:

The contents of the specified register are put into every element of the <=dest=> string starting at the element specified by the I2 register.

Terminating conditions:

```
outside dest: K=1   I2 unmodified, DR trap condition
string filled: K=1  I2 :- next element
```

Example:

Fill the remaining characters of STRING with ASCII spaces (40B)

```
BY3 := 40B
BY3 SFILL STRING
```

14.9 String fill m elements

Format: tn SFILLN <=dest/w/t/I2=>,<m/r/W>

Assembly notation		Name	Hex code	Octal code
BIn	SFILLN	string fill m bits	0FD94H+(n-1)	176624B+(n-1)
BYn	SFILLN	string fill m bytes	0FD98H+(n-1)	176630B+(n-1)
Hn	SFILLN	string fill m halfwords	0FD9CH+(n-1)	176634B+(n-1)
Wn	SFILLN	string fill m words	0FDA0H+(n-1)	176640B+(n-1)
Fn	SFILLN	string fill m floats	0FDA4H+(n-1)	176644B+(n-1)
Dn	SFILLN	string fill m double float	0FDA8H+(n-1)	176650B+(n-1)

Operation:

```
0 -> i
while not end of string and i < m do
   tn -> D(I2)
   I2 + 1 -> I2
   i + 1 -> i
enddo
```

Description:

If the number of elements in the <=dest=> string, starting at the element indicated by I2, is greater than m, the contents of the specified register are stored in the m first elements of the <=dest=> string, starting at element I2. Otherwise all elements of the <=dest=> string from I2 to the end are filled with the contents of the register.

m is unsigned.

Terminating conditions:

outside dest:	K=1	Z=0	I2 unmodified, DR trap condition
m elements filled:	K=0	Z=1	I2 :- next element
dest full:	K=1	Z=0	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
```

14.10 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	OFDACH	176654B

Operation:

```
while not end of strings
and S(I1) = D(I2) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes from the <=source-1=> string are compared with 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.

If both operands are addressed outside strings they will compare as "exact match". <=source-1=> addressed outside the string will compare as "<=source-1=> shorter than <=source-2=>". <=source-2=> addressed outside the string will compare as "<=source-1=> longer than <=source-2=>". In either case I1, I2 are unmodified and a descriptor range trap condition arises.

Terminating conditions:

both operands outside string:	K=0	Z=1	S=0	I1, I2 unmodified, DR trap condition
exact match:	K=0	Z=1	S=0	I1, I2 :-next element
source-1 longer:	K=0	Z=0	S=0	I1, I2 :- next element
source-2 longer:	K=0	Z=0	S=1	I1, I2 :- next element
greater byte in source-1:	K=1	Z=0	S=0	I1, I2 :- differing elements
smaller byte in source-1:	K=1	Z=0	S=1	I1, I2 :- differing elements

Example:

Scan INPUTLINE and local COMMAND from the current positions until different characters are found or end of string is reached

```
BY SCOMP INPUTLINE, B.COMMAND
```

14.11 String compare translated

Format: BY SCOTR <=source-1/r/BY/I1=>,<=source-2/r/BY/I2=>, <trans table/aa/BY>

Assembly notation	Name	Hex code	Octal code
BY SCOTR	byte string compare translated	OFDADH	176655B

Operation:

```
while not end of strings
and  tr(S(I1)) = tr(D(I2)) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Translated bytes from the <=source-1=> string are compared with 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.

If both operands are addressed outside strings they will compare as "exact match". <=source-1=> addressed outside the string will compare as "<=source-1=> shorter than <=source-2=>". <=source-2=> addressed outside the string will compare as "<=source-1=> longer than <=source-2=>". In either case I1, I2 are unmodified and a descriptor-range trap condition arises.

Terminating conditions:

both operands outside string:	K=0	Z=1	S=0	I1, I2 unmodified, DR trap condition
exact match:	K=0	Z=1	S=0	I1, I2 :- next element
source-1 longer:	K=0	Z=0	S=0	I1, I2 :- next element
source-2 longer:	K=0	Z=0	S=1	I1, I2 :- next element
greater byte in source-1:	K=1	Z=0	S=0	I1, I2 :- differring elements
smaller byte in source-1:	K=1	Z=0	S=1	I1, I2 :- differing elements

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
```

14.12 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	OFDBEH	176676B

Operation:

```
while not end of strings
and S(I1) = D(I2) do
   I1+1 -> I1,  I2+1 -> I2
enddo
```

Description:

Bytes from the <=source-1=> string are compared with the corresponding bytes in the <=source-2=> string until unequal bytes are found, or until the end of both strings has been reached. If the lengths of the <=source-1=> and <=source-2=> strings are not equal, the shorter 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.

An operand addressed outside the string is treated as consisting of pad bytes only. Two operands both addressed outside the strings will compare as "exact match". The pointer registers are unmodified. In either case a descriptor-range trap condition arises.

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:	K=1	Z=0	S=0	I1, I2 :- differing elements
smaller byte in source-1:	K=1	Z=0	S=1	I1, I2 :- differing elements

Example:

Compare argument ITEM with global TABLE, padding with ASCII spaces

```
    BY SCOPA IND(B.ITEM), TABLE, 20H
```

14.13 String compare translated with pad

Format: BY SCOPT <=source-1/r/BY/I1=>,<=source-2/r/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 not end of strings
and tr(S(I1)) = tr(D(I2)) do
   I1+1 -> I1,  I2+1 -> I2  (see note below)
enddo
```

Description:

Translated bytes from the <=source-1=> string are compared with the corresponding translated bytes in the <=source-2=> string. The comparison continues until unequal bytes are found or the ends of both strings has been reached. If the lengths of the <=source-1=> and <=source-2=> strings are unequal, the shorter 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.

An operand addressed outside the string is treated as consisting of pad bytes only. Two operands both addressed outside the strings will be compared as an "exact match". The pointer registers are unmodified. In either case, a descriptor range trap condition arises.

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.

Note: The index registers are not incremented when padding a string.

Terminating conditions:

```
exact match:   K=0  Z=1  S=0  I1, I2 :- next el. or end of string
greater byte
  in source-1: K=1  Z=0  S=0  I1, I2 :- differing elements
smaller byte
  in source-1: K=1  Z=0  S=1  I1, I2 :- differing elements
```

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
```

14.14 String skip elements

Format: BY SSKIP <=source/r/BY/I1=>,<test/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SSKIP	skip elements	OFDAEH	176656B

Operation:

```
while not end of string
and S(I1) = <test> do
   I1 + 1 -> I1
enddo
if S(I1) >> <test> then
   0 -> S
else
   1 -> S
endif
```

Description:

Bytes in the <=source=> operand are examined one by one until an examined byte is different from the <test> operand or until the end of the <=source=> operand is reached. A <=source=> operand addressed outside the string will cause immediate termination with I1 unmodified and cause a descriptor range trap condition.

The byte elements are considered to be unsigned values.

Terminating conditions:

```
outside source:  K=0   Z=1   S==   I1 unmodified, DR trap condition
byte >> <test> : K=0   Z=0   S=0   I1 :- differing element
byte << <test> : K=0   Z=0   S=1   I1 :- differing element
source empty:    K=0   Z=1   S=0   I1 :- next element
```

Example:

Skip ASCII spaces from the current character in the string described by record addressed LINE

```
BY SSKIP R.LINE, 32
```

14.15 String locate element

Format: t SLOCA <=source/r/t/I1=>,<test/r/BI,BY>

Assembly notation	Name	Hex code	Octal code
BI SLOCA	string locate bit	OFDAFH	176657B
BY SLOCA	string locate byte	OFDBOH	176660B

Operation:

```
while not end of string
and S(I1) >< <test> do
   I1 + 1 -> I1
enddo
```

Description:

The <=source=> operand is examined element by element until an examined element is equal to the <test> operand or until the end of <=source=> operand is reached.

Terminating conditions:

outside source:	K=0	Z=1	I1 unmodified, DR trap condition
element = <test>:	K=0	Z=1	I1 :- found element
source empty:	K=0	Z=0	I1 :- next 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
```

14.16 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	OFDB1H	176661B

Operation:

```
while not end of string
and tr(S(I1)) AND <mask> = zero  do
   I1 + 1 -> I1
enddo
```

Description:

The <=source=> operand is scanned until the result of a logical AND between the current translated byte and <mask> is different from zero, or until the end of <=source=> operand is reached.

Terminating conditions:

outside source:	K=0	Z=1	I1 unmodified, DR trap condition
byte AND mask><zero:	K=0	Z=0	I1 :- found element
source empty:	K=0	Z=1	I1 :- next 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)
```

14.17 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	OFDB2H	176662B

Operation:

```
while not end of string
and tr(S(I1)) AND <mask> >< zero do
   I1 + 1 -> I1
enddo
```

Description:

The <=source=> operand is examined until the result of a logical AND between the examined byte translated and the <mask> is equal to zero, or until the end of <=source=> operand is reached.

Terminating conditions:

outside source:	K=0	Z=0	I1 unmodified, DR trap condition
tr(byte) AND mask = zero:	K=0	Z=1	I1 :- found element
source empty:	K=0	Z=0	I1 :- next element

Example:

Skip the rest of a string fragment DIRECTIVE which is terminated by a character translating to zero in the local table CODETABLE

```
BY SSPAN DIRECTIVE, OFFH, B.CODETABLE
```

14.18 String match

Format: BY SMATCH <=substring/r/BY/I1=>,<=string/r/BY/I2=>

Assembly notation	Name	Hex code	Octal code
BY SMATCH	string match	OFDB3H	176663B

Operation:

```
while not end of <=string=>
and <=substring=> >< <=string=>(I2..I2 + substring.length-1) do
   I2 + 1 -> I2
enddo
if <=substring=> = <=string=>(I2..I2 + substring.length-1) then
   1 -> Z
else
   0 -> Z
endif
```

Description:

The <=string=> operand is examined until either a substring equal to <=substring=> is found or the end of <=string=> operand is reached. The I1 register is left unmodified.

A <=substring=> operand or both <=string=> and <=substring=> operands addressed outside the strings are treated as if the <=substring=> is immediately found (Z=1). A <=string=> operand addressed outside the string and a <=substring=> operand addressed within the string is treated as <=substring=> not found (Z=0). Both cases will cause a descriptor-range trap condition.

Terminating conditions:

outside substring:	K=0	Z=1	I2 unmodified, DR trap condition
outside string:	K=0	Z=0	I2 unmodified, DR trap condition
substring found:	K=0	Z=1	I2 :- first matching byte
source empty:	K=0	Z=0	I2 :- next element

Example:

Set I2 to point to the next occurence of COMMA in PARAMETERS

```
BY SMATCH COMMA, PARAMETERS
```

14.19 Set parity in string

Format: BY SSPAR <=string/rw/BY/I1=>,<mode/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SSPAR	set parity in string	0FDB4H	176664B

Operation:

```
while not end of string do
   parity according to <mode> -> bit 7 of S(I2)
   I1 + 1 -> I1
enddo
```

Description:

The parity bit (bit 7) in every byte in <=string=> is set according to the following values of the <mode> operand:

0	clear parity
1	set parity
2	even parity
3	odd parity

Any other value will cause an illegal operand value trap condition.

Terminating conditions: K=1

Example:

Set even parity in local string OUTPUT

```
    BY SSPAR B.OUTPUT, 2
```

14.20 Check parity in string

Format: BY SCHPAR <=string/r/BY/I1=>,<mode/r/BY>

Assembly notation	Name	Hex code	Octal code
BY SCHPAR	check parity in string	OFDB5H	176665B

Operation:

```
0 -> Z
while not end of string
and bit 7 of S(I1) = parity according to <mode> do
    I1 + 1 -> I1
enddo
if bit 7 of S(I1) >< parity according to <mode> then
    1 -> Z
endif
```

Description:

The parity bit (bit 7) in every byte in <=string=> is checked according to the following values of the <mode> operand:

0	clear parity
1	set parity
2	even parity
3	odd parity

Any other value will cause an illegal operand value trap condition.

Terminating conditions:

outside string:	K=0	Z=0	I1 unmodified, DR trap condition
string empty:	K=0	Z=0	I1 :- next element
parity error found:	K=0	Z=1	I1 :- element with wrong parity

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);
```

15 MISCELLANEOUS INSTRUCTIONS

15.1 Block move and Fill

Format: t BMOVE <source/r/t>,<dest/w/t>,<m/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:

```
0 -> i
while i < m do
   source(i) -> dest(i); i + 1 -> i
enddo
```

Description:

<m> elements are moved from the <source> to the <dest> operand. The operands are pointers to the start of the blocks. Overlap is taken care of. Constants and registers are illegal as destination operands. When a register or a constant is specified as a source operand, the destination string is filled with <m> elements equal to the value of the <source> operand. <m> is unsigned.

Trap conditions: Addressing traps

Data status bits: All cleared

Terminating conditions: m elements 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 of facilitating detection of uninitialized variables

```
W1 := 080000000H
W BMOVE W1, B.20, AREASIZE
```

15.2 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

Operation: <source> type converted from t1 to t2 -> <dest>

Description:

The <source> operand of type t1 is converted to data type t2 and the result is 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 one if the bit is set. Conversion to bit implies that the bit is set if the source is different from zero, otherwise it is cleared.

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
```

15.3 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 the R2nd element of halfword argument ROUNDEDRESULT

```
D HCONR DESC(RESULTS)(R2), IND(B.ROUNDEDRESULT)(R2)
```

15.4 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: addr(<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.

Formats other than Wn are used to give the correct scaling factor if <operand> is indexed. Fn is equivalent to Wn, but may improve readability.

Trap conditions: Addressing traps

Data status bits: address = 0 -> Z

Example:

Load the address of the R3rd element of the halfword array argument TABLE into R1

```
    H1 LADDR B.TABLE(R3)
```

15.5 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	OFC55H	176125B
BY	RLADDR	byte load address to R	OFC5AH	176132B
H	RLADDR	halfword load address to R	OFCB1H	176261B
W	RLADDR	word load address to R	OBEH	276B
F	RLADDR	float load address to R	OBEH	276B
D	RLADDR	double float load address to R	OFCB2H	176262B

Operation: addr(<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.

Trap conditions: Addressing traps

Data status bits: address = 0 -> Z

Example:

Load R with the base address of the first stack frame below the current stack frame

```
W RLADDR IND(B.0)
```

15.6 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: addr(<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.

