NORD-500 ASSEMBLER
Reference Manual

NORSK DATA A.S

NORD-500 ASSEMBLER
Reference Manual

NOTICE

PRINTING RECORD

Printing	Notes
06/79	ORIGINAL PRINTING
05/80	SECOND EDITION – Replaces Original Printing

NORD-500 ASSEMBLER Reference Manual
Publication No. ND-60.113.02

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

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

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

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

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

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

PREFACE

The Reader:

We assume that you are a programmer who has a general knowledge of Assemblers. You may be an inexperienced or experienced assembler programmer. The structure of this manual will, we hope, benefit all.

The Manual:

In this manual we begin by briefly orienting you with the NORD-500 Assembler and its environment. The NORD-500 Assembler runs under the SINTRAN III operating system. We have also written two simple assembly programs and commented on them so that you can feel more comfortable with the NORD-500 Assembler. Apart from this, the manual is organized as a reference manual.

Related Manuals:

You must have the NORD-500 CPU Reference Manual for the complete definition of instructions and addressing modes.

The Product:

This manual describes the NORD-500 Assembler language, version 1.

TABLE OF CONTENTS

\+ + +

Section:		Page:

Appendix: *Page:*

1 INTRODUCTION

The NORD-500 Computer System consists of the NORD-500 CPU, the NORD-100 CPU and a shared memory. The NORD-500 Assembler is a two pass cross assembler which runs under the SINTRAN III operating system on the NORD-100 CPU, and produces relocatable code for the NORD-500 CPU (refer to Figure 1.1). The object code produced is in standard NORD Relocatable Format (NRF), which may be loaded by the NORD-500 loader. In addition to binary code, an assembly listing is produced. This listing consists of the NORD-500 source code. You also have the option of listing the produced code in octal format. The symbol table is printed after the listing. A cross reference table may be generated and printed at the end of the listing.

The same version of this assembler will run on both 32-bit and 48-bit floating point NORD-100 Central Processing Units.

1.1 *THE NORD-500 ASSEMBLER ENVIRONMENT UNDER SINTRAN III*

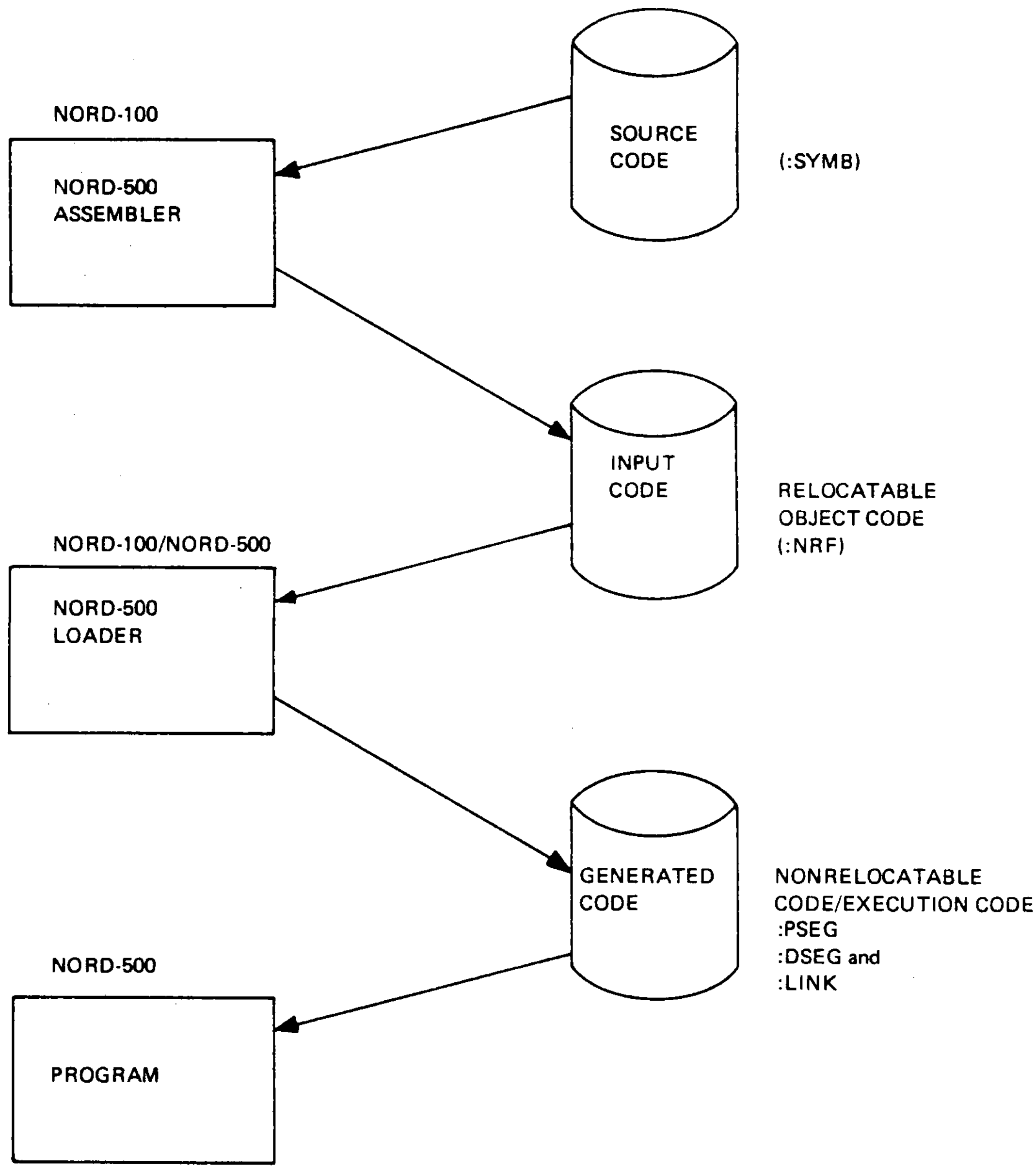

Figure 1.1.

1.2 DEFINITION OF ASSEMBLERS

During execution of a program, the instruction sequence is represented inside the computer by binary instructions. However, the programmer specifies instructions symbolically. The conversion from a symbolic representation of a program to its binary representation inside the computer can itself be performed by a computer program. This is referred to as the assembly process, and the program which performs the conversion is called an assembler.

An assembler is a program that accepts a program written in assembly language as input and produces its machine language equivalent. Each instruction word in an assembly language program is translated to only one instruction in machine language.

Thus, we can think of an assembler as a function, the domain of which is the set of all legal assembly language instructions, and the range of which is the corresponding set of machine language instructions. Operation of the assembler A on a symbolic assembly language program S produces a machine language program M, i.e., M = A (S).

Symbolic program S → [Assembler A] → Machine language program M

Figure 1.2: The Operation of an Assembler

On the following three pages are two simple examples using the NORD-500 Assembler. The output listing from Example 1 is found in Appendix E. Please note that the percent sign (%) indicates comments.

1.3 *EXAMPLE 1 – MODULE EXAMPLE*

```
I | D |     I: INSTRUCTION CODE, D: DIRECTIVE, % COMMENT
--+---+------------------------------------------------------------
  | X |         MODULE EXAMPLE          %   NAME OF MODULE
  | X |         MAIN START              %   SPECIFIES MAIN ENTRY POINT
  |   |
  | X |         ROUTINE LNG
  |   | %
  |   | %       COMPUTE:  PAR3 = SQRT( PAR1**2 + PAR2**2 )
  |   | %
  | X | DSTK:   STACK FIXED             %   START OF STACK DEFINITION
  | X | APAR1:  W BLOCK 1               %   ADDRESS OF 1. PARAMETER
  | X | APAR2:  W BLOCK 1               %   ---- " --- 2. -- " --
  | X | APAR3:  W BLOCK 1               %   ---- " --- 3. -- " --
  | X |         ENDSTACK
  |   | %
X |   | LNG:    ENTF DSTK               %   ENTER SUBROUTINE WITH
  |   |                                 %   FIXED DATA AREA BEGINNING
  |   |                                 %   AT 'DSTK'.
  |   |
X |   |         F1 := IND(B.APAR1)      %   LOAD 1. PARAMETER
X |   |         F1 * F1                 %   SQUARE
X |   |         F2 := IND(B.APAR2)      %   LOAD 2. PARAMETER
X |   |         F2 MULAD F2,F1          %   SQUARE AND ADD
X |   |         F2 SQRT F2              %   TAKE SQUAREROOT
X |   |         F2 =: IND(B.APAR3)      %   STORE IN 3. PARAMETER
X |   |         RET                     %   RETURN
  | X |         ENDROUTINE
  |   | %
  |   | %       MAIN PROGRAM
  |   | %
  | X | STK:    STACK FIXED             %   START OF STACK DEFINITION
  | X | A:      F DATA 3.0              %   DEFINE A AS 3.0
  | X | BB:     F DATA 4.0              %   DEFINE BB AS 4.0
  | X | C:      F BLOCK 1               %   DECLARE SPACE FOR
  |   |                                 %   ONE REAL VARIABLE.
  | X |         ENDSTACK
  |   | %
  |   | %       INITIATE STACK AREA WITH MAIN PROGRAM STACK
  |   | %       FRAME BEGINNING AT 'STK', LENGTH #SCLC, AND
  |   | %       TOTAL STACK DEMAND OF 100.
  |   | %          #SCLC IS AN INTRINSIC FUNCTION GIVING THE
  |   | %       SIZE OF THE STACK FRAME IN THE LAST PRECEEDING
  |   | %       DEFINITION.
  |   | %
X |   | START:  INIT STK,#SCLC,100
  |   | %
  |   | %       CALL ROUTINE WITH 3 LOCAL PARAMETERS A, BB
  |   | %       AND C.
  |   | %
X |   |         CALL LNG,3,B.A,B.BB,B.C
  |   | %
  |   | %       "RETURN" FROM MAIN PROGRAM, I.E. STOP.
  |   | %
X |   |         RET
  |   |
  | X |         ENDMODULE
```

1.4 *EXAMPLE 2 – MODULE HANOI*

```
NORD-500 ASSEMBLER 2.5      WEDNESDAY  26 MARCH 1980    1430:20      PAGE 1
MODULE HANOI.

        MODULE HANOI
        MAIN BEG
%
%       PROGRAM TO SOLVE THE PROBLEM OF THE TOWERS OF HANOI.
%
%       ONE PEG CONTAINS A STACK OF DISKS WITH DECREASING DIAMETERS,
%       SUCH THAT THE LARGEST DISK IS AT THE BOTTOM AND THE SMALLEST
%       AT THE TOP. THE OBJECTIVE IS TO MOVE THIS PILE TO ANOTHER
%       PEG, OBSERVING THE CONSTRAINTS THAT ONLY ONE DISK AT A
%       TIME IS TO BE MOVED, AND NO LARGER DISK MAY BE ON TOP OF A
%       SMALLER ONE.
%       A THIRD PEG IS USED AS AN INTERMEDIATE STORAGE.
%
%       THE RESULT OF THE PROGRAM IS A SEQUENCE OF RECORDS IN MEMORY.
%       EACH RECORD IS CONCERNED WITH THE MOVEMENT OF ONE DISK. IT
%       CONSISTS OF THE DISK NUMBER (1 BEING THE TOPMOST), THE PEG
%       FROM WHICH IT IS MOVED, AND THE DESTINATION PEG.
%
%
        RECORD                  %   START OF RECORD DEFINITION
NO:     W BLOCK 1
FR:     W BLOCK 1
TR:     W BLOCK 1
        ENDRECORD
%
%       ROUTINE TO DO THE MOVEMENT OF THE DISKS.
%
        ROUTINE MOVEV
        STACK
N:      W BLOCK 1               %   1. PARAMETER (CALL BY VALUE)
FROM:   W BLOCK 1               %   2. ----"----
VIA:    W BLOCK 1               %   3. ----"----
TO:     W BLOCK 1               %   4. ----"----
NM1:    W BLOCK 1               %   LOCAL VARIABLE
        ENDSTACK

MOVEV: ENTS #SCLC               %   ENTER STACK SUBROUTINE. STACKDEMAND
                                %   IS GIVEN BY #SCLC, THE SIZE OF THE
                                %   PRECEEDING STACK FRAME DECLARATION.

        W1 := B.N; W DECR W1    %   DECREMENT DISK NO. BY ONE AND
        W1 =: B.NM1             %   STORE IN LOCAL VARIABLE 'NM1'.

        IF-> GO MREST           %   MORE THAN ONE DISK TO MOVE ?
        CALL MOVED,0            %   NO, MOVE THIS DISK AND RETURN.
        RET
                                %   YES, MOVE 'NM1' DISKS FROM PEG 'FROM'
                                %   VIA PEG 'TO' TO PEG 'VIA'.

MREST: CALL MOVEV,4,IND(B.NM1),IND(B.FROM),IND(B.TO),IND(B.VIA)

        CALL MOVED,0            %   MOVE ONE DISK FROM 'FROM' TO 'TO'.

                                %   MOVE THE 'NM1' DISKS FROM 'VIA'
                                %   VIA 'FROM' TO PEG 'TO'.

        CALL MOVEV,4,IND(B.NM1),IND(B.VIA),IND(B.FROM),IND(B.TO)
        RET
```

```
NORD-500 ASSEMBLER 2.5      WEDNESDAY  26 MARCH 1980   1430:23       PAGE 2
MODULE HANOI.

%
%      RECORD THE MOVEMENT OF ONE DISK. R4 CONTAINS A POINTER
%      TO THE RECORD. #RCLC GIVES THE RECORD SIZE OF THE LAST
%      RECORD DEFINITION.
%
MOVED: ENTD                     %   ENTER "DIRECT" SUBROUTINE.
       W4 + #RCLC               %   INCREMENT RECORD POINTER.

       W MOVE B.N,W4.NO; W MOVE B.FROM,W4.FR; W MOVE B.TO,W4.TR

       RETD                     %   RETURN FROM "DIRECT" SUBROUTINE.
       ENDROUTINE
%
%      MAIN PROGRAM AND DATA INITIALIZATION.
%
STKSIZ:  EQU 2000               %   TOTAL (MAX) STACK DEMAND.

MSTK:  STACK FIXED
NN:    W DATA 3                 %   NUMBER OF DISKS.
NFROM: W DATA 1                 %   SOURCE PEG NO.
NVIA:  W DATA 2                 %   TEMPORARY PEG NO.
NTO:   W DATA 3                 %   DESTINATION PEG NO.
       ENDSTACK
%
%      INITIALIZE MAIN PROGRAM WITH LOCAL DATA AREA BEGINNING AT
%      'MSTK', STACKDEMAND IN MAIN PROGRAM IS #SCLC, AND TOTAL
%      STACKDEMAND 'STKSIZ'. THE RECORD AREA IS LOCATED AFTER THE
%      STACK AREA.
%
BEG:   INIT MSTK,#SCLC,STKSIZ
       W4 := ADDR(MSTK); W4 + STKSIZ-#RCLC
%
%      CALL SUBROUTINE TO DO THE MOVING. THE FOUR PARAMETERS ARE
%      PASSED WITH "CALL BY VALUE" TYPE TRANSFER.
%
       CALL MOVEV,4,IND(B.NN),IND(B.NFROM),IND(B.NVIA),IND(B.NTO)

       RET                      %   "RETURN" FROM MAIN PROG., I.E. STOP.

       ENDMODULE
```

```
NORD-500 ASSEMBLER 2.5      WEDNESDAY  26 MARCH 1980   1430:25      PAGE 3
SYMBOL TABLE

GLOBAL SYMBOLS

BEG                 W    P M     00000000105
FR                  W  A         00000000004
MOVEV               W    P       00000000000
MSTK                W     D      00000000000
NFROM               W  A         00000000030
NN                  W  A         00000000024
NO                  W  A         00000000000
NTO                 W  A         00000000040
NVIA                W  A         00000000034
STKSIZ              W  A         00000003720
TR                  W  A         00000000010


NORD-500 ASSEMBLER 2.5      WEDNESDAY  26 MARCH 1980   1430:25      PAGE 4


NO ERRORS DETECTED
```

2 THE ASSEMBLY LANGUAGE

In order to describe the syntax of the assembly language, we will use a meta language (i.e., a language to describe another language). The rules of this meta language are as follows:

- A meta variable is a sequence of *letters, digits,* and *hyphens*.
- A terminal symbol is represented as a string of characters enclosed within single or double quotes.
- Alternatives are separated by a slash /.
- Optional items are surrounded by brackets [ ].
- Parentheses ( ) can be used to group together constructs.
- A dollar sign **$** before a construct means repetition.
- A decimal number immediately preceding/following a dollar sign **$** specifies the minimum/maximum number of occurrences of the repeated construct.