Trap conditions: Addressing traps

Data status bits: address = 0 -> Z

Example:

Load B with the address of argument NEWB

```
    W BLADDR B.NEWB
```

15.7 Load address of multilevel chain

Format: Wn CHAIN <address/aa/W>,<offset/r/W>,<no of levels/r/W>

Assembly notation	Name	Hex code	Octal code
Wn CHAIN	load address of multilevel chain to register	0FD6CH+(n-1)	176554B+(n-1)

Operation:

```
<address> -> Wn
for i in (1..<no of levels>) do
while ((Wn)+<offset>) >< 0
   ((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.

If the next link in the chain is zero, the operation is terminated, Wn will contain the last element in the link (pointing to a zero location) and the K flag is set. This will also cause an illegal operand value trap condition.

A negative <no of levels> will cause an illegal operand value trap condition. <no of levels> equal to zero will have the same effect as a LADDR instruction.

Trap conditions: Addressing traps, Illegal Operand Value

Data status bits: Last address.signbit -> S

Example:

Load W1 with stack base address of a procedure five static levels up, the static link is found in local variable STATLINK

```
    W1 CHAIN B.STATLINK, STATLINK, 5
```

15.8 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	OFDOCH+(n-1)	176414B+(n-1)
Hn LIND	halfword load index	OFD10H+(n-1)	176420B+(n-1)
Wn LIND	word load index	OACH+(n-1)	254B+(n-1)
Fn LIND	floating load index	OFFC8H+(n-1)	177710B+(n-1)
Dn LIND	double floating load index	OFFCCH+(n-1)	177714B+(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
```

15.9 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)
Fn CIND	floating calculate index	0FFD0+(n-1)	177720B+(n-1)
Dn CIND	double float. calcul. index	0FFD4+(n-1)	177724B+(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 multi-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 <lower> operand or greater than the <upper> operand, the flag bit (K) 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 W1 with the address of ARR(IX1,IX2,IX3), where the indexes are local halfword variables

```
H1 CIND IX1, 1, 3
H1 CIND IX2, 5, 10
H1 CIND IX3, 2, 9
```

15.10 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
```

15.11 Set flag

Format: SETK

Assembly notation	Name	Hex code	Octal code
SETK	set flag	OFE02H	177002B

Operation: 1 -> K bit of status register

Description:

Set the flag bit of the status register

Trap conditions: None

Data status bits: Unaffected

Example:

```
SETK
```

15.12 Clear flag

Format: CLRK

Assembly notation	Name	Hex code	Octal code
CLRK	clear flag	OFE03H	177003B

Operation: 0 -> K bit of status register

Description:

Clear the flag bit of the status register

Trap conditions: None

Data status bits: Unaffected

Example:

```
    CLRK
```

15.13 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:

Allocate 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 to the specified register. Otherwise the list is examined for larger elements. If none are available, a stack overflow trap condition occurs. 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 added to the appropriate freelists.

The administration of the heap is described in section 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 W3

```
    W3 GETB 6
```

15.14 Free buddy element

Format: FREEB <log size/r/BY>,<element/s/W>

Assembly notation	Name	Hex code	Octal code
FREEB	free buddy	OFDB6H	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 section 3.3. When executing the FREEB instruction, the TOS register must point to the variables describing the heap.

Write access to the <element> is required, but if <element> is addressed with a DESC prefix, the index register is not updated.

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)
```

15.15 PLCCN - Convert PLANC descriptor to ND-500 descriptor ('87 extension)

Format: W PLCCN <source/r/W>,<destination/w/W>

Assembly notation	Name	Hex code	Octal code
W PLCCN	convert to ND-500 descriptor	FFFDH	177775B

Operation:

```
(u-l+1) -> N
a + 1 -> A
```

Description:

A PLANC descriptor is converted to an ND-500 descriptor.

The descriptors are as shown below:

Planc descriptor

address	(a)
lower	(l)
upper	(u)

ND-500 descriptor

Number of elements	(N)
Address	(A)

Data Status Bits:

```
Number of elements = 0  -> Z
Signbit                 -> S
```

15.16 NCPLC - Convert ND-500 descriptor to PLANC descriptor ('87 extension)

Format: W NCPLC <source/r/W>,<destination/w/W>

Assembly notation	Name	Hex code	Octal code
W NCPLC	convert to PLANC descriptor	FFFEH	177776B

Operation:

```
A -> a
0 -> l
N - 1 -> u
If u-l+1 < 0, 0 -> N
```

Description:

Convert ND-500 descriptor to planc descriptor.

The descriptors are as shown below:

ND-500 descriptor

Number of elements	(N)
Address	(A)

Planc descriptor

address	(a)
lower	(l)
upper	(u)

Data Status Bits:

Upper = 0 -> Z
Signbit -> S

15.17 CLINIT - Initialize local clock ('87 extension)

Assembly notation	Name	Hex code	Octal code
CLINIT	initalize CPU's clock	FF1EH	177436B

Operation: 0 -> <clock>

Description:

Privileged instruction

The CPU contains a local clock running at 1 microsecond cycle time.

Clock is reset and started.

Trap Conditions: None

Data Status Bits: Unaffected

15.18 CLREAD - Read local clock ('87 extension)

Assembly notation	Name	Hex code	Octal code
CLREAD	read CPU's clock	FF1FH	177437B

Operation: <clock> -> W1

Description:

The clock value is read into register number 1. Time is an integer value giving the number of microseconds since the last CLINIT instruction.

Note that the clock counts for a periode of 2**32 microseconds after which it starts from zero again.

Trap Conditions: None

```
Data Status Bits: <clock> = 0       -> Z
                  <clock>.signbit   -> S
```

16 SPECIAL INSTRUCTIONS

16.1 Disable process switch

Format: SOLO

Assembly notation	Name	Hex code	Octal code
SOLO	disable process switch	OFEOOH	177000B

Operation: disables process switch for maximum 256 micro-cycles

Description:

Ensure that instructions up to the next TUTTI instruction are 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 microcycle per operand specifier.

No enabled trap conditions may occur when the process switch is disabled, as any trap handling will take more than 256 micro-cycles and cause timeout. Non-ignorable and fatal traps cause a disable process switch error trap.

In privilege mode there is no limitation to the duration of a SOLO operation. Unprivileged users are not allowed to run in SOLO for more than 256 cycles. In the 500/2 implementation, these are microcycles. In the ND-5000 implementation they are macroinstruction cycles.

Disable process switch timeout occurs if unprivileged users attempt to repeat SOLO's.

Trap conditions: Disable process switch Timeout, Disable process switch Error

Data status bits: Unaffected

Example:

```
SOLO
```

16.2 Enable process switch

Format: TUTTI

Assembly notation	Name	Hex code	Octal code
TUTTI	enable process switch	0FE01H	177001B

Operation: process switch is enabled

Description:

The complement of SOLO; allows normal interleaving of process execution in the system.

Trap conditions: None

Data status bits: Unaffected

Example:

```
    TUTTI
```

16.3 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 performs the two necessary memory accesses uninterruptible by other processors or by channels connected to the memory system. It may therefore be used to implement processor synchronization. The TSET instruction always reads the contents of main memory, even if the addressed data are present in cache memory. The cache is updated for later references by ordinary load instructions.

The TSET instruction is valid in the MPM-IV and later memory systems. In installations using MPM-III, it will work algorithmically as specified here but the memory operations are independent and other memory accesses may interfere.

Register and constant operands are illegal, and will cause an illegal operand specifier trap condition.

Trap conditions: Addressing traps, Illegal Operand Specifier

Data status bits:

operand was zero before store	-> Z
operand was negative before store	-> S

Example:

Set byte variable RESERVE to all ones

```
BY4 TSET RESERVE
```

16.4 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
```

16.5 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 with 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 condition.

Trap conditions: Addressing traps, Illegal Operand Value

Data status bits: Unaffected

Example:

Enable the integer Overflow trap

```
    SETE 9
```

16.6 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 with 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:

Disable Single Instruction Trap

```
    CLTE 17
```

16.7 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 with 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. Setting status bits that are modified after each instruction is legal but meaningless, as they will be cleared before the next instruction. These include bits in the range 17 to 25, 27 and 28.

Example:

Restore the TOS register from the current top of stack after a call to a routine entered through ENTM

```
TOS:= B.SP
```

16.8 Store special register

Format: special register =: <operand/w/W>

Assembly notation	Name	Hex code	Octal code
L=:	store link register	0FDC0H	176700B
HL=:	store high limit register	0FDC1H	176701B
LL=:	store low 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
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 contents of a special register into a specified operand.

When storing the program counter (P=:), the contents of the operand will be the address of the P=: instruction.

Trap conditions: Addressing traps, illegal operand specifier

Data status bits:

```
special register = 0     -> Z
special register.signbit -> S
```

The instruction ST1=: does not affect the data status bits.

16.9 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:=	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

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
```

Example:

Store least significant part of D3 in local variable LEAST

```
E3 =: B.LEAST
```

16.10 Data cache clear

Format: DCC

Assembly notation	Name	Hex code	Octal code
DCC	data cache clear	0FF15H	177425B

Operation: Dump dirty

Description:

Data in the data cache are marked as invalid. Data marked dirty is dumped to memory. The data cache should be cleared after a DMA transfer has been performed to ensure that the cache contents are consistent with main memory contents.

If no cache is present, the instruction has no effect.

Trap conditions: None

Data status bits: Unaffected

Example:

```
DCC
```

16.11 DDIRT - Dump dirty ('87 extension)

Assembly notation	Name	Hex code	Octal code
DDIRT	dump dirty	FFFAH	177772B

Operation: Dump dirty

Description:

Data marked dirty in the data cache is written to the memory.

If no cache is present, the instruction has no effect.

Trap Conditions: None

Data Status Bits: Unaffected

Example:

```
      DDIRT
```

16.12 Program cache clear

Format: PCC

Assembly notation	Name	Hex code	Octal code
PCC	program cache clear	OFF14H	177424B

Operation: Clear program cache

Description:

Data in the program cache are marked as invalid.

If no cache is present, the instruction has no effect.

Trap conditions: None

Data status bits: Unaffected

Example:

```
PCC
```

16.13 Data memory management on

Format: DMON

Assembly notation	Name	Hex code	Octal code
DMON	data memory management on	0FF16H	177426B

Operation: turn on data memory management system

Description:

Privileged instruction.

Following data accesses will be mapped on a physical segment through the memory management system, rather than being interpreted directly as physical addresses.

If the data memory management system is already turned on, the instruction has no effect.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

Example:

```
DMON
```

16.14 Program memory management on

Format: PMON

Assembly notation	Name	Hex code	Octal code
PMON	program memory management on	OFF17H	177427B

Operation: turn on program memory management system
L -> P

Description:

Privileged instruction.

Following instruction accesses will be mapped on a physical segment through the memory management system, rather than being interpreted directly as physical addresses.

The virtual address of the next instruction to be executed is found in the L register.

If the program memory management system is already turned on, control is transferred to the instruction pointed to by the L register and the instruction has no further effect.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

Example:

```
PMON
```

16.15 Data memory management off

Format: DMOF

Assembly notation	Name	Hex code	Octal code
DMOF	data memory management off	OFF18H	177430B

Operation: turn off data memory management system

Description:

Privileged instruction.

Following data accesses will be interpreted directly as physical addresses, rather than being mapped on a physical segment through the memory management system.

If the memory management system is already turned off, the instruction has no effect.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

Example:

```
DMOF
```

16.16 Program memory management off

Format: PMOF

Assembly notation	Name	Hex code	Octal code
PMOF	program memory management off	OFF19H	177431B

Operation: turn off program memory management system
L -> P

Description:

Privileged instruction.

Following instruction accesses will be interpreted directly as physical addresses, rather than being mapped on a physical segment through the memory management system.

The physical address of the next instruction to be executed is found in the L register.

If the program memory management system is already turned off, control is transferred to the physical address specified by the L register and the instruction has no further effect.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

Example:

```
PMOF
```

16.17 Read Written In Page table

Format: tn RWIP <bit or group no./r/W>

Assembly notation	Name	Hex code	Octal code
BIn RWIP	read WIP bit	0FE94H+(n-1)	177224B+(n-1)
Hn RWIP	read WIP group	0FE98H+(n-1)	177230B+(n-1)

Operation: specified WIP bit or group -> Rn

Description:

Privileged instruction.

A bit or 16 bit group is read from the Written In Page table into the specified register. The operand specifies the physical memory page number (BIn RWIP) or physical page number/16 (Hn RWIP).

A bit set in this table indicates that the page has been written into and must be written back to disk before being replaced with another one. The bit is automatically set by hardware and is used by the swapper routines.

In hardware there are separate WIP tables for program and data. RWIP will return a logical OR of the two tables, making them appear as one. Consequently, an ND-500 system cannot have physically separate memory for program and data at the same physical addresses.

This instruction is installation dependent; using it requires knowledge of the physical memory configuration. Only the lower 25 bits of the bit number are significant. Reading bits representing non-existing memory will give a zero result.

Trap conditions: Addressing traps,Illegal Instruction Code

Data status bits:

bit or bit group = 0 -> Z

16.18 Clear Written In Page bit

Format: BI ZWIP <bit no./r/W>

Assembly notation	Name	Hex code	Octal code
BI ZWIP	clear WIP bit	0FE9CH	177234B

Operation: 0 -> specified WIP bit

Description:

Privileged instruction.

The specified bit in the Written In Page table is cleared. This instruction is used by the swapper routines after a new page has been read from disk into physical memory.

In hardware there are separate WIP tables for program and data. ZWIP will clear both tables. Consequently, an ND-500 system cannot have physically separate memory for program and data at the same physical addresses.

This instruction is installation dependent; using it requires knowledge of the physical memory configuration.

Trap conditions: Illegal Instruction Code, Illegal Operand Value

Data status bits: Unaffected

16.19 Clear Written In Page table

Format: CWIP

Assembly notation	Name	Hex code	Octal code
CWIP	clear WIP table	0FF1BH	177433B

Operation: 0 -> entire WIP table

Description:

Privileged instruction.

The entire written in page table is cleared. This instruction is used by the swapper routines.

This instruction is installation dependent; using it requires knowledge of the physical memory configuration.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

16.20 Read Page Used table

Format: tn RPGU <bit or group no./r/W>

Assembly notation	Name	Hex code	Octal code
BIn RPGU	read PGU bit	0FE88H+(n-1)	177210B+(n-1)
Hn RPGU	read PGU group	0FE8CH+(n-1)	177214B+(n-1)

Operation: specified PGU bit or group -> Rn

Description:

Privileged instruction.

A bit or 16-bit group is read from the Page Used table into the specified register. The operand specifies the physical memory page number (BIn RPGU) or physical page number/16 (Hn RPGU).

A bit set in this table indicates that the page has been used in some instruction since the last time the bit was cleared. The bit is automatically set by hardware, and is used by the swapping routines.

In hardware there are separate PGU tables for program and data. RPGU will return a logical OR of the two tables, making them appear as one. Consequently, an ND-500 system cannot have physically separate memory for program and data at the same physical addresses.

This instruction is installation dependent; using it requires knowledge of the physical memory configuration. Only the lower 25 bits of the bit number are significant. Reading bits representing non-existing memory will give a zero result.

Trap conditions: Illegal Instruction Code, Illegal Operand Value

Data status bits:

bit or bit group = 0 -> Z

16.21 Clear Page Used bit

Format: BI ZPGU <bit no./r/W>

Assembly notation	Name	Hex code	Octal code
BI ZPGU	clear PGU bit	0FE90H	177220B

Operation: 0 -> specified PGU bit

Description:

Privileged instruction.

The specified bit in the page used table is cleared. This instruction is used by the swapper routines after a new page has been read from disk into physical memory.

In hardware there are separate PGU tables for program and data. ZPGU will clear the specified bit in both tables. Consequently, an ND-500 system cannot have physically separate memory for program and data at the same physical address.

This instruction is installation dependent; using it requires knowledge of the physical memory configuration.

Trap conditions: Illegal Instruction Code, Illegal Operand Value

Data status bits: Unaffected

16.22 Clear Page Used table

Format: CPGU

Assembly notation	Name	Hex code	Octal code
CPGU	clear PGU table	OFF1AH	177432B

Operation: 0 -> entire PGU table

Description:

Privileged instruction.

The entire page used table is cleared. This instruction is used by the swapper routines.

This instruction is installation dependent; using it requires knowledge of the physical memory configuration.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

16.23 Read I/O processor memory

Format: H RIOM <ND-100 addr/r/W>,<buffer/w/H>,<no of halfwords>

Assembly notation	Name	Hex code	Octal code
H RIOM	read I/O processor memory	0FE76H	177166B

Operation: I/O processor memory -> ND-500 memory

Description:

Privileged instruction.

The I/O processor (ND-100) memory contents are copied to the ND-500 memory buffer through the ND-500 interface. The <ND-100 addr> specifies the physical ND-100 address and is usually private ND-100 memory, not directly addressable by the ND-500 . <buffer> is a logical ND-500 address.

The ND-100 memory is accessed by DMA, and does not interrupt the ND-100 program execution.

Trap conditions: Addressing traps,Illegal Instruction Code, Illegal Operand Value

Data status bits: Unaffected

Example:

Copy one page (1024 halfwords) from ND-100 address 66000B to array PG

```
     H  RIOM  66000B:W, PG, 1024
```

16.24 Clear translation speedup buffer

Format: PCTSB
DCTSB

Assembly notation	Name	Hex code	Octal code
PCTSB	clear prog translation speedup buffer	OFF1CH	177434B
DCTSB	clear data translation speedup buffer	OFF1DH	177435B

Operation: 0 -> translation speedup buffer

Description:

Privileged instruction.