Some basic constructs that are used in this manual are defined below:

- letter = "A"/ "B"/... / "Y"/ "Z";
- digit = "0"/ "1"/ .../ "8"/ "9";
- decimal number = 1 **$** digit;

2.1 *SOURCE PROGRAM FORMAT*

- The ASCII character set is used to represent source programs.
- All characters in the interval of 0-37B are ignored, *except* for form feed (14B), carriage return (15B), and end-of-file (27B).
- Lower case letters are converted to upper case.
- A percent sign (%), not occurring inside a string constant, means that the rest of the line is a comment.
- Blank lines are treated as comment lines.
- An ampersand &, not occurring inside a string constant, means that the current statement continues on the next line. You may only have blanks and comments after the ampersand on the current line. Ampersands may occur between basic elements, but not within them.
- A statement is terminated by a semicolon (;) or carriage return.
- Empty statements are permitted.

There are three types of "orders" (statements) you may give to the assembler:

Instructions (for example, W ADD2 OP1, OP2)

Instructions are translated into machine language instructions for placement in the user's program memory.

Directives (for example, MODULE)

Directives specify attributes of the generated NRF (such as naming the main entry point), allocate data storage, and preset constant data.

Commands (for example, $LIST)

Commands control the processing of the program text through conditional assembly, macro definition, listing options, and selection of program statements for assembly.

2.2 *BASIC ELEMENTS*

The basic elements which make up a source program are: identifiers, string constants, integer constants, real constants, and file names.

IDENTIFIER

An identifier may consist of letters, digits, number signs (#) and underscores (_). The first character must be a letter, question mark (?) or number sign. Two underscore characters may not be placed side by side. The underscore character is significant in the identifier. If an identifier starts with a question mark it is called invisible and is never listed in the symbol table dump. This feature is intended for use with generated symbols in macro calls. An identifier may be of any length, but only the first 16 characters are significant. The word "symbo," is synonymous with identifier. For a list of reserved symbols refer to Appendix C.

STRING CONSTANT

A string constant consists of a sequence of characters enclosed with single quotes. If a single quote is to be included in the string it must be written twice. The maximum length of a string constant is 80 characters.

INTEGER CONSTANT

An integer constant may be one of four forms: binary, octal, decimal, and hexadecimal. It consists of a sequence of digits, followed by a radix specifier, optionally followed by an exponent. The default radix is decimal. The radix specifiers are; X = binary, B = octal, D = decimal (default), and H = hexadecimal. The exponent is always interpreted as a decimal number. As an example: 15B3 is the same as 15000B or 1AH2. In order to avoid conflicts with identifiers, a hexadecimal constant must always start with a decimal digit (i.e., the constant FF_{16} must be written ØFFH). An integer constant is represented internally as a 32-bit 2's complement number.

REAL CONSTANT

A real constant must contain a decimal point which must not be the first character. An exponent may be specified, preceded by the letter E. A real constant is represented internally in the NORD-500 double precision floating point format (sign bit, 9-bit exponent, 54 (+ 1)-bit mantissa).

FILE NAME

A file name is a string of any characters. It is terminated by a space, comma, or carriage-return. No syntax check of file names is performed by the assembler. File names are used only in commands.

SYNTAX OF BASIC ELEMENTS:

identifier =	id-part-1 $(break-character id-part-2);
id-part-1 =	letter/ "#"/ "?";
id-part-2 =	letter/ digit/ "#";
break-character =	"_";
string-constant =	" ' " $ (<any character except ' >/ " ' ' ") " ' ";
integer-constant =	binary-constant/ octal-constant/ decimal-constant/ hex-constant;
binary-constant =	1$ binary-digit "X" [exponent];
octal-constant =	1$ octal-digit "B" [exponent];
decimal-constant =	1$ digit ["D" [exponent]];
hex-constant =	digit $ hex-digit "H" [exponent];
binary-digit =	"0"/ "1";
octal-digit =	"0"/ "1"/ ... "6"/ "7";
hex-digit =	digit/ "A"/ "B"/ "C"/ "D"/ "E"/ "F";
exponent =	decimal-number;
real-constant =	1$ digit "." $ digit ["E" ["+"/ "-"] exponent] ;
file-name =	1$<any character except comma or space >;

2.3 *INSTRUCTIONS*

This section describes the assembly format for NORD-500 instruction codes and operand specifiers. Please refer to the NORD-500 CPU Reference Manual for a complete description of instruction codes (octal value and assembly notation), addressing modes, address codes and operand specifiers. Refer also to Appendixes F, G, H and I. The assembly format for an instruction is:

[label] instruction code [operand specifiers].

Each part is described in the following sections.

2.3.1 *Labels*

A label is a definition of a symbol's address. The optional label consists of an identifier followed by a colon. An instruction may have more than one label. Labels are also allowed on empty statements (i.e., the label is immediately followed by end-of-line or semicolon). Labels on instruction lines are assigned the current value of the program location counter. (See Section 2.5 on DIRECTIVES, STACK and RECORD.)

2.3.2 *Instruction Codes*

The instruction code name is the main part of the instruction code. The instruction code name is a string of characters identifying the operation to be performed. The instruction code names are not reserved symbols in the assembler. If the instruction code name does not end with a special character (=, :, +, –, *, or /) it must be terminated by at least one space.

Many instruction codes start with a data type specifier. These are:

BI	Bit
BY	Byte (8 bits)
H	Half-word (16 bits)
W	Word (32-bit integer)
F	Single precision real (32-bit floating point)
D	Double precision real (64-bit floating point)

If the instruction uses one of the integer or floating point accumulators as a destination and/or source operand, the register number is specified following the data type specifier (e.g., W1 for integer accumulator one).

Spaces are allowed following the data type specifier and the register number. For the IF and GO operations, spaces are allowed before and after 'cond'. The following are examples of legal operation codes:

```
BY 1 COMP      BY 1COMP
BY1 COMP       BY1COMP
W SUB2         WSUB2
IF=GO          IF = GO
```

2.3.3 *Operand Specifiers*

The instruction code is followed by a list of zero or more operand specifiers, separated by commas. Operand specifiers are divided into two main categories: direct operands and general operands. Direct operands are operands found in the bytes immediately following the instruction code or the previous operand specifier. General operands are operands accessed via an address code.

2.3.3.1 Direct Operands

A direct operand is an absolute addresses of program or data; or a displacement, which applies to program addresses only.

Direct Absolute Addressing

A direct absolute addressing operand is always assembled as a 32-bit word. Examples of direct absolute addressing operands are the address in CALL (but not CALLG) and the address of the stack in ENTM. The former is a program address, the latter a data address.

Displacement Addressing
Displacements are used in the LOOP and GO instructions to address the destination. A displacement is stored as a word, half-word, or byte depending upon its magnitude. To force the displacement to be stored in a particular format, the following length specifiers can be used:

:B Store operand as a byte (8 bits)
:H Store operand as a half-word (16 bits)
:W Store operand as a word (32 bits)

:B and :H are legal for all GO and LOOP instructions while :W is legal only for GO (not for IF cond GO).

If the assembler is unable to select the correct storage format for a displacement, :B is selected. If this is not large enough, an error diagnostic results in pass two and the programmer is responsible for adding the correct length specifier. Example of legal GO instructions are:

```
GO LABX          GO LABX:B
GO LABX:W        IF = GO LABZ:H
```

2.3.3.2 General Operands

The general operand is the most common operand type. It is used when accessing constants, registers, and data memory. The NORD-500 has 10 different addressing modes and 2 operand specifier prefixes.

In most cases the assembler selects the optimal storage format for constants and displacements in general operands. If, however, you want to force the storage format to a particular length, the following data part length specifiers are available:

:S	Short (6 bits)
:B	Byte (8 bits)
:H	Half-word (16 bits)
:W	Word (32 bits)
:F	Single precision real (32-bit floating point)
:D	Double precision real (64-bit floating point)

Note that no type conversion of values is performed at assembly time. This means that an integer constant cannot be converted to a real constant by appending any of the :F or :D modifiers and vice versa.

The addressing modes and address codes are described in more detail in the "NORD-500 CPU Reference Manual". Otherwise, refer to Appendix F and G. All possible addressing modes, followed by a short description, are listed here. The following notation is used:

constant	Integer or real constant
disp	Displacement (absolute value)
dlabel	A data label
plabel	A program label
ADDR(label)	An assembler notation for converting the value of a label to a constant.

Rn Register number

BI1	BI2	BI3	BI4	BIn
BY1	BY2	BY3	BY4	BYn
H1	H2	H3	H4	Hn
W1	W2	W3	W4	Wn
F1	F2	F3	F4	Fn
D1	D2	D3	D4	Dn
R1	R2	R3	R4	Rn

Local Addressing

B.disp	Assembler selected format
B.disp:S	Forced short displacement
B.disp:B	Forced byte displacement
B.disp:H	Forced half-word displacement
B.disp:W	Forced word displacement

Local, Post Indexed Addressing

B.disp(Wn)	Assembler selected displacement format
B.disp:B(Wn)	Forced byte displacement
B.disp:H(Wn)	Forced half-word displacement
B.disp:W(Wn)	Forced word displacement

Local Indirect Addressing

IND (B.disp)	Assembler selected displacement format
IND (B.disp:B)	Forced byte displacement
IND (B.disp:H)	Forced half-word displacement
IND (B.disp:W)	Forced word displacement

Local Indirect, Post Indexed Addressing

IND (B.disp) (Wn)	Assembler selected displacement format
IND (B.disp:B) (Wn)	Forced byte displacement
IND (B.disp:H) (Wn)	Forced half-word displacement
IND (B.disp:W) (Wn)	Forced word displacement

Record Addressing

R.disp	Assembler selected displacement format
R.disp:S	Forced short displacement
R.disp:B	Forced byte displacement
R.disp:H	Forced half-word displacement
R.disp:W	Forced word displacement

Pre-Indexed Addressing

Rn.disp	Assembler selected displacement format
Rn.disp:B	Forced byte displacement
Rn.disp:H	Forced half-word displacement
Rn.disp:W	Forced word displacement

Absolute Addressing

dlabel	Absolute address (always 4 bytes)
dlabel:W	Absolute address (always 4 bytes)

Absolute, Post Indexed Addressing

dlabel (Wn)	Absolute address (always 4 bytes)
dlabel:W(Wn)	Absolute address (always 4 bytes)

Constant Operand

constant	Assembler selected constant format
constant :S	Forced short constant
constant :B	Forced byte constant
constant :H	Forced half-word constant
constant :W	Forced word constant
constant :F	Forced real constant
constant :D	Forced double real constant
ADDR (dlabel)	The address of a data memory location
ADDR (dlabel) :W	The address of a data memory location
ADDR (plabel)	The address of a program memory location
ADDR (plabel) :W	The address of a program memory location

Register Addressing

Rn	Register as operand BIn, BYn, Hn, Wn, Fn, and Dn.

Note: the register symbol used must be of the correct type.

For Example:

BY WCONV BY2, W4 is correct, while BY WCONV W2, W4 is illegal.

When used as an index register (pre-indexing or post-indexing) only W is legal. R1, ... R4 is legal in all positions. The register names are reserved symbols.

Descriptor Addressing

DESC (operand) (Rn)	The operand can be any general operand, except constant, register, descriptor, and alternative area.

Alternative Area

ALT (operand)	The operand can be any general operand, except alternative area, register, and constant.

2.4 EXPRESSIONS

Expressions are made up of operators and operands. The operator conducts the action which is to be performed upon the operands. An operand can have one of the following data types:

I Integer (32 bits, 2's complement number)
R Real (64 bits, NORD-500 double precision)
S String (character string, maximum 80 characters)

2.4.1 Operators and Operand Data Types

The available operators, in order of increasing priority, are listed below:

Priority:	Operator:	Operand Data Type:	Description:
1	OR	I	Logical or
1	XOR	I	Logical exclusive or
2	AND	I	Logical and
3	NOT	I	Logical negation (1's complement)
4	<	I, S	Less than
4	< =	I, S	Less than or equal to
4	=	I, R, S	Equal to
4	><	I, R, S	Not equal to
4	> =	I, S	Greater than or equal to
4	>	I, S	Greater than
5	+	I	Addition
5	–	I	Subtraction
6	*	I	Multiplication
6	/	I	Division
6	MOD	I	Modulo
6	SHIFT	I	Shift
7	Unary +	I, R	Unary plus
7	Unary –	I, R	Unary minus

In all cases where an integer and/or real operand is required, a string constant of length 0-4 will be converted to an integer where the characters are represented by their internal binary value, e.g., A = 101_8. A string constant of length 5-8 will be converted to a real value in the same manner.

In addition, an integer value can have one of the three following attributes:

A Absolute
P Program address
D Data address

No binary operator may have a program address on one side and a data address on the other side of it. The following table shows which combinations of operands are possible and what type the result has. Blank indicates that the combination is illegal, while a horizontal line indicates a non-existent combination. The slash (/) means operated on.

Operator:	A/A	A/P,D	P,D/A	P,D/P,D
OR	A			
XOR	A			
AND	A			
NOT	A		—	—
<	A			A
< =	A			A
=	A			A
><	A			A
> =	A			A
>	A			A
+	A	P,D	P,D	
—	A		P,D	A
*	A			
/	A			
MOD	A			
SHIFT	A			
Unary +	A	P,D	—	—
Unary —	A		—	—

In general, address arithmetic is allowed only for data addresses. If imported symbols are used in an arithmetic expression, only one symbol may occur in each expression, i.e., the difference between two imported symbols is not legal. With program addresses, arithmetic is allowed only with the special symbols defined above.

Note that address arithmetic, as program addresses, is permitted with the special symbols defined above. For example, GO LABX+3 is illegal while GO #PCLC+3 is legal. Because almost all NORD-500 instructions have variable length it is strongly advised not to use constructs such as #PCLC+3.

2.4.2 *Intrinsic Constants*

Intrinsic constants are constants that are pre-defined or system-supplied. The following five intrinsic constant names may be used to refer to the locations in the stack entry header.

PREVB	0	Saved B-register
RETA	4	Saved return address
SP	8	Stack pointer
AUX	12	System cell
NARG	16	Number of arguments supplied in call

The constant #ZEROP has a value of zero and is used as a program address.

The constant #ZEROD has a value of zero and is used as a data address.

```
            MODULE EXTRA
            .
            .
SIZ:        W DATA ELAB– #ZEROP
            .
            .
ELAB:
            ENDMODULE
```

will place the size of the program part of the module in the data location SIZ.

2.4.3 *Intrinsic Functions*

Intrinsic functions are functions that are pre-defined or system-supplied. A function can have arguments, enclosed within parentheses and separated by commas. This section describes the different intrinsic functions which are available to you.

These are the location counter symbols:

#PCLC	Program location counter
#DCLC	Data location counter
#SCLC	Stack location counter
#RCLC	Record location counter

These functions return the current value of the location counters. #SCLC is used when processing statements between STACK and ENDSTACK, and #RCLC when processing statements between RECORD and ENDRECORD. When used in the operand field of an instruction, a location counter symbol represents the address of the first byte of the instruction. When used in the operand field of an assembler directive (see Section 2.5), it represents the address of the first byte of the current data element. For example:

```
W   MOVE ADDR(#PCLC), R1

W   DATA 100, #DCLC+4
W   BLOCK 100
```

The first instruction loads the R1 register with the address of the instruction itself. The two following instructions define a descriptor with the described array immediately following it.

When #SCLC is used inside a STACK-ENDSTACK pair it represents the current stack displacement. When it is used outside a STACK-ENDSTACK pair it holds the size of the last stack block defined. This means that it can be used directly as the "stack demand" parameter in the entry point instructions. For example:

```
            STACK
PAR1A:      W BLOCK 1       % ADDRESS OF PARAMETER ONE
PAR2A:      W BLOCK 1       % ADDRESS OF PARAMETER TWO
            ENDSTACK
ROUTX:      ENTS #SCLC      % ENTER STACK
            .
            .
            .
            .
```

These statements define a stack block and insert the correct stack demand in the ENTS instruction.