The entire program or data translation speedup buffer is cleared, forcing the following accesses to reinitialize the buffer from the capability table, segment table and page index table.

Trap conditions: Illegal Instruction Code

Data status bits: Unaffected

Example:

```
DCTSB
```

16.25 Load bypassing cache

Format: tn RDUS <source/r/t>

Assembly notation	Name	Hex code	Octal code
BIn RDUS	load bit, bypass cache	OFEAOH+(n-1)	177240B+(n-1)
BYn RDUS	load byte, bypass cache	OFEA4H+(n-1)	177244B+(n-1)
Hn RDUS	load halfword, bypass cache	OFEA8H+(n-1)	177250B+(n-1)
Wn RDUS	load word, bypass cache	OFEACH+(n-1)	177254B+(n-1)

Operation: <source> -> Rn

Description:

The operand is loaded from main memory, disregarding cache contents. This is primarily useful after a DMA transfer to memory has been performed to prevent use of obsolete data in the cache. Register and constant operands are illegal and will cause an illegal operand specifier trap condition.

If the shared segment bit in the capability table is set, the cache will under no circumstances be used for accesses to that segment. Thus in multiprocess applications it is usually unnecessary to use the RDUS instruction to ensure data consistency; the ordinary load (:=) will have the same effect.

The addressed data are also loaded into the cache for later references. If no cache is present, RDUS is equivalent to :=.

Trap conditions: Addressing traps, Illegal Operand Specifier

Data status bits:

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

Example:

Read the field STAT in the record pointed to by the R register into W3, not using the cache

```
W3  RDUS R.STAT
```

16.26 OPERATING SYSTEMS SUPPORT INSTRUCTIONS

The following instructions, described on page 303 to page 322, are for running low level operating systems tasks. These tasks, known as NUCLEUS, support communication between processors in a machine (intramachine communication) and between different machines (intermachine communications).

16.26.1 RHOLE - read from NUCLEUS Hole ('87 extension)

Format: BY RHOLE <=hole/r/by/I1=>,<=string/w/by/I2=>

Assembly notation	Name	Hex code	Octal code
BY RHOLE	Read hole	FE9EH	177236B

Operation:

```
while not end of strings do
    S(I1) -> D(I2),I1+1 -> I1, I2+1 -> I2
enddo
```

Description:

Bytes are moved from source hole to destination string until either source is empty or until destination is full.

String descriptor :

Length of source string
Start address of string

Hole descriptor :

Hole number
Reserved

Trap Conditions:

No access to hole	: PV trap. Nothing moved, registers unchanged.
The hole is not a message	: IOV trap. Nothing moved, registers unchanged.
Outside source or destination	: Descriptor Range Trap.

Data Status Bits:

Outside source	: K = 0, I1, I2 Unchanged, DR trap condition.
Outside destination	: K = 1, I1, I2 Unchanged, DR trap condition.
Source empty	: K = 0, I1, I2 next element.
Destinaion full	: K = 1, I1, I2 next element.

16.26.2 WHOLE - write to NUCLEUS hole ('87 extension)

Format: BY WHOLE <=string/r/by/I1=>,<=hole/w/by/I2=>

Assembly notation	Name	Hex code	Octal code
BY WHOLE	Write hole	FE9DH	177235B

Operation:

```
while not end of strings do
    S(I1) -> D(I2),I1+1 -> I1, I2+1 -> I2
enddo
```

Description:

Bytes are moved from source string to destination hole until either source is empty or until destination is full.

String descriptor :

Length of source string
Start address of string

Hole descriptor :

Hole number
Reserved

Trap Conditions:

No access to hole	: PV trap. Nothing moved, registers unchanged.
The hole is not a message	: IOV trap. Nothing moved, registers unchanged.
Outside source or destination	: Descriptor Range Trap.

Data Status Bits:

Outside source	: K = 0, I1, I2 Unchanged, DR trap condition.
Outside destination	: K = 1, I1, I2 Unchanged, DR trap condition.
Source empty	: K = 0, I1, I2 next element.
Destinaion full	: K = 1, I1, I2 next element.

16.26.3 SEND - Send to port ('87 extension)

Format: W1 SEND <hole number/r/W>

Assembly notation	Name	Hex code	Octal code
W1 SEND	send to port	B6H	266B

Operation: I1 -> <hole numer>

Description:

Message of register 1 is sent to hole number as specified by the operand.

Trap Conditions: Protect violation, Illegal operand specifier

Data Status Bits: Unaffected

16.26.4 RECVE - Receive from port ('87 extension)

Format: W1 RECVE <hole number/r/W>,<number of bytes/w/W>

Assembly notation	Name	Hex code	Octal code
W1 RECVE	receive from port	B7H	267B

Operation: <hole number> -> I1,
length of message -> <number of bytes>

Description:

Receive message from hole number. Message is returned in register 1. Size of message is returned in 'number of bytes'.

Trap Conditions: Protect violation, Illegal operand specifier

Data Status Bits: Unaffected

16.27 INSTRUCTIONS MANIPULATING REGISTER- AND CONTEXT BLOCK

Formats:

```
        SREGBL <mask/r/W>,<address/r/W>

        LREGBL <mask/r/W>,<address/r/W>

        SCNTXT <mask/r/W>,<address/r/W>

        LCNTXT <mask/r/W>,<address/r/W>,<process number/r/W>
```

Operation:

Load and store registers and context information indicated by 'mask' into addresses given by register number and offset address.

Description:

Register block layout used in store and load register block is the same as used in store and load context, as shown in chapter 2. Register number*4 gives displacement relative to the start of the save area (Program counter is register number=0).

Address is pointer to the save and load area to be used.

Registers residing in the domain information table are modified whenever they are changed. These registers are loaded from the domain information table before execution is started. It is not neccesary to save these registers in the save area when saving the context block or the register block. Thus, the domain information table registers may be excluded from the mask.

The LCNTXT and LREGBL instructions will load registers residing in the domain information table before execution is started. If registers residing in the domain information table are included in the 'mask', these registers are loaded into the domain information table from save area. Changing domain information table, by changing PS and/or CED, will cause domain information table registers of a new domain to be loaded. The privileged instruction bit(PIA) of the status word will also be modified according to the new domain information table.

The SCNTXT and SREGBL instructions will read registers residing in the domain information table and store them in the save area if included in the mask.

When loading registers residing in the domain information table or affecting the domain information selection according to 'mask', registers are loaded from context or register block addresses while the corresponding register is updated in the domain information table. Hence this gives an opportunity to start a process with a completely new register set. Note that this will only be possible when executed as a privileged instruction.

When LREGBL is executed in non-privileged mode, it is not possible to modify the ST2, PS, CED, CAD, CTE, MTE and TEMM registers.

The CTE, MTE and TEMM registers cannot be changed by assembly instructions and since these registers do not have any corresponding hardware register, LREGBL should not attempt to modify these registers.

The LCNTXT and SCNTXT are privileged instructions, since these are using physical address when accessing the context block for load and store.

The meaning of 'mask' in REGBL and CNTXT load and store instructions are shown in the table below.

* A '1' in bit position of the 'mask' will cause register to be loaded.

Reg.	Bit.no	Reg.	Bit.no	Reg.	Bit.no	Reg.	Bit.no
P	0	A1	10	STS	20	MIC	30
L	1	A2	11	PS	21	OTE	31
B	2	A3	12	TOS	22	CTE	32
R	3	A4	13	LL	23	MTE	33
I1	4	E1	14	HL	24	TEMM	34
I2	5	E2	15	THA	25	free	35
I3	6	E3	16	CED	26	free	36
I4	7	E4	17	CAD	27	free	37

16.27.1 SREGBL - Save register block ('87 extension)

Format: SREGBL <mask/r/W>,<address/r/W>

Assembly notation	Name	Hex code	Octal code
SREGBL	save register block	FFF7H	177767B

Operation: Save register block registers in specified address according to 'mask'.

Description:

The registers specified in the mask are stored in logical memory locations addressed by <address> plus register number*4. The register numbers are shown in chapter 2.

16.27.2 LREGBL - Load register block ('87 extension)

Format: LREGBL <mask/r/W>,<address/r/W>

Assembly notation	Name	Hex code	Octal code
LREGBL	load register block	FFF6H	177766B

Operation: Load register block from logical address according to 'mask'.

Description:

The registers specified in the mask are loaded from logical memory locations addressed by <address> plus register number*4. The register numbers are shown in chapter 2.

When executed in non privileged mode, the 'mask' will be reduced to include only registers that may be modified by assembly instructions in non privileged mode.

When included in the mask, registers residing in the domain information table are loaded from the logical address to the domain information table pointed out by PS and CED as result of the LREGBL instruction.

16.27.3 SCNTXT - Save context block ('87 extension)

Format: SCNTXT <mask/r/W>,<address/r/W>

Assembly notation	Name	Hex code	Octal code
SCNTXT	save context	FFF9H	177771B

Operation: Store context block registers in specified address according to 'mask'.

Description:

Privileged instruction

Context block of current process number is saved in physical address according to 'mask'. If address = 0, context save area of the current process is used.

The registers specified in the mask are stored in locations addressed by <address> plus register number*4. The register numbers are shown in chapter 2.

When context save area is used, this is addressed by:

(process number+1)*400B + an operating system defined address.

16.27.4 LCNTXT - Load context block ('87 extension)

Format: LCNTXT <mask/r/W>,<address/r/W>,<process number/r/W>

Assembly notation	Name	Hex code	Octal code
LCNTXT	load context	FFF8H	177770B

Operation: Load context block registers from specified address according to mask.

Description:

Privileged instruction

Context block of 'process number' is loaded from <u>physical address</u> according to 'mask'. If address = 0, context save area of the current process is used. If
process number is less than 0, current process number is maintained.

The registers specified in the mask are loaded from locations addressed by <address> plus register number*4. The register numbers are shown in chapter 2.

When context block save area is used, this is addressed by:

(process number+1)*400B + an operating system defined address.

16.28 REXT - Read from device external to CPU ('87 extension)

Format: Wn REXT <device/r/W>

Assembly notation	Name	Hex code	Octal code
Wn REXT	read from external	FFE8H	177750B+n-1

Operation: <device> -> In

Description:

Privileged instruction.

Information is read from external device into the specified register. Further devices will be supported in later versions.

Device numbers:

Device = 0 : OCTO-bus / ACCP.

Data Status Bits:

```
        Nothing read      1 -> K
                      else
                          0 -> K
```

16.29 WEXT - Write to device external to CPU ('87 extension)

Format: Wn WEXT <device/r/W>

Assembly notation	Name	Hex code	Octal code
Wn WEXT	write to external device	FFECH	177754B+n-1

Operation: In -> <device>

Description:

Privileged instruction.

Information is written into external device from the specified register. Further devices will be supported in later versions.

Register 'n' is written to 'device'.

Device numbers:

Device = 0 : OCTO-bus / ACCP.

Data Status Bits:

```
        Unable to write data      1 -> K
                             else
                                  0 -> K
```

16.30 TOSSP - Special load of TOS ('87 extension)

Format: TOSSP := <operand/r/W>

Assembly notation	Name	Hex code	Octal code
TOSSP	special load of TOS	FE9F	177237B

Operation: <operand> -> TOS

Description:

The TOS register is loaded from the operand. Before the value is loaded, a check on magnitude greater than B.SP is performed. If true, a stack overflow trap condition exists.

Trap condition: Stack overflow trap

```
Data Status Bits: <operand> = 0       -> Z
                  <operand>.signbit   -> S
```

16.31 RPHS - Read from physical segment ('87 extension)

Format: RPHS <domain number/r/W>

Assembly notation	Name	Hex code	Octal code
RPHS	read from physical segment	FFF5H	177765B

Operation:

```
while I1 > 0 do
    S(I4.I3) -> D(<domain number>.I2)
    I3 + 1 -> I3
    I2 + 1 -> I2
    I1 - 1 -> I1
enddo
```

Description:

Privileged instruction

Copy a number of bytes from physical address on physical segment to logical address on the domain.

I1	: Number of bytes to be moved.
I2	: Logical address on the domain.
I3	: Address on the physical segment.
I4	: Physical segment number.
Operand	: domain number.

The copy operation is continued until the number of bytes left is equal to 0 (I1 = 0) or a page boundary is reached on the physical segment. Number of bytes to be moved is counted down and will be zero when the move operation is completed. Physical and logical addresses are incremented during the copy operation.

Data Status Bits:

no bytes left = 0	: 1 -> Z
page boundary and no bytes left < 0	: 0 -> Z

16.32 WPHS - Write to physical segment ('87 extension)

Format: WPHS <domain number/r/W>

Assembly notation	Name	Hex code	Octal code
WPHS	write to physical segment	FFF4H	177764B

Operation:

```
while I1 > 0 do
    S(<domain number>.I2) -> D(I4.I3)
    I3 + 1 -> I3
    I2 + 1 -> I2
    I1 - 1 -> I1
enddo
```

Description:

Privileged instruction

Copy number of bytes from logical address on the domain to physical address on physical segment.

I1	: Number of bytes to be moved.
I2	: Logical address on the domain.
I3	: Address on the physical segment.
I4	: Physical segment number.
Operand	: domain number.

The copy operation is continued until the number of bytes left is equal to 0 (I1 = 0) or a page boundary is reached on the physical segment. Number of bytes to be moved is counted down and will be zero when the move operation is completed. Physical and logical addresses are incremented during the copy operation.

Data Status Bits:

no bytes left = 0	: 1 -> Z
page boundary and no bytes left < 0	: 0 -> Z

16.33 CAD - load CAD ('87 extension)

Format: CAD := <operand/r/W>

Assembly notation	Name	Hex code	Octal code
CAD	load CAD	FDBAH	176672B

Operation: <operand> -> CAD

Description:

Privileged instruction

Load current alternative domain register.

Data Status Bits:

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

16.34 JUMPS - Call supervisor ('87 extension)

Format: JUMPS <address/r/W>

Assembly notation	Name	Hex code	Octal code
JUMPS	call supervisor	B9H	271B

Operation:

```
P -> context.P
B -> context.B
<address> -> P
<cpuno>   -> W1
```

Description:

Save P and B register in context block. Execution is started in <address>. The instruction implies SOLO mode.

W1 returns the ND-500/ ND-5000 CPU number.

Trap Conditions: None

16.35 SVERS - Store microprogram version ('87 extension)

Format: SVERS <destination/w/W>

Assembly notation	Name	Hex code	Octal code
SVERS	store version	FFFBH	177773B

Operation: <microprog.vers> -> <destination>

Description:

Store microprogram version to destination address.

Data Status Bits:
Status bit set according to version.

16.36 SCPUNO - Store CPU number ('87 extension)

Format: SCPUNO <destination/w/W>

Assembly notation	Name	Hex code	Octal code
SCPUNO	store CPU number	FFFCH	177774B

Operation: <CPUNO> -> <destination>

Description:

Store CPU number in destination address.

Data Status Bits:
Status bit set according to CPU number.

16.37 PHYLADR - Get physical address ('87 extension)

Format: tn PHYLADR <operand/aa/W>

Assembly notation	Name	Hex code	Octal code
tn PHYLADR	get physical address	FFF0+n-1	177760B+n-1

Operation: tr(addr(operand)) -> In

Description:

The specified index register is loaded with the logical address of operand translated to physical ND-500/ND-5000 address.

Trap Conditions:

Data Status Bits:

17 BINARY CODED DECIMAL INSTRUCTIONS (Option)

17.1 Introduction

These instructions are available only if the BCD hardware option is selected and the proper microprogram loaded.

BCD (PACKED) FORMAT

A BCD number is represented by coding each individual decimal digit using four bits, called a nibble. This significantly eases the translation to or from a printable form, ASCII characters in particular.

The digits 0 to 9 are coded by their binary equivalents:

Digit	Internal (binary) representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The codes 1010 to 1111 are invalid as digits, but are used to represent the sign. Also the code 0000 represents the sign +. The sign is placed in the rightmost nibble, following the least significant digit.