#RCLC is used in a similar way for records. #SCLC is initialized to 20 at the start of a new stack definition while #RCLC is initialized to zero at the start of a new record definition.

#NCHR

The function #NCHR takes a string as its only argument and returns the length of the string. The length is returned as an absolute integer value. For example:

```
XSTR:        SEQU 'STRING OF CHARACTERS'
             BY  DATA  #NCHR (XSTR) ,XSTR
```

assembles a string preceded by its length.

#NARG

The function #NARG, which takes no arguments, returns the number of arguments supplied in the call to the macro currently being expanded. If used outside a macro its value is zero.

#DATE

To read the current date and time the function #DATE can be used. It is a function of no arguments and returns the current date and time in a double word as follows:

Bits 63-48,	16 bits,	Year
Bits 47-40,	8 bits,	Month
Bits 39-32,	8 bits,	Day
Bits 31-24,	8 bits,	Hour
Bits 23-16,	8 bits,	Minute
Bits 15-8,	8 bits,	Second
Bits 7-0,	8 bits,	Unused

This function is useful in keeping track of different versions of a program.

#LOG2

The function #LOG2, which takes an integer value as argument, returns the logarithm to base two of the argument. This funciton can be useful when used with the instructions ENTB, GETB and FREEB.

2.4.4 *Expression Syntax*

```
expression =    lfact $(("OR"/ "XOR") lfact);

lfact =         lneg $("AND" lneg);

lneg =          ["NOT"] rel;

rel =           sum relop sum;

relop =         "=" / "><" / "< =" / "<" / "> =" / ">";

sum =           factor $(("+" / "–") factor);

factor =        primary $(("*" / "/" / "MOD" / "SHIFT") primary);

primary =       ["+" / "–"]
                "("             (" expression ")"/
                                identifier/
                                string-constant/ integer-constant/ real-constant/
                                iconstant/ ifunction);

iconstant =     "PREVB"/ "RETA"/ "SP"/ "AUX"/ "NARG"/
                "#ZEROP", "#ZEROD";

ifunction =     "#NARG"/
                "#NCHR" "(" expression ")"/
                "#PCLC"/ "#DCLC"/ "#SCLC"/ "#RCLC"/
                "#DATE"/
                "#LOG2" "(" expression ")";
```

2.5 DIRECTIVES

Directives specify attributes of the generated NRF (NORD Relocatable Format), allocate data storage, and preset constant data. See Appendix A for a summary of directives.

This section describes all available directives. The format of a directive is similar to that of an instruction.

[label] directive-name [operands]

or

[label] data-type, directive-name [operands]

The label, if present, is assigned the value of the current program or data location counter depending on which directive follows it. If a directive has several labels, all but the last are always assigned the value of the current program location counter.

The data type specifiers used for directives are the same as those used for instructions. The directive names are not reserved symbols.

The operands, if any, are separated by commas and have different formats for each individual directive.

2.5.1 *Declaration and Definition Directives*

2.5.1.1 MODULE and ENDMODULE

A NORD-500 assembly program consists of one or more modules which are delimited by MODULE and ENDMODULE. The format is:

```
MODULE [module-name [',' priority [',' language-code]]]
          .
          .
          .
          .
statements
          .
          .
          .
          .
ENDMODULE [module-name]
```

The module-name, which may be any legal identifier, is included in the page heading of the assembly listing. If specified, the name in the ENDMODULE directive must correspond to that in the matching MODULE directive. Except for these two functions the module-name is ignored by the assembler.

If specified, the priority must be an integer constant in the range 0-255. This value is output to the object code as the first of the two data bytes following the BEG control byte. The default value is zero.

The third parameter, language-code, is output as the second of the two data bytes following the BEG control byte. It must be an integer constant in the range 0-255. Values are: 0, assembly code; 1, FORTRAN; 2, PLANC. The default value is zero.

2.5.1.2 IMPORT-P and IMPORT-D

These two directives are used to make external data accessible within the current module. The format is:

```
IMPORT-P  identifier  $ ("," identifier)

IMPORT-D  identifier  $ ("," identifier)
```

An identifier which is mentioned in an IMPORT directive must not be defined in the current module. IMPORT-P is used to import program addresses (entry points) while IMPORT-D is used to import data addresses.

2.5.1.3 EXPORT

This directive is used to make addresses defined in the current module accessible to other modules. The format is:

EXPORT identifier $ ("," identifier)

An identifier that is mentioned in an EXPORT directive must be defined in the current module. Both program addresses and data addresses can be EXPORTed.

2.5.1.4 MAIN

The MAIN directive, which has the format:

MAIN identifier

specifies the main entry point of a program. The identifier must be defined as a program address in the current module. The identifier need not be EXPORTed. Only one main entry point can be specified.

2.5.1.5 LIB

The LIB directive has the format:

LIB identifier $ ("," identifier)

The current module will be loaded only if one or more of the identifiers mentioned in a LIB directive is undefined (in the loader table). Otherwise the entire module is skipped. Both program addresses and data addresses may be used as library symbols.

2.5.1.6 ALIAS

The ALIAS directive has the form:

identifier ":" ALIAS string-valued-expression

This directive defines the external representation of the symbol, i.e., the string which is output to the object stream. The use of this directive is to generate names that are syntactically illegal in the NORD-500 assembly language but are used by other language processors (e.g., operator names in PLANC). It can also be used to generate names which the user of other language processors is unable to duplicate. For example:

```
                ROUTINE CLOSE
CLOSE:          ALIAS '+ + +CLOSE'
CLOSE:          ENTD
                  .
                  .
                  .
                  .
```

2.5.1.7 ROUTINE and ENDROUTINE

A subroutine starts with a ROUTINE directive and ends with an ENDROUTINE directive. The ROUTINE directive is followed by a list of entry points. The entry points will be global labels while all other symbols defined within a ROUTINE – ENDROUTINE pair will be local to the subroutine. A local symbol cannot have the same name as a global symbol. The ROUTINE and ENDROUTINE directives do not generate any code. The ROUTINE and ENDROUTINE directives may not be nested. For an example of a subroutine refer to Appendix E.

2.5.1.8 STACK and ENDSTACK

These directives are used to declare data in the form of a stack entry. Data declared this way may be addressed through the B-register. A stack declaration can have one of the two forms:

```
[label]        STACK  FIXED
                 .
                 .
                 .
                 .
               data allocation directives
                 .
                 .
                 .
                 .
               ENDSTACK
or
               STACK
                 .
                 .
                 .
                 .
               data allocation directives
                 .
                 .
                 .
                 .
               ENDSTACK
```

The first form is used for data allocated statically in the data memory, while the second form is used for data allocated dynamically on a stack. The first form allows initialization of data, while the second form does not.

The optional label is assigned the address of the first byte and is used when referring to the stack block (e.g., in the ENTM and ENTF instructions).

A label occurring inside the stack definition is assigned an absolute value corresponding to the displacement from the start of the stack block currently being defined. This displacement is initialized to 20, leaving 20 bytes (5 words) for the stack header.

The first five words constitute the stack header. These words may be accessed by the following standard names.

PREVB	Saved B-register
RETA	Saved return address
SP	Stack pointer (next B)
AUX	System cell
NARG	Number of arguments supplied in call

If FIXED is specified, these words are initialized to zero at load time.

The stack location counter (address relative to the start of the current stack block) can be referenced as #SCLC. When referenced outside a stack definition #SCLC holds the size of the last stack block defined, thus it can be used directly as the "stack demand" argument in, for example, ENTS.

An example of a routine using dynamically allocated data can be found in Example 2. The following is an example of a routine using statically allocated local variables:

```
            ROUTINE CRLFX
CRLSS:      STACK FIXED
LINENO:     W DATA 1
            ENDSTACK
CRLFX:      ENTFN ENTF CRLSS, 0
            BY COMP2 B.LINENO, 72; IF < GO CR1
            CALLG NEWPAGE, 0; W SET1 B.LINENO; RET
CR1:        CALLG NEWLINE, 0; W INCR B.LINENO; RET
            ENDROUTINE
```

2.5.1.9 RECORD and ENDRECORD

RECORD and ENDRECORD are similar to STACK and ENDSTACK except that no stack header is allocated. Therefore, the displacement of the first variable is zero. Data declared with RECORD and ENDRECORD may be accessed through the R-register. The symbol #RCLC is called the record location counter and is used in the same way as #SCLC is used with STACK and ENDSTACK.