+	0000
	1010
	1100
	1110
-	1011
	1101
unsigned	1111

Arithmetic operations will return results using 1100 for plus, 1101 for minus, but all sign codes are allowed in operands. Unsigned is treated as plus.

ASCII CODED DECIMAL NUMBERS

A decimal number may also be represented using the ASCII characters. Each digit occupies one byte (8 bits). The upper four bits of the byte, called the zone, have the value 0011 unless they are used to represent the sign. The lower four bits are encoded as for BCD numbers.

Before arithmetic operations are performed on the number, it must be packed into a BCD format (PPACK instruction).

A number consists of a sequence of ASCII digits which may be preceded or followed by a sign. The sign may occupy a separate byte containing the ASCII value of + (40B or 020H, or 53B or 02BH) or - (55B or 02DH). It may also be stored in the same byte as the rightmost or leftmost digit (embedded sign representation). When the sign is embedded, the byte containing the sign has the value as follows:

positive number:	0	=>	173B	07BH
	1..9	=>	101B..111B	041H..049H
(with or without parity)				
negative number:	0	=>	175B	07DH
	1..9	=>	112B..122B	04AH..052H

The embedded sign format is also termed "overpunch" format.

When embedded, the sign byte is also allowed to be the ASCII digits alone. The sign is then positive. The ordinary digit values are also valid as embedded sign with + sign.

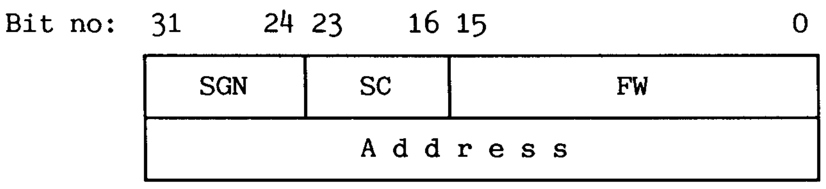
The five possible sign representations are

- **embedded trailing**, the rightmost byte contains the sign and the least significant digit
- **separate trailing**, the sign is represented by its ASCII code in a separate byte to the right of the least significant digit
- **embedded leading**, the leftmost byte contains the sign and the most significant digit
- **separate leading**, the sign is represented by its ASCII code in a separate byte to the left of the most significant digit
- **unsigned**

DESCRIPTOR FORMAT FOR ASCII AND BCD

A decimal number is addressed indirectly via a two word descriptor giving the sign representation, scaling factor, number of digits of the operand and the address of its first byte. Descriptor addressing is implicit in the BCD instructions.

The descriptor consists of two words (64 bits) with the following layout:



SGN: Sign representation of ASCII coded decimal:

bit 26	25	24	Sign representation:
0	0	0	embedded trailing
0	0	1	separate trailing
0	1	0	embedded leading
0	1	1	separate leading
1	0	0	unsigned

For BCD format the unsigned bit in the BCD descriptor is only valid for destination operands. Sign codes different from unsigned in the source operands are legal and effective even if the unsigned bit in the descriptor is set. The destination field will always be generated with the binary value 1111 in the sign nibble when the destination descriptor unsigned bit is set.

For ASCII operands, the unsigned bit in the descriptor is effective for all operands. If a sign code is detected in a source operand and the source descriptor unsigned bit is set, it is an condition. Destination operands are always generated in unsigned format when the unsigned bit in the descriptor is set.

SC: Scaling factor, specifying the position of the decimal point. Legal range is from -32 through +31. Negative values are represented as a two's complement byte. SC=0 indicates that the decimal point is immediately to the right of the least significant digit; SC>0 indicates that the decimal point is to the left of the least significant digit (the SC rightmost digits are the fractional part); SC<0 indicates that the decimal point is to the right of the least significant digit (the number has SC non-represented zeros to the right).

FW: Field width, range 0 through 31; the number of nibbles (BCD packed) or bytes (ASCII) used to represent the number, including the sign. An unsigned ASCII number with embedded sign may be up to 31 digits, a BCD packed or ASCII number with separate sign may be up to 30 digits.

EMPTY OPERANDS

A field width of zero will cause a descriptor-range trap condition. The address is not checked; no addressing traps will occur from the address part of the descriptor.

DECIMAL OPERAND ADDRESSING

Decimal operands are never loaded into registers; both descriptors and numeric fields are always found in memory. The address field in the descriptor gives the address of the leftmost byte of the numeric field. For BCD (packed) operands the numeric field is right justified in (FW+1)/2 bytes; if the field width FW is odd the leftmost nibble in the leftmost byte is not significant. The operands of an instruction may have different scaling factors and field widths. The decimal points of the operand values are automatically aligned before the operation is executed. The result value is scaled according to the scale factor in the destination descriptor.

Descriptor addressing is implicit; a DESC prefix is not allowed in the operand specifier.

OPERAND OVERLAP

An operand may be used both as source and as destination, and is described by one descriptor or by two different descriptors with equal address fields.

ROUNDING

If the instruction specifies rounding the result value may be rounded before storing in the destination operand. If the result has one or more digits to the right of the least significant digit in the destination, the leftmost digit not stored is inspected. If this digit is 5, 6, 7, 8 or 9 the least significant digit actually stored is incremented by 1. Otherwise, the digits that are not stored are ignored.

If rounding is not specified in the instruction, digits to the right of the least significant digit represented will not affect the result.

STATUS BITS

Decimal instructions will affect BCD overflow, the invalid operation value, K flag, zero and sign bits. BCD overflow and the invalid operation may be taken care of by a trap handler.

BCD overflow occurs if the destination field is too narrow to hold the result value after rounding.

An invalid operation occurs if a code representing anything other than a digit is encountered in a digit position, or anything other than a sign code is encountered in the sign position. The numeric string is checked for illegal codes in all instructions.

The packed to binary conversion instruction may also cause integer overflow.

Data status bits (Zero, Sign) are set or reset after rounding (if specified), and after the result value has been scaled according to the destination descriptor.

The K flag is set upon BCD overflow or invalid operation, otherwise the flag is cleared.

NEGATIVE AND POSITIVE ZERO

A result value of zero from an instruction will usually have a positive sign code, or unsigned if so specified in the descriptor. Source operands of value zero may have positive or negative sign; negative zero is equivalent to positive zero and will compare as equal in the PCOMP instruction.

If significant digits are lost due to a BCD overflow, the result value will have the sign of what the correct result would have had. This may give a result value of negative zero. The Z and S bits in the status register are set the same as for a positive zero value (Z=1, S=0).

BCD OVERFLOW

On BCD overflow, the result is replaced by the correctly signed least significant digits.

Restriction on Scaling Difference in Packed Add

For add, subtract, and compare the following must hold:

```
   -32≤((operand1.field width+1)/2*2-(operand1.scaling factor)-
         ((operand2.field width+1)/2*2-(operand2.scaling)))≤32
```

otherwise it is an invalid trap condition.

17.2 Packed add

Format: PADD <=a/r/BCD=>, <=b/r/BCD=>, <=c/w/BCD=>

Assembly notation	Name	Hex code	Octal code
PADD	packed add	0FEB0H	177260B
PADDR	packed add rounded	0FE85H	177205B

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

Description:

The <a> operand is added to the <b> operand and the sum is stored in the <c> operand.

The result is scaled according to the scale factor in the <c> operand before storing.

Trap conditions: Addressing traps, BCD Overflow, InValid Operation

Data status bits:

```
sum = 0       ->  Z
sum.signbit   ->  S
BCD overflow  ->  BO
BO or IVO     ->  K
```

Example:

Add local variables PRICE and TAX to form global value TOTAL

```
PADD B.PRICE, B.TAX, TOTAL
```

17.3 Packed subtract

Format: PSUB <=a/r/BCD=>, <=b/r/BCD=>, <=c/w/BCD=>

Assembly notation	Name	Hex code	Octal code
PSUB	packed subtract	0FEB1H	177261B
PSUBR	packed subtract rounded	0FE86H	177206B

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

Description:

The <b> operand is subtracted from the <a> operand and the difference is stored in the <c> operand.

The result is scaled according to the scale factor in the <c> operand descriptor before storing.

Trap conditions: Addressing traps, BCD Overflow, InValid Operation

Data status bits:

difference = 0	->	Z
difference.signbit	->	S
BCD overflow	->	BO
BO or IVO	->	K

Example:

Subtract local variable DISCOUNT from global variable TOTAL and round the resulting value before storing it

```
    PSUBR TOTAL, B.DISCOUNT, TOTAL
```

17.4 Packed multiply

Format: PMPY <=a/r/BCD=>, <=b/r/BCD=>, <=c/w/BCD=>

Assembly notation	Name	Hex code	Octal code
PMPY	packed multiply	0FEB4H	177264B
PMPYR	packed multiply rounded	0FE91H	177221B

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

Description:

The <a> operand is multiplied by the <b> operand and the product is stored in the <c> operand.

The result is scaled according to the scale factor in the <c> operand descriptor before storing.

For PMPY/PMPYR, an operand with invalid digit * ZRO gives the result 0, not IVO.

Trap conditions: Addressing traps, BCD Overflow, InValid Operation

Data status bits:

product = 0	->	Z
product.signbit	->	C
BCD overflow	->	BO
BO or IVO	->	K

Example:

Multiply local variable PRICE with DISCOUNT giving local NET. Round the resulting value before storing it

```
    PMPYR B.PRICE, DISCOUNT, B.NET
```

17.5 Packed compare

Format: PCOMP <=a/r/BCD=>, <=b/r/BCD=>

Assembly notation	Name	Hex code	Octal code
PCOMP	packed compare	OFEB3H	177263B

Operation: <a> - <b>

Description:

The <b> operand is subtracted from the <a> operand and the status bits are set according to the result. The result is discarded.

Before the comparison is performed, the operands are automatically shifted to the same decimal point position (scale) and extended with zeros if necessary. An unsigned number is treated as positive, and positive and negative zero are equal.

Trap conditions: Addressing traps, InValid Operation

Data status bits:

```
difference = 0       -> Z
difference.signbit   -> S
IVO                  -> K
```

Example:

Compare TOTAL with MAX and set status bits

```
    PCOMP TOTAL, MAX
```

17.6 Packed shift

Format: PSHIFT <=source/r/BCD=>, <=dest/w/BCD=>

Assembly notation	Name	Hex code	Octal code
PSHIFT	packed shift	OFEB2H	177262B
PSHIFTR	packed shift rounded	OFE87H	177207B

Operation: <source> -> <dest>

Description:

The content of the <source> operand is shifted to the scaling factor of the <dest> operand and, if specified, rounded before storing it in the <dest> operand. The destination string is extended with zeroes if necessary.

With the exception of rounding, the value is not modified, but the number of decimal positions may be changed. If the <source> and <dest> operands have the same scaling factor, a move is performed.

If bit 26 in the descriptor of the <dest> operand is set, the value is stored with a sign code equal to 1111 (unsigned). Otherwise, <dest> will be given the sign of the <source> value.

Trap conditions: Addressing traps, BCD Overflow, InValid Operation

Data status bits:

value after rounding = 0	->	Z
value.signbit	->	S
BCD overflow	->	BO
BO or IVO	->	K

Example:

Copy SUBTOTAL to TOTAL

```
    PSHIFT SUBTOTAL, TOTAL
```

17.7 Convert ASCII to packed

Format: PPACK <=source/r/ASCII=>, <=dest/w/BCD=>

Assembly notation	Name	Hex code	Octal code
PPACK	convert ASCII to packed	0FEB5H	177265B
PPACKR	convert ASCII to packed rounded	0FE92H	177222B

Operation: <source> -> <dest>

Description:

The content of the <source> operand in ASCII coded decimal is packed into the <dest> operand in packed format. If specified, the value is rounded before storing it in the <dest> operand.

If bit 26 in the descriptor of the <dest> operand is set, the value is stored with a sign code equal to 1111 (unsigned). Otherwise, <dest> will be given the sign of the <source> value. The <source> value consists of ASCII digits and a sign according local variables PRICE and TAX to form global value TOTAL;

????the SGN code in the <source> descriptor only.

Trap conditions: Addressing traps, BCD Overflow, InValid Operation

Data status bits:

value after rounding = 0	->	Z
value.signbit	->	S
BCD overflow	->	BO
BO or IVO	->	K

Example:

Convert ASCII value IFIELD to packed VAR1

```
PPACK IFIELD, VAR1
```

17.8 Convert packed to ASCII

Format: PUPACK <=source/r/BCD=>, <=dest/w/ASCII=>

Assembly notation	Name	Hex code	Octal code
PUPACK	convert packed to ASCII	0FEB6H	177266B
PUPACKR	convert packed to ASCII rounded	0FE93H	177223B

Operation: <source> -> <dest>

Description:

The content of the <source> operand in packed decimal format is unpacked into the <dest> operand in ASCII format. If specified, the value is rounded before storing it in the <dest> operand. The sign representation is determined by the SGN field in the <dest> descriptor.

The <dest> string is extended with leading ASCII zeros if necessary, and the parity bit for all digits will be zero.

Trap conditions: Addressing traps, BCD Overflow, InValid Operation

Data status bits:

value after rounding = 0	->	Z
value.signbit	->	S
BCD overflow	->	BO
BO or IVO	->	K

Example:

Unpack VAR1 into IFIELD and round the value according to the IFIELD descriptor

```
    PUPACKR VAR1, IFIELD
```

17.9 Convert packed to binary word

Format: Wn PWCONV <=source/r/BCD=>

Assembly notation	Name	Hex code	Octal code
Wn PWCONV	convert packed to binary	OFEBCH+(n-1)	177274B+(n-1)

Operation: <source> -> Rn

Description:

The contents of the <source> operand in packed decimal format are converted to binary format and loaded into the specified register. The fractional part of <source> is lost; no rounding is performed before the conversion.

On integer overflow the result is the least significant 32 bits of the binary result.

Trap conditions: Addressing traps, Integer Overflow, InValid Operation

Data status bits:

```
   value = 0      ->  Z
   value.signbit  ->  S
   overflow       ->  O
   IVO or O       ->  K
```

Example:

Convert IFIELD to an integer number in W1

```
    W1  PWCONV IFIELD
```

17.10 Convert binary word to packed

Format: Wn WPCONV <=dest/w/BCD=>

Assembly notation	Name	Hex code	Octal code
Wn WPCONV	convert binary to packed	0FEB8H+(n-1)	177270B+(n-1)

Operation: Rn -> <dest>

Description:

The contents of the specified word register are converted to packed decimal and stored in the <dest> operand. If the scaling factor of <dest> is negative, the least significant digits are lost. <dest> is extended with low order or high order zeros as required by the scaling factor.