A record definition may occur inside a stack definition and vice versa. Stack and record definitions may not, however, be nested.

Example 1, Fixed Record:

```
RLOC:        RECORD  FIXED
RX1:         W  DATA  1,2
RX2:         DESC  10, LXX1
             ENDRECORD
                 .
                 .
             R: = ADDR(RLOC)
             W1: = R.RX1
             W2: = DESC(R.RX2)(R1)
                 .
                 .
```

Example 2, Symbol Table Element:

```
             RECORD
INAME:       W  BLOCK  1
ITYPE:       W  BLOCK  1
ISCOPE:      W  BLOCK  1
IMISC:       W  BLOCK  1
             ENDRECORD
                 .
                 .
XLOOP:       R: = B.ELEMENT
             W  COMP2  R.INAME,B.SNAME
             IF = GO  FOUND
             W  ADD2  B.ELEMENT, #RCLC
             GO  XLOOP
                 .
                 .
```

2.5.1.10 EQU and SEQU

These directives are used to assign a value to an identifier. They have the form:

identifier ":" EQU expression
identifier ":" SEQU string-valued-expression

For both directives the expression in the argument field must be evaluatable in pass one.

EQU assigns the value in the argument field to the identifier in the label field. The identifier gets the same type as the expression value.

SEQU is similar to EQU except that it always performs a string assignment, while EQU converts a string into an integer constant before the assignment is performed.

Identifiers defined with EQU or SEQU cannot be redefined.

Examples:

```
INT1:    EQU 101B            % INT1 GETS VALUE 101B
INT2:    EQU 'A'             % INT2 GETS VALUE 101B
PI:      EQU 3.1415926536    % DOUBLE PRECISION REAL

STR1:    SEQU 'A'            % STRING VALUE: A
```

2.5.2 *Data Allocation Directives*

2.5.2.1 BLOCK

The BLOCK directive, which has the format:

    [label] data-type BLOCK expression

allocates a block of data memory. The expression in the argument field specifies the size of the block in units of the data-type. All data-types are valid. The block is initialized to all zeros at load time.

If this directive is used in stack or record definition without the FIXED attribute, no memory is allocated, but the #SCLC or #RCLC is updated to reflect the amount of space needed at runtime.

The expression in the argument field must result in an absolute value and it must be evaluatable in pass one.

2.5.2.2 DATA and PROG

These directives are used to assemble data constants in the data memory (DATA) or the program memory (PROG). The format is:

```
[label] data-type  DATA  expression  $ ("," expression)
[label] data-type  PROG  expression  $ ("," expression)
```

All data types are valid. However, two special cases arise: BI DATA (or BI PROG) and BY DATA (or BY PROG). BY DATA is special only when an argument is a string valued expression.

BI DATA allocates memory in units of bytes and inserts the specified bits starting with the most significant bit (bit 7). Unused bits are set to zero. For example:

```
BI  DATA 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1
```

causes the two bytes 311B and 240B to be assembled in the data memory.

When BY DATA operates on an argument which represents a string, this string is not converted to an integer value but assembled byte for byte into the memory.

For Example:

```
BY  DATA  'NORD-500 ASSEMBLER'
BY  DATA  15B, 12B, 15B, 12B, '$'        % CR-LF, CR-LF, $
```

2.5.2.3 DESC

The format of the DESC directive, which is used to allocate a two word array descriptor, is:

[label] DESC [expression "," expression]

The first and second expression corresponds to the first and second word of the NORD-500 hardware array descriptor. If the expressions are omitted, two words, which are initialized to zero at load time, are allocated in the data memory.

If this directive is used in a stack or record definition without the FIXED attribute, the two expressions must not be specified.

When used without arguments, the DESC directive is equivalent to W BLOCK 2 or W DATA 0, 0 but may be preferred if the allocated space is to be used for descriptor storage.

2.5.2.4 ARRAY and STRING

These directives, which have the format:

[label] data-type ARRAY expression
[label] STRING expression

allocate a block of data memory immediately preceded by a descriptor. The ARRAY directive can be described in terms of the DATA and BLOCK directives as follows:

```
[label]  W DATA expression, #DCLC+4
         data-type BLOCK expression
```

All data types are valid. The block is initialized to zero at load time.

The directive STRING is equivalent to BY ARRAY. This form may, however, be preferred when used with the NORD-500 string instructions.

The expression in the argument field must evaluate to an absolute value and be evaluatable in pass one.

2.5.2.5 ARRAYDATA and STRINGDATA

These directives, which have the format:

```
[label] data-type  ARRAYDATA  expression $ ("," expression)
[label] STRINGDATA  expression  $ ("," expression)
```

are used to assemble constants in the form of arrays into the data memory. The data constants are assembled in the same way as for DATA. The block of constants is, however, preceded by a descriptor with the correct length information filled in. All data types are valid.

The directive STRINGDATA is equivalent to BY ARRAYDATA.

Example:

```
W ARRAYDATA 1, 2, 3, 4, 5, 6
```

is equivalent to:

```
DESC 6, #DCLC+4
W DATA 1, 2, 3, 4, 5, 6
```

2.5.3 *Location Counter Control Directives*

2.5.3.1 ORG-P and ORG-D

These directives set absolute origin in the program memory (ORG-P) or the data memory (ORG-D). They have the form:

```
[label] ORG-P [expression ]
[label] ORG-D [expression ]
```

The expression in the argument field must evaluate to an absolute value. It must be evaluatable in pass one. If present, the label in the label field is assigned the same value as the expression in the argument field.

If no argument is given, then relative assembly is resumed at the last relative address before absolute mode was entered.

2.5.3.2 BOUND-P and BOUND-D

The format of these directives is:

[label] BOUND-P expression
[label] BOUND-D expression

The expression in the argument field must result in an absolute value which is a power of two. The program location counter (#PCLC) for BOUND-P, or the data location counter (#DCLC) for BOUND-D is set to the next multiple of the value in the argument field. If the location counter already has a value which is a multiple of the value in the argument field, no action is taken.

These directives operate only on the assembly location counters. Therefore, if they are not used together with the ORG directive, the module must be loaded starting at a multiple of the maximum boundary size used in the module in order to ensure correct operation.

2.5.4 *Miscellaneous Directive*

MESSAGE

The specified message will be output by the loader when the object file is loaded. Message has the form:

MESSAGE expression

The expression in the argument field must evaluate to an absolute value and be evaluatable in pass one.

2.6 *COMMANDS*

A command consists of a dollar sign ($) followed by a command name. Command names are not reserved identifiers. Command parameters have different formats and are described for each particular command. See Appendix B for a summary of commands.

2.6.1 *Listing Control Commands*

2.6.1.1 $LIST and $NOLIST

The listing options which can be specified interactively with the LIST and NO-LIST commands (refer to Section 3.5) can be specified in the text of an assembly program through the $LIST and $NOLIST commands. Refer to Section 3.5 for a description of the argument format and each individual listing option.

2.6.1.2 $TITLE

The title command is used to define a title string which will be included in the page headings of the assembly listing. The title is specified as a string constant following the $TITLE command.

For Example:

```
$TITLE 'BASIC I/O ROUTINES'
```

causes the specified string to be included in the second line of the page heading, after the module name (if any).

2.6.1.3 $EJECT and Form Feed

A page eject in the assembly listing can be obtained in several ways:

- After a specified number of lines have been printed on the same page, the assembler automatically performs a page eject. The page size can be specified with the LINES command (see Section 3.3).

- If a source line contains one or more form feeds (ASCII 14B) a page eject is issued before this line is listed. If used within a macro definition, a form feed character causes a page eject. A page eject is not performed when the macro is expanded.

- The command $EJECT, which has no arguments, causes a page eject to be issued. Used within a macro definition the $EJECT command is ignored, but the page eject is performed when the macro is expanded.

2.6.2 *Conditional Assembly Commands*

2.6.2.1 $IF, $ELSIF, $ELSE and $ENDIF

Conditional assembly commands give you the possibility to conditionally include or ignore blocks of source code in the assembly process.