Trap conditions: Addressing traps, BCD Overflow

Data status bits:

```
value = 0      ->  Z
value.signbit  ->  S
BCD overflow   ->  BO
BO             ->  K
```

Example:

Convert W1 to packed and store in IFIELD

```
W1 WPCONV IFIELD
```

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	

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:

	PREFIX
DESCRIPTOR	0F0H+
ALTERNATIVE	0C8H

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

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

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

DATA TRANSFER AND LOGICAL INSTRUCTIONS

BIn :=	load bit	page 125
BYn :=	load byte	
Hn :=	load halfword	
Wn :=	load word	
Fn :=	load float	
Dn :=	load double float	
B :=	load local base	page 126
R :=	load record base	page 127
BIn =:	store bit	page 128
BYn =:	store byte	
Hn =:	store halfword	
Wn =:	store word	
Fn =:	store float	
Dn =:	store double float	
B =:	local base store	page 129
R =:	record base store	page 130
BI MOVE	move bit	page 131
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	page 132
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	page 133
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	page 134
BY COMP2	byte compare	
H COMP2	halfword compare	
W COMP2	word compare	
F COMP2	float compare	
D COMP2	double float compare	
BI TEST	bit test against zero	page 135
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	page 136
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	page 137
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	page 139
Hn	ABS	halfword absolute value	
Wn	ABS	word absolute value	
Fn	ABS	float absolute value	
Dn	ABS	double float absolute value	
BIn	CLR	bit register clear	page 140
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	page 141
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	page 142
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	page 143
H	INCR	halfword increment	
W	INCR	word increment	
F	INCR	float increment	
D	INCR	double float increment	
BY	DECR	byte decrement	page 144
H	DECR	halfword decrement	
W	DECR	word decrement	
F	DECR	float decrement	
D	DECR	double float decrement	
BIn	AND	bit and register	page 145
BYn	AND	byte and register	
Hn	AND	halfword and register	
Wn	AND	word and register	

BIn	OR	bit or register	page 146
BYn	OR	byte or register	
Hn	OR	halfword or register	
Wn	OR	word or register	
BIn	XOR	bit exclusive or register	page 147
BYn	XOR	byte exclusive or register	
Hn	XOR	halfword exclusive or register	
Wn	XOR	word exclusive or register	
BY	SHL	byte shift logical	page 148
H	SHL	halfword shift logical	
W	SHL	word shift logical	
BY	SHA	byte shift arithmetical	page 149
H	SHA	halfword shift arithmetical	
W	SHA	word shift arithmetical	
BY	SHR	byte shift rotational	page 150
H	SHR	halfword shift rotational	
W	SHR	word shift rotational	
BYn	GETBI	byte get bit	page 151
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	page 153
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	page 155
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	REM	float divide with remainder	page 157
Dn	REM	double float divide with remainder	
Fn	INT	float integer part	page 158
Dn	INT	double float integer part	
Fn	INTR	float integer part with rounding	
Dn	INTR	double float integer part with rounding	
BYn	AMODB	byte integer modulo	page 160
Hn	AMODB	halfword integer modulo	
Wn	AMODB	word integer modulo	
F	ENTIER	float SIMULA entier function	page 161
D	ENTIER	double float SIMULA entier function	

ARITHMETICAL INSTRUCTIONS

BYn	+	byte add	page 165
Hn	+	halfword add	
Wn	+	word add	
Fn	+	floating add	
Dn	+	double float add	
BYn	-	byte subtract	page 166
Hn	-	halfword subtract	
Wn	-	word subtract	
Fn	-	float subtract	
Dn	-	double float subtract	
BYn	*	byte multiply	page 167
Hn	*	halfword multiply	
Wn	*	word multiply	
Fn	*	floating multiply	
Dn	*	double float multiply	
BYn	/	byte divide	page 168
Hn	/	halfword divide	
Wn	/	word divide	
Fn	/	float divide	
Dn	/	double float divide	
BY	ADD2	byte add two arguments	page 169
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	page 170
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	page 171
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	page 172
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	page 173
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	page 174
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	page 175
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	page 176
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	page 177
Hn	MUL4	halfword multiply with overflow	
Wn	MUL4	word multiply with overflow	
BYn	DIV4	byte divide with remainder	page 178
Hn	DIV4	halfword divide with remainder	
Wn	DIV4	word divide with remainder	
Wn	UMUL	word unsigned multiplication	page 179
Wn	UDIV	word unsigned divide	page 180
Wn	ADDC	word add with carry	page 181
Wn	SUBC	word subtract with carry	page 182
BYn	MULAD	byte multiply and add	page 183
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	page 184
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	

MATHEMATICAL FUNCTIONS

Fn	AXI	float <A> to the <I>'th power	page 187
Dn	AXI	double float <A> to the <I>'th power	
BYn	IXI	byte <I> to the <J>'th power	page 188
Hn	IXI	halfword <I> to the <J>'th power	
Wn	IXI	word <I> to the <J>'th power	
Fn	POLY	floating polynomial	page 189
Dn	POLY	double float polynomial	
Fn	SQRT	float square root	page 190
Dn	SQRT	double float square root	
Fn	SIN	float sine	page 191
Dn	SIN	double float sine	
Fn	ASIN	float arc sine	page 192
Dn	ASIN	double float arc sine	
Fn	COS	float cosine	page 193
Dn	COS	double float cosine	
Fn	ACOS	float arc cosine	page 194
Dn	ACOS	double float arc cosine	
Fn	TAN	float tangent	page 195
Dn	TAN	double float tangent	
Fn	ATAN	float arc tangent	page 196
Dn	ATAN	double float arc tangent	
Fn	ATAN2	float two argument arc tangent	page 197
Dn	ATAN2	double float two argument arc tangent	
Fn	EXP	float exponential	page 198
Dn	EXP	double float exponential	
Fn	ALOG	float natural logarithm	page 199
Dn	ALOG	double float natural logarithm	
Fn	ALOG2	float binary logarithm	page 200
Dn	ALOG2	double float binary logarithm	
Fn	ALOG10	float common logarithm	page 201
Dn	ALOG10	double float common logarithm	

CONTROL INSTRUCTIONS

GO:B	jump byte	page 205
GO:H	jump halfword	
GO:W	jump word	
JUMPG	jump general	page 206

IF = GO	Z=1	equal	page 207
IF Z GO		(alt. assembly notation)	
IF = GO:B		byte displacement	
IF = GO:H		halfword displacement	
IF >< GO	Z=0	unequal	page 207
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	page 207
IF > GO:B			
IF > GO:H			
IF < GO	S=1	less signed	page 207
IF S GO		(alt. assembly notation)	
IF < GO:B			
IF < GO:H			
IF >= GO	S=0	greater or equal signed	page 207
IF -S GO		(alt. assembly notation)	
IF >= GO:B			
IF >= GO:H			
IF <= GO	S=1 or Z=1	less or equal signed	page 207
IF <= GO:B			
IF <= GO:H			
IF K GO	K=1	flag	page 207
IF K GO:B			
IF K GO:H			
IF -K GO	K=0	not flag	page 207
IF -K GO:B			
IF -K GO:H			
IF >> GO	C=1 and Z=0	greater magnitude	page 207
IF >> GO:B			
IF >> GO:H			
IF >>= GO	C=1	greater or equal magnitude	page 207
IF C GO		(alt. assembly notation)	
IF >>= GO:B			
IF >>= GO:H			
IF << GO	C=0	less magnitude	page 207
IF -C GO		(alt. assembly notation)	
IF << GO:B			

	IF << GO:H			
	IF <<= GO	C=0 or Z=1	less or equal magnitude	page 207
	IF <<= GO:B			
	IF <<= GO:H			
	IF ST GO		specified bit in status register set	page 207
	IF ST GO:B			
	IF ST GO:H			
	IF -ST GO		specified bit in status register not set	page 207
	IF -ST GO:B			
	IF -ST GO:H			
BY	LOOPI:B		byte loop increment	page 209
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	page 211
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	page 213
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	

CALLG	call subroutine general	page 215
CALL	call subroutine absolute	page 216
INIT	initialize stack	page 217
ENTM	enter module	page 219
ENTD	enter subroutine directly	page 220
ENTS	enter stack subroutine	page 221
ENTF	enter subroutine	page 222
ENTSN	enter max argument stack subroutine	page 221
ENTFN	enter max argument subroutine	page 222
ENTT	enter trap handler	page 223
ENTB	enter buddy subroutine	page 225
RET	clear flag return from subroutine	page 226
RETK	set flag return from subroutine	page 226
RETD	return from direct subroutine	page 226
RETT	trap handler return	page 226
IF K RET	if flag set subroutine return	page 226
RETB	buddy subroutine return	page 226
RETBK	set flag buddy subroutine return	page 226

STRING INSTRUCTIONS

BI	SMOVE	bit string move	page 234
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	page 235
BY	SMVUN	byte move string until	page 236
BY	SMVTR	move translated string	page 237
BY	SMVTU	move string translated until	page 238
BI	SMOVN	string move n bits	page 239
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	page 240
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	page 241
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	page 242
BY	SCOTR	string compare translated	page 243
BY	SCOPA	string compare with pad	page 244
BY	SCOPT	string compare translated with pad	page 245
BY	SSKIP	skip elements	page 246
BI	SLOCA	string locate bit	page 247
BY	SLOCA	string locate byte	page 247
BY	SSCAN	string scan	page 248
BY	SSPAN	string span	page 249
BY	SMATCH	string match	page 250
BY	SSPAR	set parity in string	page 251
BY	SCHPAR	check parity in string	page 252

MISCELLANEOUS INSTRUCTIONS

BY	BMOVE	byte block move	page 255
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	page 256
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	page 256
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	page 256
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	page 256
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	page 256
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	page 256
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	page 258
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 with rounding	
W	FCONR	word to float convert with rounding	page 258
D	FCONR	double float to float convert with rounding	
BIn	LADDR	bit load address	page 259
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	page 260
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	page 261
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	page 262
BYn	LIND	byte load index	page 263
Hn	LIND	halfword load index	
Wn	LIND	word load index	
BYn	CIND	byte calculate index	page 264
Hn	CIND	halfword calculate index	
Wn	CIND	word calculate index	
	NOOP	no operation	page 265
	SETK	set flag	page 266
	CLRK	clear flag	page 267
Wn	GETB	get buddy	page 268
	FREEB	free buddy	page 269
W	PLCCN	convert PLANC descriptor to ND-500 descriptor	page 270
W	NCPLC	convert ND-500 descriptor to PLANC descriptor	page 271
	CLINIT	initialize local clock	page 272
	CLREAD	read local clock	page 273

SPECIAL INSTRUCTIONS

	SOLO	disable process switch	page 277
	TUTTI	enable process switch	page 278
BYn	TSET	test and set	page 279
	BP	break point instruction	page 280
	SETE	set bit in trap enable register	page 281
	CLTE	clear bit in trap enable register	page 282
	L :=	load link register	page 283
	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	page 284
	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	
	PS =:	store process segment register	
	TOS =:	store top of stack register	
	THA =:	store trap handler register	
	P =:	store program counter	
	An :=	load most sign. part of double float reg.	page 285
	En :=	load least sign. part of double float reg.	
	An =:	store most sign. part of double float reg.	
	En =:	store least sign. part of double float reg.	
	DCC	data clear cache	page 286
	DDIRT	dump dirty	page 287
	PCC	program clear cache	page 288
	DMON	data memory management on	page 289
	PMON	program memory management on	page 290
	DMOF	data memory management off	page 291
	PMOF	program memory management off	page 292
BIn	RWIP	read Written In Page bit	page 293
Hn	RWIP	read Written In Page group	
BI	ZWIP	clear Written In Page bit	page 294

	CWIP	clear Written In Page table	page 295
BIn	RPGU	read PaGe Used bit	page 296
Hn	RPGU	read PaGe Used group	
BI	ZPGU	clear PaGe Used bit	page 297
	CPGU	clear PaGe Used table	page 298
Hn	RIOM	read ND-100 memory	page 299
	PCTSB	clear program translation speedup buffer	page 300
	DCTSB	clear data translation speedup buffer	page 300
BIn	RDUS	load bit bypassing cache	page 301
BYn	RDUS	load byte bypassing cache	
Hn	RDUS	load halfword bypassing cache	
Wn	RDUS	load word bypassing cache	
BY	RHOLE	read from NUCLEUS hole	page 303
BY	WHOLE	write to NUCLEUS hole	page 304
W1	SEND	send to port	page 305
W1	RECVE	receive from port	page 306
	SREGBL	save register block	page 309
	LREGBL	load register block	page 310
	SCNTXT	save context block	page 311
	LCNTXT	load context block	page 312
Wn	REXT	read from device external to CPU	page 313
Wn	WEXT	write to device external to CPU	page 314
	TOSSP	special load of TOS	page 315
	RPHS	read from physical address	page 316
	WPHS	write to physical address	page 317
	CAD :=	load alternative domain register	page 318
	JUMPS	call supervisor	page 319
	SVERS	store version	page 320
	SCPUNO	store CPU number	page 321
tn	PHYLADR	get physical address	page 322

BCD INSTRUCTIONS (Option)

PADD	packed add	page 330
PADDR	packed add rounded	page 330
PSUB	packed subtract	page 331
PSUBR	packed subtract rounded	page 331
PMPY	packed multiply	page 332
PMPYR	packed multiply rounded	page 332
PCOMP	packed compare	page 333
PSHIFT	packed shift	page 334
PSHIFTR	packed shift rounded	page 334
PPACK	convert ASCII to packed	page 335
PPACKR	convert ASCII to packed rounded	page 335
PUPACK	convert packed to ASCII	page 336
PUPACKR	convert packed to ASCII rounded	page 336
PWCONV	convert packed to binary	page 337
WPCONV	convert binary to binary	page 338

Legal data formats	Assembly notation	Name	Page
BY H W F D	tn *	multiply	167
BY H W F D	tn +	add	165
BY H W F D	tn -	subtract	166
BY H W F D	tn /	divide	168
BI BY H W F D	tn :=	load	125
BI BY H W F D	tn =:	store	128
BY H W F D	tn ABS	absolute value	139
F D	tn ACOS	arc cosine	194
BY H W F D	t ADD2	add two arguments	169
BY H W F D	t ADD3	add three arguments	173
W	t ADDC	add with carry	181
F D	tn ALOG	natural logarithm	199
F D	tn ALOG10	common logarithm	201
F D	tn ALOG2	binary logarithm	200
BI BY H W	tn AND	AND register	145
BY H W	tn AMODB	integer modulo	160
F D	tn ASIN	arc sine	192
F D	tn ATAN	arc tangent	196
F D	tn ATAN2	arc tangent two argument	197
F D	tn AXI	register <A> to the <I>'th power	187
	An :=	load most significant part of double float reg	285
	An =:	store most significant part of double float reg	285
	B :=	load local base	126
	B =:	local base store	129
BI BY H W F D	t BLADDR	load address local	261
BY H W F D	t BMOVE	block move	255
	BP	break point instruction	280
BI H W F D	t BYCONR	convert to byte with rounding	258
BI H W F D	t BYCONV	convert to byte	256
	CAD :=	load alternative domain register	318
	CAD =:	store alternative domain register	284
	CALL	call subroutine absolute	216
	CALLG	call subroutine general	215
	CED =:	store current executing domain reg.	284
W	tn CHAIN	load address of multilevel link	262
BY H W	tn CIND	calculate index	264
BY H W	t CLEBI	clear bit	153
	CLINIT	initialize local clock	272
BI BY H W F D	tn CLR	register clear	140
	CLREAD	read local clock	273
	CLRK	clear flag	267
	CLTE	clear bit in trap enable register	282
BI BY H W F D	tn COMP	register compare	133
BI BY H W F D	t COMP2	compare	134
F D	tn COS	cosine	193
	CPGU	clear page used table	298
	CTE1 =:	store first child trap enable reg.	284
	CTE2 =:	store second child trap enable reg.	284
	CWIP	clear written in page table	295
	DCC	data cache clear	286
BI BY H W F	t DCONV	convert to double float	256
	DCTSB	clear data TSB	300

Legal data formats	Assembly notation	Name	Page
	DDIRT	dump dirty	287
BY H W F D	t DECR	decrement	144
BY H W F D	t DIV2	divide two arguments	172
BY H W F D	t DIV3	divide three arguments	176
BY H W F D	tn DIV4	divide with remainder	178
	DMOF	data memory management off	291
	DMON	data memory management on	289
	ENTB	enter buddy subroutine	225
	ENTD	enter subroutine directly	220
	ENTF	enter subroutine	222
	ENTFN	enter max argument subroutine	222
F D	t ENTIER	SIMULA entier function	161
	ENTM	enter module	219
	ENTS	enter stack subroutine	221
	ENTSN	enter max argument stack subroutine	221
	ENTT	enter trap handler	223
	En :=	load least significant part of double float register	285
	En =:	store least significant part of double float register	285
F D	tn EXP	exponential	198
W D	t FCONR	convert to float with rounding	258
BI BY H W D	t FCONV	convert to float	256
	FREEB	free buddy	269
W	t GETB	get buddy	268
BY H W	tn GETBF	get bit field	155
BY H W	tn GETBI	get bit	151
	GO:B	jump byte	205
	GO:H	jump halfword	205
	GO:W	jump word	205
F D	t HCONR	convert to halfword with rounding	258
BI BY W F D	t HCONV	convert to halfword	256
	HL :=	load upper limit register	283
	HL =:	store upper limit register	284
BY H	IF -ST GO:t	jump if status bit not set	207
BY H	IF -C GO:t	jump if magnitude less	207
BY H	IF -K GO:t	jump if flag not set	207
BY H	IF -S GO:t	jump if signed greater or equal	207
BY H	IF -Z GO:t	jump if not equal	207
BY H	IF<rel>GO:t	jump if relation true	207
BY H	IF C GO:t	jump if magnitude greater or equal	207
BY H	IF K GO:t	jump if flag set	207
BY H	IF K RET	subroutine return if flag set	207
BY H	IF S GO:t	jump if signed less	207
BY H	IF ST GO:t	jump if specified status bit set	207
BY H	IF Z GO:t	jump if equal	207
BY H W F D	t INCR	increment	143
	INIT	initialize stack	217
F D	tn INT	float integer part	158
F D	tn INTR	float integer part with rounding	159
BI BY H W	tn INV	invert register	137
W	tn INVC	word invert register with carry	138

Legal data formats	Assembly notation	Name	Page
F D	tn IXI	register I to the <J>'th power	188
	JUMPG	jump general	206
	JUMPS	call supervisor	319
	L :=	load link register	283
	L =:	store link register	284
BI BY H W F D	tn LADDR	load address	259
	LCNTXT	load context block	312
BY H W	tn LIND	load index	263
	LL :=	load lower limit register	283
	LL =:	store lower limit register	284
BY H W F D	t LOOP:B	loop general step	213
BY H W F D	t LOOP:H	loop general step	213
BY H W F D	t LOOPD:B	loop decrement	211
BY H W F D	t LOOPD:H	loop decrement	211
BY H W F D	t LOOPI:B	loop increment	209
BY H W F D	t LOOPI:H	loop increment	209
	LREGBL	load register block	310
BI BY H W F D	t MOVE	move	131
	MTE1 =:	store first mother trap enable reg.	284
	MTE2 =:	store second mother trap enable reg.	284
BY H W F D	t MUL2	multiply two arguments	171
BY H W F D	t MUL3	multiply three arguments	175
BY H W F D	tn MUL4	multiply with overflow	177
BY H W F D	tn MULAD	multiply and add	183
W	NCPLC	convert ND-500 descriptor to PLANC descriptor	271
BY H W F D	tn NEG	register negate	136
	NOOP	no operation	265
BI BY H W	tn OR	OR register	146
	OTE1 :=	load first own trap enable reg.	283
	OTE1 =:	store first own trap enable reg.	284
	OTE2 :=	load second own trap enable reg.	283
	OTE2 =:	store second own trap enable reg.	284
	P =:	store program counter	284
	PADD	packed add	330
	PADDR	packed add rounded	330
	PCC	program cache clear	288
	PCOMP	packed compare	333
	PCTSB	clear program TSB	300
	tn PHYLADR	get physical address	322
W	PLCCN	convert PLANC descriptor to ND-500 descriptor	270
	PMOF	program memory management off	292
	PMON	program memory management on	290
	PMPY	packed multiply	332
	PMPYR	packed multiply rounded	332
F D	tn POLY	polynomial	189
	PPACK	convert ASCII to packed	335
	PPACKR	convert ASCII to packed rounded	335
	PS =:	store process segment register	284
	PSHIFT	packed shift	334
	PSHIFTR	packed shift rounded	334
	PSUB	packed subtract	331
	PSUBR	packed subtract rounded	331

Legal data formats	Assembly notation	Name	Page
BY H W F D	tn PSUM	add and multiply	184
	PUPACK	convert packed to ASCII	336
	PUPACKR	convert packed to ASCII rounded	336
BY H W	tn PUTBF	put bit field	156
BY H W	tn PUTBI	put bit	152
W	tn PWCONV	convert packed to binary word	337
	R :=	load record base	127
	R =:	record base store	130
BI BY H W	tn RDUS	read bypassing cache	301
W	RECVE	receive from port	306
F D	tn REM	divide with remainder	157
	RET	clear flag return from subroutine	226
	RETB	buddy subroutine return	226
	RETBK	set flag buddy subroutine return	226
	RETD	return from direct subroutine	226
	RETK	set flag subroutine return	226
	RETT	trap handler return	226
W	REXT	read from device external to CPU	313
BY	RHOLE	read from NUCLEUS hole	303
H	t RIOM	read ND-100 memory	299
BI BY H W F D	t RLADDR	load address record	260
BI H	tn RPGU	read page used table	296
	RPHS	read from physical address	316
BI H	tn RWIP	read written in page table	293
BY	t SCHPAR	check parity in string	252
	SCNTXT	save context block	311
BY	t SCOMP	string compare	242
BY	t SCOPA	string compare with pad	244
BY	t SCOPT	string compare translated with pad	245
BY	t SCOTR	string compare translated	243
	SCPUNO	store CPU number	321
W	SEND	send to port	305
BI BY H W F D	t SET1	set to one	142
BY H W	t SETBI	set bit	154
	SETE	set bit in trap enable register	281
	SETK	set flag	266
BI BY H W F D	tn SFILL	string fill	240
BI BY H W F D	tn SFILLN	string fill n elements	241
BY H W	t SHA	shift arithmetical	149
BY H W	t SHL	shift logical	148
BY H W	t SHR	shift rotational	150
F D	tn SIN	sine	191
BI BY	t SLOCA	string locate	247
BY	t SMATCH	string match	250
BI BY H W F D	t SMOVE	string move	234
BI BY H W F D	t SMOVN	string move n elements	239
BY	t SMVTR	move translated string	237
BY	t SMVTU	move string translated until	238
BY	t SMVUN	move string until	236
BY	t SMVWH	move string while	235
	SOLO	disable process switch	277
F D	tn SQRT	register square root	190
	SREGBL	save register block	309
BY	t SSCAN	string scan	248

Legal data formats	Assembly notation	Name	Page
BY	t SSKIP	skip elements	246
BY	t SSPAN	string span	249
BY	t SSPAR	set parity in string	251
	ST1 :=	load first status register	283
	ST1 =:	store first status register	284
BI BY H W F D	t STZ	store zero	141
BY H W F D	t SUB2	subtract two arguments	170
BY H W F D	t SUB3	subtract three arguments	174
W	tn SUBC	subtract with carry	182
	SVERS	store microprogram version	320
BI BY H W F D	t SWAP	swap	132
F D	tn TAN	tangent	195
	TEMM1 =:	store 1st trap enable mod. mask	284
	TEMM2 =:	store 2nd trap enable mod. mask	284
BI BY H W F D	t TEST	test against zero	135
	THA :=	load trap handler register	283
	THA =:	store trap handler register	284
	TOS :=	load top of stack register	283
	TOS =:	store top of stack register	284
	TOSSP	special load of TOS	315
W	tn TSET	test and set	279
	TUTTI	enable process switch	278
W	tn UDIV	unsigned divide	180
W	tn UMUL	unsigned multiply	179
BI BY H F D	t WCONR	convert to word with rounding	258
BI BY H F D	t WCONV	convert to word	256
W	WEXT	write to device external to CPU	314
BY	WHOLE	write to NUCLEUS hole	304
W	tn WPCONV	convert word to packed	338
	WPHS	write to physical address	317
BI BY H W	tn XOR	exclusive OR register	147
BI	ZPGU	reset page used table bit	297
BI	ZWIP	reset written in page table bit	294

Appendices G and H are connected through a reference number (column Ref.). The numbers found in the cross reference table of appendix H correspond to the reference number in appendix G. This helps translation from instruction codes, as found when dumping programs, to named instructions.

	BI	BY	H	W	F	D	Ref.	Page
tn :=	176004	004	010	014	020	024	1	125
B :=				176010			2	126
R :=				030			3	127
tn =:	176014	034	176020	040	044	050	4	128
B =:				176012			5	129
R =:				176011			6	130
t MOVE	176013	031	176024	032	033	054	7	131
t SWAP	176275	176276	176277	122	176334	176335	8	132
tn COMP	176030	060	176034	064	070	074	9	133
t COMP2	176025	055	176026	056	057	100	10	134
t TEST	101	102	103	104	105	106	11	135
tn NEG		177010	177014	220	224	224	12	136
tn INV	177020	177024	177030	230			13	137
tn INVC				177420			14	138
tn ABS		177400	177404	177410	177414	177414	15	139
tn CLR	204	204	204	204	210	214	16	140
t STZ	176205	110	111	112	113	114	17	141
t SET1	176206	176207	176210	115	107	176211	18	142
t INCR		176212	116	117	120	176213	19	143
t DECR		176214	176215	121	176216	176217	20	144
tn AND	176714	176220	176224	344			21	145
tn OR	176770	176230	176234	240			22	146
tn XOR	176774	176240	176244	244			23	147
t SHL		176250	176251	176252			24	148
t SHA		176253	176254	176255			25	149
t SHR		176256	176257	176260			26	150
tn GETBI		176264	176270	176720			27	151
tn PUTBI		176724	176730	176734			28	152
t CLEBI		177175	177176	177177			29	153
t SETBI		177200	177201	177202			30	154
tn GETBF		176740	176744	176750			31	155
tn PUTBF		176754	176760	176764			32	156
tn AMODB		177674	177700	177704			33	160
tn REM					177130	177134	34	157
tn INT					177140	177144	35	158
tn INTR					177150	177154	36	159
tn +		176064	176070	124	130	134	37	165
tn -		176074	176100	140	144	150	38	166
tn *		176104	176110	154	160	164	39	167
tn /		176114	176120	170	174	350	40	168
t ADD2		176027	176124	123	176126	176127	41	169
t SUB2		176130	176131	340	176133	176134	42	170
t MUL2		176135	176136	176137	176140	176141	43	171
t DIV2		176142	176143	176144	176145	176146	44	172

	BI	BY	H	W	F	D	Ref.	Page
t ADD3		176147	176150	176151	176152	176153	45	173
t SUB3		176154	176155	176156	176157	176160	46	174
t MUL3		176161	176162	176163	176164	176165	47	175
t DIV3		176166	176167	176170	176171	176172	48	176
tn MUL4		176040	176044	176050			49	177
tn DIV4		176054	176060	176174			50	178
tn UMUL				176200			51	179
tn UDIV				177110			52	180
tn ADDC				177100			53	181
tn SUBC				177104			54	182
tn MULAD		176350	176354	250	176360	176364	55	183
tn PSUM		176370	176374	176400	176404	176410	56	184
tn AXI					176300	176304	57	187
tn IXI		176310	176314	176320			58	188
tn POLY					176340	176344	59	189
tn SQRT					176324	176330	60	190
tn SIN					177530	177604	61	191
tn ASIN					177534	177610	62	192
tn COS					177540	177614	63	193
tn ACOS					177544	177620	64	194
tn TAN					177550	177624	65	195
tn ATAN					177554	177630	66	196
tn ATAN2					177560	177634	67	197
tn EXP					177564	177640	68	198
tn ALOG					177570	177644	69	199
tn ALOG2					177574	177650	70	200
tn ALOG10					177600	177654	71	201
:B GO				300			72	205
:H GO				301			73	205
:W GO				302			74	205
JUMPG				264			75	206
:B IF = GO				304			76	207
:H IF = GO				305			77	207
:B IF >< GO				306			78	207
:H IF >< GO				307			79	207
:B IF > GO				310			80	207
:H IF > GO				311			81	207
:B IF < GO				312			82	207
:H IF < GO				313			83	207
:B IF >= GO				314			84	207
:H IF >= GO				315			85	207
:B IF <= GO				316			86	207
:H IF <= GO				317			87	207
:B IF K GO				320			88	207

	BI	BY	H	W	F	D	Ref.	Page
:H IF K GO				321			89	207
:B IF -K GO				322			90	207
:H IF -K GO				323			91	207
:B IF >> GO				324			92	207
:H IF >> GO				325			93	207
:B IF >>= GO				326			94	207
:H IF >>= GO				327			95	207
:B IF << GO				330			96	207
:H IF << GO				331			97	207
:B IF <<= GO				332			98	207
:H IF <<= GO				333			99	207
:B IF ST GO				176173			100	207
:H IF ST GO				176544			101	207
:B IF -ST GO				176545			102	207
:H IF -ST GO				176204			103	207
:B t LOOPI		176336	176337	277	176434	176435	104	209
:H t LOOPI		176436	176437	341	176441	176442	105	209
:B t LOOPD		176443	176444	176445	176446	176447	106	211
:H t LOOPD		176450	176451	176452	176453	176454	107	211
:B t LOOP		176455	176456	176457	176460	176461	108	213
:H t LOOP		176462	176463	176464	176465	176466	109	213
CALL				303			110	216
CALLG				265			111	215
INIT				334			112	217
ENTM				337			113	219
ENTD				234			114	220
ENTS				270			115	221
ENTF				335			116	222
ENTSN				272			117	221
ENTFN				336			118	222
ENTT				274			119	223
ENTB				275			120	225
RET				200			121	226
RETK				201			122	226
RETB				177034			123	226
RETBK				177035			124	226
RETD				202			125	226
RETT				203			126	226
IF K RET				235			127	226
t SMOVE	176546	176547	176550	176551	176552	176553	128	234
t SMVWH		176562					129	235
t SMVUN		176563					130	236
t SMVTR		176564					131	237
t SMVTU		176565					132	238

		BI	BY	H	W	F	D	Ref.	Page
t	SMOVN	176566	176567	176570	176571	176572	176573	133	239
tn	SFILL	176574	176600	176604	176610	176614	176620	134	240
tn	SFILLN	176624	176630	176634	176640	176644	176650	135	241
t	SCOMP		176654					136	242
t	SCOTR		176655					137	243
t	SCOPA		176676					138	244
t	SCOPT		176677					139	245
t	SSKIP		176656					140	246
t	SLOCA	176657	176660					141	247
t	SSCAN		176661					142	248
t	SSPAN		176662					143	249
t	SMATCH		176663					144	250
t	SSPAR		176664					145	251
t	SCHPAR		176665					146	252
t	BMOVE		176440	177170	177171	177172	177173	147	255
t	BICONV		176511	176516	176523	176530	176535	148	256
t	BYCONV	176504		176517	176524	176531	176536	149	256
t	HCONV	176505	176512		176525	176532	176537	150	256
t	WCONV	176506	176513	176520		176533	176540	151	256
t	FCONV	176507	176514	176521	176526		176541	152	256
t	DCONV	176510	176515	176522	176527	176534		153	256
t	BYCONR					177160	177161	154	258
t	HCONR					177162	177163	155	258
t	WCONR					177164	177165	156	258
t	FCONR				177203		177204	157	258
t	ENTIER					176707	176710	159	161
tn	LADDR	177040	177044	177050	176474	176474	177054	160	259
t	RLADDR	176125	176132	176261	276	276	176262	161	260
t	BLADDR	176263	176274	176467	176543	176543	176470	162	261
tn	CHAIN				176554			163	262
tn	LIND		176414	176420	254	177710	177714	164	263
tn	CIND		176424	176430	260	177720	177724	165	264
	NOOP				003			166	265
	SETK				177002			167	266
	CLRK				177003			168	267
Wn	GETB				177114			169	268
	FREEB				176666			170	269
	SOLO				177000			171	277
	TUTTI				177001			172	278
t	TSET		176500					173	279
	BP				002			174	280
	SETE				176471			175	281
	CLTE				176472			176	282
L	:=				176473			177	283