The general form of a conditional block is:

```
$IF EXPRESSION         % START OF CONDITIONAL BLOCK
  .
  .
  .
$ELSIF EXPRESSION      % ZERO OR MORE $ELSIF COMMANDS
  .
  .
  .
$ELSE                  % OPTIONAL $ELSE COMMAND
  .
  .
  .
$ENDIF                 % END OF CONDITIONAL BLOCK
```

The expression which is the argument of the $IF and $ELSIF command is evaluated. If the resulting value is nonzero (TRUE), the source code between the command and the next $ELSIF, $ELSE or $ENDIF command is assembled. If the resulting value is zero (FALSE) the source code is ignored.

The source code included between a $IF command and its required associated $ENDIF command is defined as a conditional block. A conditional block may contain any number (including zero) of $ELSIF commands, but only one $ELSE command. No $ELSIF command may appear between a $ELSE command and its matching $ENDIF command. Only the source code following the first satisfied condition in a conditional block is assembled.

Conditional blocks may be nested to any desired level.

2.6.3 *Source File Library Commands*

2.6.3.1 $INCLUDE and $SECTION

The format of the $INCLUDE command is:

    $INCLUDE file-name ["," section-name]

where section-name is syntactically equivalent to file-name. If only the file-name is present, the text of the specified file is included in the source text.

If the section-name is present, only the named section, located on the specified file, is included. Sections are defined by means of the $SECTION command which has the format:

    $SECTION section-name

The text which comprises the section starts with the statement following the $SECTION and ends with the next $SECTION or $EOF command (or at end-of-file). If the specified section-name does not exist on the specified file, no text is included.

If a file containing section definitions is included as a whole (no section-name specified in the $INCLUDE command), the section definitions are ignored.

2.6.4 *Macro Definitions and Macro Calls*

2.6.4.1 $MACRO

The first statement of a macro definition must be a $MACRO command. The $ MACRO command is of the form:

```
$MACRO macro-name ["(" [formal-parameters] ")"]
```

where macro-name is the name of the macro. The macro-name is any legal identifier. The name cannot be used as a label anywhere else in the program. Macros are not local to modules but exist throughout the entire file on which they are defined. Formal-parameters are a list of identifiers separated by commas. These identifiers can be used elsewhere in the program without conflicts of definition. When a formal-parameter is referenced in the macro body it must be enclosed within double quotes (e.g., "PAR1").

2.6.4.2 $ENDMACRO

The final statement of every macro definition must be a $ENDMACRO command of the form:

```
$ENDMACRO [macro-name]
```

where macro-name is an optional argument and is the name of the macro being terminated by the statement. If specified, the name in the $ENDMACRO command must correspond to that in the matching $MACRO command. Specification of the macro-name in the $ENDMACRO command permits the assembler to detect missing $ENDMACRO commands or improperly nested macro definitions.

An example of a macro definition is shown below:

```
$MACRO  CHECK (GVX, LABX)
      W1: = IND (B.GVARIDX)
      W COMP2  IND (B.GVAR) (R1), "GVX"
      IF >< GO  "LABX"
$ENDMACRO  CHECK
```

2.6.4.3 $EXITMACRO

In order to implement alternate exit points from a macro (particularly nested macros), the $EXITMACRO command is provided. $EXITMACRO terminates the current macro as though a $ENDMACRO command was encountered. SEXITMACRO bypasses the complication of conditional nesting and alternate paths. For example:

```
$MACRO XMK (NN, AA, BB)
  .
  .
$IF "NN"=0                % START OF CONDITIONAL BLOCK
  .
  .
$EXITMACRO                % EXIT DURING CONDITIONAL BLOCK
$ENDIF                    % END OF CONDITIONAL BLOCK
  .
  .
$ENDMACRO                 % NORMAL MACRO EXIT
```

In an assembly where NN = 0, the $EXITMACRO command terminates the macro expansion.

When macros are nested, $EXITMACRO causes an exit to the next higher level.

2.6.4.4 Macro Calls

A macro must be defined prior to its first reference. A macro call may occur anywhere an instruction, directive, or command is legal. Macro calls are of the form:

macro-name ["(" [actual-parameters] ")"]

where macro-name is the name of a macro defined in a preceding $MACRO command. The actual-parameters are a list of values, separated by commas, which replace the formal-parameters in the macro definition.

If an actual parameter contains a separating character (e.g., comma or right parenthesis) it can be enclosed within angle brackets (< >).

For Example:

CHECK (<IND (B.XDJ)>, XLABEL)

This call causes the general operand IND (B.XDJ) to replace all occurrences of "GVX" in the macro CHECK (defined above).

An exclamation mark (!) can be used as an escape character. It is used primarily to pass an angle bracket as part of an actual parameter. To pass an exclamation mark write !!.

2.6.4.5 Macro Nesting

Nested macro calls, where the expansion of one macro contains one or more macro calls, causes one set of angle brackets to be removed from an argument with each level of nesting.

Recursive macro calls are permitted. As an example, consider the following pair of macros which evaluate the factorial function (as a constant value):

```
$MACRO FACT(N)
      XFACT("N",1)
$ENDMACRO FACT

$MACRO XFACT(N,HOLD)
$IF "N" = 0
      W DATA "HOLD"
$ELSE
      XFACT("N" - 1,<("N")*("HOLD")>)
$ENDIF
$ENDMACRO XFACT
```

Note the use of parentheses and angle brackets in the recursive call on XFACT. The parentheses are necessary in order to obtain the correct value because the argument is passed as an expression, not as an evaluated value. The angle brackets must be used because the expression contains right parenthesis. An exclamation mark in front of each right parentheses is not sufficient because the argument "HOLD" contains right parentheses.

If macro definitions are nested (that is, a macro definition is entirely contained within the definition of another macro) the inner macro is not defined as a callable macro until the outer macro has been called and expanded.

2.6.4.6 Special Forms: #NARG, "LABEL" and "MNO"

If more arguments appear in the macro call than in the macro definition, the excess arguments are ignored. If fewer arguments appear in the macro call than in the macro definition, missing arguments are assumed to be null (consist of no characters).

The intrinsic function #NARG (see Section 2.4.3) can be used to test for the presence or absence of an argument

If a label is placed in the label field of a macro call, this label is not defined before the call, but is passed as a special kind of argument. The label can be referenced by the special formal parameter name "LABEL" which expands to the label name followed by a colon (:). This enables the user to determine exactly where in the macro body the label definition is to take place.

For Example:

```
$MACRO  BES (TYPE, SIZE)
     "TYPE"  BLOCK  "SIZE"
"LABEL"  BY  BLOCK  0
$ENDMACRO  BES
```

is one possible definition of the common macro BES (Block Ending Symbol). A typical call might be:

```
BLK1:  BES (BY, 103)
```

To create unique symbols in a macro expansion the special form "MNO" (macro number) can be used. "MNO" expands to a five digit decimal number which is the serial number of the current macro call. To provide several unique symbols within the same macro "MNO" is concatenated with different strings. If the first character of the generated symbol is a question mark, the symbol will be invisible, i.e., not listed in the symbol table dump. Symbols generated in this way are not different from other symbols used in the assembler. They may be referenced outside the macro if desired. As an example of generated symbols consider:

```
$MACRO  GOIFWRONG
      W  COMP2  B.EXPECTED,  B.ACTUAL
      IF  =  GO  ?A"MNO"
      W  MOVE  B.EXPECTED,  FPAR1
      W  MOVE  B.ACTUAL,  FPAR2
      GO  ERRFATAL:H
?A"MNO"
$ENDMACRO
```

The second time this macro is called the label ?A00002 is generated.

2.6.5 *Miscellaneous Commands*

2.6.5.1 $PACK and $ALIGN

These commands control the packing of data allocated in the data memory.

If $ALIGN is specified, half-word data is aligned on half word boundaries (0, 2, 4, ...) and word data is aligned on word boundaries (0, 4, 8, ...). Descriptors (e.g., in ARRAY and ARRAYDATA) are also aligned on word boundaries.

If $PACK is specified, no alignment is performed.

The default mode is $ALIGN.

2.6.5.2 $EOF

The $EOF command signals the end of the source file or end of included file (see $INCLUDE). The effect of this command is simulated when an end of file indication is received from the file system.