	BI	BY	H	W	F	D	Ref.	Page
HL :=				176667			178	283
LL :=				176670			179	283
ST1:=				176671			180	283
OTE1:=				176673			181	283
OTE2:=				176674			182	283
TOS:=				176675			183	283
TOSSP:=				177237			184	315
THA:=				176712			185	283
CAD:=				176672			186	318
L =:				176700			187	284
HL =:				176701			188	284
LL =:				176702			189	284
ST1=:				176703			190	284
OTE1=:				176705			191	284
OTE2=:				176706			192	284
MTE1=:				176560			193	284
MTE2=:				176561			194	284
CTE1=:				177120			195	284
CTE2=:				177121			196	284
TEMM1=:				177122			197	284
TEMM2=:				177123			198	284
CED=:				177124			199	284
CAD=:				177125			200	284
PS=:				177174			203	284
TOS=:				176711			204	284
THA=:				176713			205	284
P =:				176542			206	284
An :=				177060			207	285
En :=				177064			208	285
An =:				177070			209	285
En =:				177074			210	285
DCC				177425			211	286
PCC				177424			212	288
DMON				177426			213	289
PMON				177427			214	290
DMOF				177430			215	291
PMOF				177431			216	292
tn RWIP	177224		177230				217	293
BI ZWIP	177234						218	294
CWIP				177433			219	295
tn RPGU	177210		177214				220	296
BI ZPGU	177220						221	297
CPGU				177432			222	298

		BI	BY	H	W	F	D	Ref.	Page
t	RIOM			177166				223	299
	PCTSB				177434			224	300
	DCTSB				177435			225	300
	DDIRT				177772			226	287
tn	RDUS	177240	177244	177250	177254			227	301
	PLCCN				177775			228	270
	NCPLC				177776			229	271
	WPHS				177764			230	317
	RPHS				177765			231	316
tn	REXT				177750			232	313
tn	WEXT				177754			233	314
	WHOLE		177235					234	304
	RHOLE		177236					235	303
W1	SEND				266			236	305
W1	RECVE				267			237	306
	LREGBL				177766			238	310
	SREGBL				177767			239	309
	LCNTXT				177770			240	312
	SCNTXT				177771			241	311
	JUMPS				271			242	319
	SVERS				177773			243	320
	SCPUNO				177774			244	321
Wn	PHYLADR				177760			245	322
	PADD				177260			246	330
	PADDR				177205			247	330
	PSUB				177261			248	331
	PSUBR				177206			249	331
	PMPY				177264			250	332
	PMPYR				177221			251	332
	PCOMP				177263			252	333
	PSHIFT				177262			253	334
	PSHIFTR				177207			254	334
	PPACK				177265			255	335
	PPACKR				177222			256	335
	PUPACK				177266			257	336
	PUPACKR				177223			258	336
Wn	PWCONV				177274			259	337
Wn	WPCONV				177270			260	338
t	SSMOV		177167	(SSMOV reserved for future use)				261	
t	RES1				236			262	
t	RES2				237			263	
t	RES3				177004			264	
t	RES4				177005			265	
t	RES5				177006			266	

	BI	BY	H	W	F	D	Ref. Page
t RES6				177007			267
t RES7				177036			268
t RES8				177037			269
t CLINIT				177436			270
t CLREAD				177437			271
tn RES11		177300	177320	177340	177360	177440	272
tn RES12		177304	177324	177344	177364	177444	273
tn RES13		177310	177330	177350	177370	177450	274
tn RES14		177314	177334	177354	177374	177454	275
t RES15		177460	177470	177500	177510	177520	276
t RES16		177461	177471	177501	177511	177521	277
t RES17		177462	177472	177502	177512	177522	278
t RES18		177463	177473	177503	177513	177523	279
t RES19		177464	177474	177504	177514	177524	280
t RES20		177465	177475	177505	177515	177525	281
t RES21		177466	177476	177506	177516	177526	282
t RES22		177467	177477	177507	177517	177527	283
tn				360			284
tn				364			285
tn				370			286
tn				374			287

Appendices G and H are connected through a reference number (column Ref.). The numbers found in the cross reference table of appendix H correspond to the reference number in appendix G. This helps translation from instruction codes, as found when dumping programs, to named instructions.

	0	1	2	3	4	5	6	7
000000	0	0	174W	166W	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	18F
000110	17BY	17H	17W	17F	17D	18W	19H	19W
000120	19F	20W	8W	41W	37W	37W	37W	37W
000130	37F	37F	37F	37F	37D	37D	37D	37D
000140	38W	38W	38W	38W	38F	38F	38F	38F
000150	38D	38D	38D	38D	39W	39W	39W	39W
000160	39F	39F	39F	39F	39D	39D	39D	39D
000170	40W	40W	40W	40W	40F	40F	40F	40F
000200	121W	122W	125W	126W	16W *	16W *	16W *	16W *
000210	16F	16F	16F	16F	16D	16D	16D	16D
000220	12W	12W	12W	12W	12D *	12D *	12D *	12D *
000230	13W	13W	13W	13W	114W	127W	262W	263W
000240	22W	22W	22W	22W	23W	23W	23W	23W
000250	55W	55W	55W	55W	164W	164W	164W	164W
000260	165W	165W	165W	165W	75W	111W	236W	237W
000270	115W	242W	117W	0	119W	120W	161F *	104W
000300	72W	73W	74W	110W	76W	77W	78W	79W
000310	80W	81W	82W	83W	84W	85W	86W	87W
000320	88W	89W	90W	91W	92W	93W	94W	95W
000330	96W	97W	98W	99W	112W	116W	118W	113W
000340	42W	105W	0	0	21W	21W	21W	21W
000350	40D	40D	40D	40D	0	0	0	0

Note: 000360 to 000377 are codes reserved for two-byte instruction codes:

000360	284W	284W	284W	284W	285W	285W	285W	285W
000370	286W	286W	286W	286W	287W	287W	287W	287W

Note: 170000 to 175777 are reserved codes.

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	41BY
176030	9BI	9BI	9BI	9BI	9H	9H	9H	9H
176040	49BY	49BY	49BY	49BY	49H	49H	49H	49H
176050	49W	49W	49W	49W	50BY	50BY	50BY	50BY
176060	50H	50H	50H	50H	37BY	37BY	37BY	37BY
176070	37H	37H	37H	37H	38BY	38BY	38BY	38BY
176100	38H	38H	38H	38H	39BY	39BY	39BY	39BY
176110	39H	39H	39H	39H	40BY	40BY	40BY	40BY
176120	40H	40H	40H	40H	41H	161BI	41F	41D
176130	42BY	42H	161BY	42F	42D	43BY	43H	43W
176140	43F	43D	44BY	44H	44W	44F	44D	45BY
176150	45H	45W	45F	45D	46BY	46H	46W	46F
176160	46D	47BY	47H	47W	47F	47D	48BY	48H
176170	48W	48F	48D	100W	50W	50W	50W	50W

	0	1	2	3	4	5	6	7
176200	51W	51W	51W	51W	103W	17BI	18BI	18BY
176210	18H	18D	19BY	19D	20BY	20H	20F	20D
176220	21BY	21BY	21BY	21BY	21H	21H	21H	21H
176230	22BY	22BY	22BY	22BY	22H	22H	22H	22H
176240	23BY	23BY	23BY	23BY	23H	23H	23H	23H
176250	24BY	24H	24W	25BY	25H	25W	26BY	26H
176260	26W	161H	161D	162BI	27BY	27BY	27BY	27BY
176270	27H	27H	27H	27H	162BY	8BI	8BY	8H
176300	57F	57F	57F	57F	57D	57D	57D	57D
176310	58BY	58BY	58BY	58BY	58H	58H	58H	58H
176320	58W	58W	58W	58W	60F	60F	60F	60F
176330	60D	60D	60D	60D	8F	8D	104BY	104H
176340	59F	59F	59F	59F	59D	59D	59D	59D
176350	55BY	55BY	55BY	55BY	55H	55H	55H	55H
176360	55F	55F	55F	55F	55D	55D	55D	55D
176370	56BY	56BY	56BY	56BY	56H	56H	56H	56H
176400	56W	56W	56W	56W	56F	56F	56F	56F
176410	56D	56D	56D	56D	164BY	164BY	164BY	164BY
176420	164H	164H	164H	164H	165BY	165BY	165BY	165BY
176430	165H	165H	165H	165H	104F	104D	105BY	105H
176440	147BY	105F	105D	106BY	106H	106W	106F	106D
176450	107BY	107H	107W	107F	107D	108BY	108H	108W
176460	108F	108D	109BY	109H	109W	109F	109D	162H
176470	162D	175W	176W	177W	160F *	160F *	160F *	160F *
176500	173BY	0	0	0	149BI	150BI	151BI	152BI
176510	153BI	148BY	150BY	151BY	152BY	153BY	148H	149H
176520	151H	152H	153H	148W	149W	150W	152W	153W
176530	148F	149F	150F	151F	153F	148D	149D	150D
176540	151D	152D	206W	162F *	101W	102W	128BI	128BY
176550	128H	128W	128F	128D	163W	163W	163W	163W
176560	193W	194W	129BY	130BY	131BY	132BY	133BI	133BY
176570	133H	133W	133F	133D	134BI	134BI	134BI	134BI
176600	134BY	134BY	134BY	134BY	134H	134H	134H	134H
176610	134W	134W	134W	134W	134F	134F	134F	134F
176620	134D	134D	134D	134D	135BI	135BI	135BI	135BI
176630	135BY	135BY	135BY	135BY	135H	135H	135H	135H
176640	135W	135W	135W	135W	135F	135F	135F	135F
176650	135D	135D	135D	135D	136BY	137BY	140BY	141BI
176660	141BY	142BY	143BY	144BY	145BY	146BY	170W	178W
176670	179W	180W	186W	181W	182W	183W	138BY	139BY
176700	187W	188W	189W	190W	0	191W	192W	159F
176710	159D	204W	185W	205W	21BI	21BI	21BI	21BI
176720	27W	27W	27W	27W	28BY	28BY	28BY	28BY
176730	28H	28H	28H	28H	28W	28W	28W	28W
176740	31BY	31BY	31BY	31BY	31H	31H	31H	31H
176750	31W	31W	31W	31W	32BY	32BY	32BY	32BY
176760	32H	32H	32H	32H	32W	32W	32W	32W
176770	22BI	22BI	22BI	22BI	23BI	23BI	23BI	23BI
177000	171W	172W	167W	168W	264W	265W	266W	267W
177010	12BY	12BY	12BY	12BY	12H	12H	12H	12H
177020	13BI	13BI	13BI	13BI	13BY	13BY	13BY	13BY
177030	13H	13H	13H	13H	123W	124W	268W	269W
177040	160BI	160BI	160BI	160BI	160BY	160BY	160BY	160BY
177050	160H	160H	160H	160H	160D	160D	160D	160D
177060	207W	207W	207W	207W	208W	208W	208W	208W

	0	1	2	3	4	5	6	7
177070	209W	209W	209W	209W	210W	210W	210W	210W
177100	53W	53W	53W	53W	54W	54W	54W	54W
177110	52W	52W	52W	52W	169W	169W	169W	169W
177120	195W	196W	197W	198W	199W	200W	201W	202W
177130	34F	34F	34F	34F	34D	34D	34D	34D
177140	35F	35F	35F	35F	35D	35D	35D	35D
177150	36F	36F	36F	36F	36D	36D	36D	36D
177160	154F	154D	155F	155D	156F	156D	223H	261BY
177170	147H	147W	147F	147D	203W	29BY	29H	29W
177200	30BY	30H	30W	157W	157D	247W	249W	254W
177210	220BI	220BI	220BI	220BI	220H	220H	220H	220H
177220	221BI	251W	256W	258W	217BI	217BI	217BI	217BI
177230	217H	217H	217H	217H	218BI	234BY	235BY	184W
177240	227BI	227BI	227BI	227BI	227BY	227BY	227BY	227BY
177250	227H	227H	227H	227H	227W	227W	227W	227W
177260	246W	248W	253W	252W	250W	255W	257W	0
177270	260W	260W	260W	260W	259W	259W	259W	259W
177300	272BY	272BY	272BY	272BY	273BY	273BY	273BY	273BY
177310	274BY	274BY	274BY	274BY	275BY	275BY	275BY	275BY
177320	272H	272H	272H	272H	273H	273H	273H	273H
177330	274H	274H	274H	274H	275H	275H	275H	275H
177340	272W	272W	272W	272W	273W	273W	273W	273W
177350	274W	274W	274W	274W	275W	275W	275W	275W
177360	272F	272F	272F	272F	273F	273F	273F	273F
177370	274F	274F	274F	274F	275F	275F	275F	275F
177400	15BY	15BY	15BY	15BY	15H	15H	15H	15H
177410	15W	15W	15W	15W	15D *	15D *	15D *	15D *
177420	14W	14W	14W	14W	212W	211W	213W	214W
177430	215W	216W	222W	219W	224W	225W	270W	271W
177440	272D	272D	272D	272D	273D	273D	273D	273D
177450	274D	274D	274D	274D	275D	275D	275D	275D
177460	276BY	277BY	278BY	279BY	280BY	281BY	282BY	283BY
177470	276H	277H	278H	279H	280H	281H	282H	283H
177500	276W	277W	278W	279W	280W	281W	282W	283W
177510	276F	277F	278F	279F	280F	281F	282F	283F
177520	276D	277D	278D	279D	280D	281D	282D	283D
177530	61F	61F	61F	61F	62F	62F	62F	62F
177540	63F	63F	63F	63F	64F	64F	64F	64F
177550	65F	65F	65F	65F	66F	66F	66F	66F
177560	67F	67F	67F	67F	68F	68F	68F	68F
177570	69F	69F	69F	69F	70F	70F	70F	70F
177600	71F	71F	71F	71F	61D	61D	61D	61D
177610	62D	62D	62D	62D	63D	63D	63D	63D
177620	64D	64D	64D	64D	65D	65D	65D	65D
177630	66D	66D	66D	66D	67D	67D	67D	67D
177640	68D	68D	68D	68D	69D	69D	69D	69D
177650	70D	70D	70D	70D	71D	71D	71D	71D
177660	0	0	0	0	0	0	0	0
177670	0	0	0	0	33BY	33BY	33BY	33BY
177700	33H	33H	33H	33H	33W	33W	33W	33W
177710	164F	164F	164F	164F	164D	164D	164D	164D
177720	165F	165F	165F	165F	165D	165D	165D	165D
177730	0	0	0	0	0	0	0	0
177740	0	0	0	0	0	0	0	0
177750	232W	232W	232W	232W	233W	233W	233W	233W

	0	1	2	3	4	5	6	7
177760	245W	245W	245W	245W	230W	231W	238W	239W
177770	240W	241W	226W	243W	244W	228W	229W	0

This table indicates the effect of all instructions on the status register. The following codes are used:

C	- unconditionally cleared
S	- unconditionally set
space	- unaffected
*	- set or reset depending on operand value
I	- set or reset if integer instruction, otherwise cleared
F	- set or reset if float instruction, otherwise cleared
A	- addressing status; set or reset depending on operand addresssing
PV	- protect violation

Staus bits abbreviations:

ATF	Address trap fetch	IOS	Illegal operand specifier
ATR	Address trap read	IOV	Illegal operand value
ATW	Address trap write	ISE	Instruction sequence error
AZ	Address zero trap	IVO	Invalid operation
BO	BCD overflow	IX	Illegal index
BPT	Breakpoint instruction trap	K	Flag
BT	Branch trap	O	Integer overflow
C	Carry	PSD	Process switch disabled
CT	Call trap	S	Sign
DR	Descriptor range	SIT	Single instruction trap
DZ	Divide by zero	STO	Stack overflow
FO	Floating overflow	STU	Stack underflow
FU	Floating underflow	XSE	Index scaling error
IIC	Illegal instruction code	Z	Zero

Some traps conditions not listed in the table may occur in all instructions. They are not caused by execution of any specific instruction, but may be set at any time if certain hardware or software conditions occur. These trap conditions include:

- programmed trap
- disable process switch timeout
- disable process switch error
- protect violation
- trap handler missing
- page fault
- power fail
- processor fault
- hardware fault

	P						I					I	S			B	A	A	A				S	S	X	I	I	I
	S						V	D	F	F	B	O	I	B	C	P	T	T	T	A	D	I	T	T	S	I	O	S
....	D	Z	C	S	K	O	O	Z	U	O	O	V	T	T	T	T	F	R	W	Z	R	X	O	U	E	C	S	E
*		*	I	*		I	C	C	F	F	C		A				A	A		A	A	A			A		*	
+		*	I	*		I	C	C	F	F	C		A				A	A		A	A	A			A		*	
-		*	I	*		I	C	C	F	F	C		A				A	A		A	A	A			A		*	
/		*	I	*		I	C	*	F	C	C		A				A	A		A	A	A			A		*	
:=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
=:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
ABS		*	C	*		I	C	C	C	C	C		A				A											
ACOS		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
ADD2		*	I	*		I	C	C	F	F	C		A				A	A	A	A	A	A			A		*	
ADD3		*	I	*		I	C	C	F	F	C		A				A	A	A	A	A	A			A		*	
ADDC		*	*	*		*	C	C	C	C	C		A				A	A		A	A	A			A		*	
ALOG		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
ALOG10		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
ALOG2		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
AND		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
AMODB		*	C	*		I	C	*	C	C	C		A				A	A		A	A	A			A		*	
ASIN		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
ATAN		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
ATAN2		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
AXI		*	C	*		C	C	C	F	F	C		A				A	A		A	A	A			A		*	
An :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
An =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
B :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
B =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
BLADDR		*	C	C		C	C	C	C	C	C		A				A			A	A	A			A			
BMOVE		C	C	C		C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
BP													A			*	A									*		
BYCONR		*	C	*		*	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
BYCONV		*	C	*		*	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
CAD :=		*	C	C		C	C	C	C	C	C		A				A	A	A	A	A	A			A	*	*	
CAD =:		*	C	C		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
CALL													A				A	A		A	A	A			A			A
CALLG													A				A	A		A	A	A			A		*	A
CED =:		*	C	C		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
CHAIN		*	C	C	*	C	C	C	C	C	C	*	A				A	A		A	A	A			A		*	
CIND		*	C	*		*	C	C	C	C	C		A				A	A		A	A	A			A		*	
CLEBI		S	C	C		C	C	C	C	C	C	*	A				A	A	A	A	A	A			A		*	
CLINIT													A				A									*		
CLR		S	C	C		C	C	C	C	C	C		A				A											
CLREAD													A				A											
CLRK					C								A				A								A		*	
CLTE												*	A				A	A		A	A	A			A		*	
COMP		*	I	*		C	C	C	F	F	C		A				A	A		A	A	A			A		*	
COMP2		*	I	*		C	C	C	F	F	C		A				A	A		A	A	A			A		*	
COS		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
CPGU													A				A									*		
CTE1 =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
CTE2 =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
CWIP													A				A									*		
DCC													A				A									*		
DCONV		*	C	*		C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
DCTSB													A				A									*		
DECR		*	I	*		I	C	C	C	C	C		A				A	A	A	A	A	A			A		*	

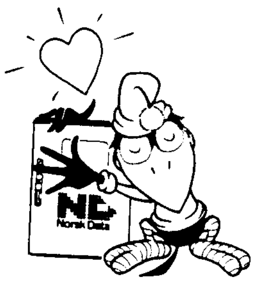
```
        P      :    I  :      I:S     B:A A A  :    S S:X I I I
        S      :    V D:F F B O:I B C P:T T T A:D I T T:S I O S
........D Z C S:K O O Z:U O O V:T T T T:F R W Z:R X O U:E C S E
               :       :       :       :       :       :
DIV2      * C *:  I C *:F F C  :A      :A A A A:A A    :A   *
DIV3      * C *:  I C *:F F C  :A      :A A A A:A A    :A   *
DIV4      * C *:  I C *:C C C  :A      :A A A A:A A    :A   *
DMOF           :       :       :A      :A      :       :  *
DMON           :       :       :A      :A      :       :  *
ENTB           :       :       :A S S  :A A   A:A A *  :A   * A
ENTD           :       :       :A S S  :A A    :  A    :      A
ENTF           :       :       :A S S  :A A   A:A A    :A   * A
ENTFN          :       :      *:A S S  :A A   A:A A    :A   * A
ENTIER    * C *:  I C C:C C C C:A A A  :A A   A:A A    :A   * A
ENTM           :       :       :A S S  :A A   A:A A *  :A   * A
ENTS           :       :       :A S S  :A A   A:A A *  :A   * A
ENTSN          :       :      *:A S S  :A A   A:A A *  :A   * A
ENTT           :       :       :A S S  :A A   A:A A *  :A   * A
En :=     * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
En =:     * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
FCONR     * C *:  C C C:C C C  :A      :A A A A:A A    :A   *
FCONV     * C *:  C C C:C C C  :A      :A A A A:A A    :A   *
FREEB          :       :       :A      :A   A A:A A    :A   *
GETB           :       :       :A      :A A   A:A A *  :A   *
GETBF     * C *:  C C C:C C C *:A      :A A   A:A A    :A   *
GETBI     * C C:  C C C:C C C *:A      :A A   A:A A    :A   *
GO:B           :       :       :A S    :A     A:       :
GO:H           :       :       :A S    :A     A:       :
GO:W           :       :       :A S    :A     A:       :
HCONR     * C *:  * C C:C C C  :A      :A A A A:A A    :A   *
HCONV     * C *:  * C C:C C C  :A      :A A A A:A A    :A   *
HL :=     * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
HL =:     * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
IF -C  GO      :       :       :A *    :A     A:       :
IF -K  GO      :       :       :A *    :A     A:       :
IF -S  GO      :       :       :A *    :A     A:       :
IF -ST GO      :       :      *:A *    :A A   A:       :A   *
IF -Z  GO      :       :       :A *    :A     A:       :
IF <rel> GO    :       :       :A *    :A     A:       :
IF C   GO      :       :       :A *    :A     A:       :
IF K   GO      :       :       :A *    :A     A:       :
IF K   RET     :       :       :A *    :A     A:       :
IF S   GO      :       :       :A *    :A     A:       :
IF ST  GO      :       :      *:A *    :A A   A:       :A   *
IF Z   GO      :       :       :A *    :A     A:       :
INCR      * C *:  I C C:C C C  :A      :A A A A:A A    :A   *
INIT           :       :       :A      :A A   A:A A *  :A   *
INT       * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
INTR      * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
INV       * C *:  C C C:C C C  :A      :A      :       :
INVC      * * *:  * C C:C C C  :A      :A      :       :
IXI       * C *:  * C C:C C C *:A      :A A   A:A A    :A   *
JUMPG          :       :       :A S    :A A   A:A A    :A   *
JUMPS   S      :       :       :       :       :       :  *
L :=      * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
L =:      * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
LADDR     * C *:  C C C:C C C  :A      :A     A:A A    :A   *
```

	P						I					I	S			B	A	A	A				S	S	X	I	I	I
	S						V	D	F	F	B	O	I	B	C	P	T	T	T	A	D	I	T	T	S	I	O	S
....	D	Z	C	S	K	O	O	Z	U	O	O	V	T	T	T	T	F	R	W	Z	R	X	O	U	E	C	S	E
LCNTXT	*	*	*	*	*	*	*	*	*	*	*	*	A	*	*	*	A	A		A	A	A	*	*	A	*	A	*
LIND		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
LL :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
LL =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
LOOP		*	C	*		C	C	C	C	C	C	*	A	*			A	A	A	A	A	A			A		*	A
LOOPD		*	C	*		C	C	C	C	C	C		A	*			A	A	A	A	A	A			A		*	A
LOOPI		*	C	*		C	C	C	C	C	C		A	*			A	A	A	A	A	A			A		*	A
LREGBL													A				A	A		A	A	A			A		*	
MOVE		*	C	*		C	C	C	C	C	C		A	A			A	A		A	A	A			A		*	
MTE1 :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
MTE1 =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
MTE2 :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
MTE2 =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
MUL2		*	I	*		I	C	C	F	F	C		A				A	A	A	A	A	A			A		*	
MUL3		*	I	*		I	C	C	F	F	C		A				A	A	A	A	A	A			A		*	
MUL4		*	C	*		*	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
MULAD		*	I	*		I	C	C	F	F	C		A				A	A	A	A	A	A			A		*	
NCPLC		*	C	*		C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
NEG		*	*	*		*	C	C	C	C	C		A				A											
NOOP													A				A											
OR		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
OTE1 :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
OTE1 =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
OTE2 :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
OTE2 =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
P =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
PADD		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PADDR		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PCC													A				A											
PCOMP		*	C	*		C	*	C	C	C	*	C	A				A	A		A	A						*	
PCTSB													A				A									*		
PHYLADR													A				A			A	A	A			A		A	
PLCCN		*	C	*		C	C	C	C	C	C	C	A				A	A	A	A	A	A			A		A	
PMOF													A	*			A									*		
PMON													A	*			A									*		
POLY		*	C	*		C	C	C	*	*	C		A				A	A		A	A	A			A		*	
PMPY		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PMPYR		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PPACK		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PPACKR		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PS =:		*	C	C		C	C	C	C	C	C		A				A		A	A	A	A					*	
PSHIFT		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PSHIFTR		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PSUB		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PSUBR		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PSUM		*	I	*		I	C	C	*	*	C		A				A	A		A	A	A			A		*	
PUPACK		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PUPACKR		*	C	*		C	*	C	C	C	*	C	A				A	A	A	A	A						*	
PUTBF		*	C	*		C	C	C	C	C	C	*	A				A		A	A	A	A			A		*	
PUTBI		*	C	C		C	C	C	C	C	C	*	A				A		A	A	A	A			A		*	
PWCONV		*	C	*		*	*	C	C	C	*	C	A				A	A	A	A	A						*	
R :=		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A			A		*	
R =:		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	

....	P S D	Z	C	S	K	O	I V O	D Z	F U	F O	B O	I O V	S I T	B T	C T	B P T	A T F	A T R	A T W	A Z	D R	I X	S T O	S T U	X S E	I I C	I O S	I S E
RDUS		*	C	*		C	C	C					A				A	A		A	A	A					*	
RECVE					*								A				A	A		A	A	A			A			
REM		*	C	*		C	C	*	C	C	C		A				A	A		A	A	A			A		*	
RET					C								A	*			A			A				*				
RETB					C								A	*			A			A				*				
RETBK					S								A	*			A			A				*				
RETD													A	*			A			A				*				
RETK					S								A	*			A			A				*				
RETT													A	*			A			A								
REXT					*								A				A	A		A	A	A			A	*		
RHOLE					*								A				A	A	A	A	A						*	
RIOM													A				A	A		A	A	A				*	*	
RLADDR		*	C	C		C	C	C	C	C	C		A				A			A	A	A			A		*	
RPGU		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A				*		
RPHS		*	C	C		C	C	C	C	C	C		A				A		A	A						*		
RWIP		*	C	*		C	C	C	C	C	C		A				A	A		A	A	A				*		
SCHPAR		*	C	*	*	C	C	C	C	C	C	*	A				A	A		A	A	A					*	
SCNTXT													A				AA			A					*	*	A	
SCOMP		*	C	*	*	C	C	C	C	C	C		A				A	A		A	A						*	
SCOPA		*	C	*	*	C	C	C	C	C	C		A				A	A		A	A						*	
SCOPT		*	C	*	*	C	C	C	C	C	C		A				A	A		A	A	A					*	
SCOTR		*	C	*	*	C	C	C	C	C	C		A				A	A		A	A	A					*	
SCPUNO		*	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
SEND					*								A				A	A		A	A	A			A		*	PV
SET1		C	C	*		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
SETBI		C	C	C		C	C	C	C	C	C	*	A				A		A	A	A	A			A		*	
SETE												*	A				A	A		A	A	A			A		*	
SETK					S								A				A											
SFILL		C	C	C	*	C	C	C	C	C	C		A				A		A	A	A						*	
SFILLN		*	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
SHA		*	C	*		C	C	C	C	C	C	*	A				A	A	A	A	A	A			A		*	
SHL		*	C	*		C	C	C	C	C	C	*	A				A	A	A	A	A	A			A		*	
SHR		*	C	*		C	C	C	C	C	C	*	A				A	A	A	A	A	A			A		*	
SIN		*	C	*		C	*	C	C	C	C		A				A	A		A	A	A			A			
SLOCA		*	C	C	C	C	C	C	C	C	C		A				A	A		A	A	A					*	
SMATCH		*	C	C	C	C	C	C	C	C	C		A				A	A		A	A						*	
SMOVE		C	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A						*	
SMOVN		*	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
SMVTR		C	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
SMVTU		*	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
SMVUN		*	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
SMVWH		*	C	C	*	C	C	C	C	C	C		A				A	A	A	A	A	A			A		*	
SOLO	S																											
SQRT		*	C	*		C	*	C	*	*	C		A				A	A		A	A	A			A		*	
SREGBL													A				A	A		A	A	A			A		A	
SSCAN		*	C	C	C	C	C	C	C	C	C		A				A	A		A	A	A			A		*	
SSKIP		*	C	*	C	C	C	C	C	C	C		A				A	A		A	A	A			A		*	
SSPAN		*	C	C	C	C	C	C	C	C	C		A				A	A		A	A	A			A		*	
SSPAR		C	C	C	S	C	C	C	C	C	C	*	A				A	A		A	A	A			A		*	
ST1 :=		*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	A		*	
ST1 =:													A				A		A	A	A	A			A		*	
STZ		S	C	C		C	C	C	C	C	C		A				A		A	A	A	A			A		*	
SUB2		*	I	*		I	C	C	F	F	C		A				A	A	A	A	A	A			A		*	

```
        P      :    I  :      I:S     B:A A A  :    S S:X I I I
        S      :    V D:F F B O:I B C P:T T T A:D I T T:S I O S
........D Z C S:K O O Z:U O O V:T T T T:F R W Z:R X O U:E C S E
               :       :       :       :       :       :
SUB3      * I *:  I C C:F F C  :A      :A A A A:A A    :A   *
SUBC      * * *:  * C C:C C C  :A      :A A   A:A A    :A   *
SVERS     * C *:  C C C:C C C  :A      :A   A A:A A    :A   A
SWAP      * C *:  * C C:C C C  :A      :A A A A:A A    :A   *
TAN       * C *:  C * C:C C C  :A      :A A   A:A A    :A
TEMM1 =:  * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
TEMM2 =:  * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
TEST      * I *:  C C C:C C C  :A      :A A   A:A A    :A   *
THA :=    * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
THA =:    * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
TOS :=    * C *:  C C C:C C C  :A      :A A   A:A A    :A   *
TOS =:    * C *:  C C C:C C C  :A      :A   A A:A A    :A   *
TOSSP     * C *:  C C C:C C C  :A      :A A   A:A A    :A
TSET      * C *:  C C C:C C C  :A      :A A A A:A A    :A   *
TUTTI   C      :       :       :A      :A      :       :
UDIV      * C *:  C C *:C C C  :A      :A A   A:A A    :A   *
UMUL      * C *:  * C C:C C C  :A      :A A   A:A A    :A   *
WCONR     * C *:  * C C:C C C  :A      :A A A A:A A    :A   *
WCONV     * C *:  * C C:C C C  :A      :A A A A:A A    :A   *
WEXT           :*      :       :A      :A A   A:A A    :A *
WHOLE          :*      :       :A      :A A   A:A           * PV
WPCONV    * C *:  C C C:C C *  :A      :A     A:A      :    *
WPHS      * C C:  C C C:C C C  :A      :A A   A:       :  *
XOR       * C *:  C C C:C C C  :A      :A A A A:A A    :A   *
ZPGU           :       :       :A      :A A    :A A    :A * *
ZWIP           :       :       :A      :A A    :A A    :A * *
```

INDEX LIST



SEND US YOUR COMMENTS!

Are you frustrated because of unclear information in our manuals? Do you have trouble finding things?

Please let us know if you:
— find errors
— cannot understand information
— cannot find information
— find needless information.

Do you think we could improve our manuals by rearranging the contents? You could also tell us if you like the manual.

Send to:
Norsk Data A.S
Documentation Department
P.O. Box 25 BOGERUD
N - 0621 OSLO 6 - Norway

NOTE!

This form is primarily for documentation errors. Software and system errors should be reported on Customer System Reports.

Manual Name: ______________________ Manual number: ____________

Which version of the product are you using? ______________________

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? ______________________

____

Answer from Norsk Data

Answered by Date

Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
Oslo 6, Norway