3 ASSEMBLER OPERATING PROCEDURE

To start the assembler from SINTRAN III one types the following:

@N500-ASSEMBLER [<source> <list> ... etc.] cr

NORD-500 ASSEMBLER 2.5, 19 November 1979

$

The command processor is now ready to accept commands. Whenever the command processor expects the operator to enter a command, it outputs a dollar sign ($). A command consists of a command name followed by zero or more parameters. Several commands, along with all required parameters, may be written on the same line.

The command name consists of one or more parts separated by hyphens ("-"). Each part of the command name may be abbreviated as long as the command can be distinguished from all other command names.

The standard editing characters are available while typing commands.

The collection of parameters is done in a standarized way as follows:

- Parameters are separated by either a comma or any number of spaces or a combination of comma and spaces.
- Parameters may be null in which case a default value is assigned.
- When a parameter is missing (as opposed to null) it is asked for, and the command processor expects you to supply the required parameter plus more parameters if you wish.
- When a parameter syntax error is detected, an error message is printed and the parameter is asked for.
- Excess parameters are ignored.

Commands can be given directly to the SINTRAN III command processor by preceding them with an @ sign. In this case commands to the local command processor following the SINTRAN III command are ignored.

3.1 *HELP <command name>*

The HELP command lists available commands on the terminal. Only those commands that have <command name> as a subset are listed. If <command name> is null then all available commands are listed.

3.2 *EXIT*

The EXIT command returns control to the SINTRAN III command processor.

3.3 *LINES <lines per page>*

This command enables the user to specify the number of lines per page on the assembly listing.

3.4 *ASSEMBLE <source file> <list file> <object file>*

This command assembles the specified <source file> with listing on <list file> and object output to <object file>. If no list file is specified, no listing is produced, but error messages are printed on the terminal. If no object file is specified, no object output is produced. The default file types are: :SYMB, :LIST, and :NRF.

3.5 *LIST <list directives> ...*
NO-LIST <list directives> ...

These commands are used to set/reset various internal flags which control the format and extent of the assembly listing. A LIST command with an empty parameter will cause the listing mode to be set to its default (initial) value. A NO-LIST command with an empty parameter will cause all output, except error messages, to be suppressed.

The following are legal list directives:

HELP <command name>

Lists available list directives on the terminal. Only those list directives that have <command name> as a subset are listed. If <command name> is null then all available list directives are listed.

GLOBAL-SYMBOLS

Controls the listing of the "global symbols" part of the symbol table. Global symbols are those symbols not defined within any ROUTINE – ENDROUTINE pair. Default is LIST.

LOCAL-SYMBOLS

Controls the listing of the "local symbols" part of the symbol table. A symbol is called local if it is defined within a ROUTINE - ENDROUTINE pair and is not mentioned as an entry point in a ROUTINE statement. Default is NO-LIST.

LOCATION-COUNTER

Controls the listing of the assembly location counter field. The location counter is listed as an eleven digit octal number. Default is LIST.

GENERATED-CODE

Controls the listing of the generated binary code. The generated code will be listed as several fields containing octal numbers. Default is NO-LIST.

MACRO-EXPANSIONS

Controls the listing of macro expansions. With this directive the macro expansions are listed out. Default is NO-LIST.

CROSS-REFERENCE-TABLE

Controls the generation of and printing of an alphabetically sorted cross-reference table at the end of the assembly. The cross-reference table consists of all the user defined symbols and for each of them a list of line numbers. The number of a line where the symbol is defined is followed by an asterisk (*). Default is NO-LIST.

3.6 *PRINT-MACRO <macro name> <output file>*

This command prints the currently defined macros on the specified output file. Parameters are named P1, P2, etc. The default output file is the terminal and the default file type is :SYMB. If <macro name> is null, all macros are printed. Otherwise only the specified macro is printed.

3.7 *TABLE-SIZES <size parameter>*

This command enables the user to change the size of any of the tables allocated in the assembler's dynamic work area. If the new table size is accepted, the old size is printed on the terminal and the assembler is initialized.

The possible size parameters are listed below.

HELP<command name>

> Lists available size parameters on the terminal. Only those size parameters that have <command name> as a subset are listed. If <command name> is null then all available size parameters are listed.

MACRO-TABLE<macro table size>

> Specifies the size of the macro table. This area is used for storing macro bodies and for the macro/include stack.

SOURCE-LINE-BUFFER<source line buffer size>

> This command can be used to avoid the SOURCE LINE BUFFER TOO SMALL error message.

OBJECT-CODE-BUFFER<object code buffer size>

> This command can be used to avoid the OBJECT CODE BUFFER TOO SMALL error message.

4 ASSEMBLY LISTING FORMAT

The assembly listing consists of three parts for every module: the assembled program, the symbol table of the assembly and an alphabetically sorted cross-reference table. Every page of the listing starts with a page heading. A description of the format follows. Appendix E contains an example of the assembly listing format.

4.1 *PAGE HEADING*

The first four lines of a page constitute the page heading. Before the heading lines are printed, the listing device is advanced to a new page. If the listing device is the terminal, a blank line is printed instead of advancing it to the next page. The heading consists of the following fields:

— Assembler name and version number

— Current date and time

— Page number

— The name of the module currently being assembled followed by the title string if a title has been specified

— Two blank lines

4.2 PROGRAM LISTING

The program listing consists of several fields on each line. If an instruction has more than one operand specifier or if several instructions are written on the same source line, then the generated code may require several lines on the listing. The following description assumes that all listing options are enabled. Refer to Section 3.5 for an explanation of the listing options.

– Source line number

  This field is blank if the line was not read from the source input file.

– Current location counter

  This field is blank if the operation does not change the location counter or if the line is a binary extension line, i.e., the location counter is only printed at the start of each instruction. The location counter is printed as an eleven digit octal number. It is preceded by a letter specifying which of the location counters is printed: P (Program location counter); D (Data location counter); S (Stack location counter); R (Record location counter).

– Generated code

  This field is divided into several subfields: operation code (8 or 16 bits), prefix operand code number 1 (if ALT, 8 bits), prefix operand code number 2 (if DESC, 8 bits), operand code (if general operand, 8 bits) and address/displacement (all types except S). If an imported quantity is referenced, it is printed in symbolic form plus the displacement.

– Source code

– Error messages

  If one or more errors are detected in a line, the error message(s) are output following the line in error. The error message is preceded by four asterisks ("*****"), the name of the current source file, the last label encountered and the displacement (in lines) since the last label. At the end of the entire listing the following two lines are printed:

  – Number of errors detected during the assembly
  – CPU time used.

4.3 *SYMBOL TABLE*

When listing the symbol table, the title is set to "SYMBOL TABLE". The symbols are listed in alphabetical order. The fields are as follows:

- Symbol name (maximum 16 characters)

- Symbol type. The types are:

  1 U = Undefined
  2 W = Integer (Word), D = Double real, S = String
  3 A = Absolute
  4 P = Program address
  5 D = Data address
  6 M = Main entry point
  7 I = Imported
  8 E = Exported
  9 L = Library symbol

- Symbol value. The value is given in the following formats, depending upon the data type:

  Integer: Eleven digit octal number
  Real: Two eleven digit octal numbers, separated by space
  String: A character string

- If the symbol has an alternative name (an ALIAS), this name is printed following the value.

4.4 *CROSS-REFERENCE TABLE*

When listing the cross-reference table the title is set to "CROSS-REFERENCE TABLE". The cross-reference table is an alphabetically sorted list of all symbols used in the program. Each symbol is followed by a list of line numbers. The line numbers of the lines where the symbol is defined are followed by an asterisk (*). If a symbol name is used more than once (as local symbol), a separate list of line numbers is given for each version of the symbol.

APPENDIX A

SUMMARY OF DIRECTIVES

MODULE [module-name [″,″ priority [″,″ code]]]	Define start of module. The default value for priority and code is zero.
ENDMODULE [module-name]	Define end of module. The name must be the same as in the matching MODULE.
IMPORT-P identifier-list	Import external routines
IMPORT-D identifier-list	Import external data.
EXPORT identifier-list	Export internal routines or data.
MAIN identifier	Define main entry point.
LIB identifier-list	Define library symbols.
identifier ″:″ ALIAS string	Define alternative external representation.
ROUTINE identifier-list	Start of subroutine with local symbols.
ENDROUTINE	End of subroutine.
STACK [FIXED]	Start of stack definition.
ENDSTACK	End of stack definition.
RECORD [FIXED]	Start of record definition.
ENDRECORD	End of record definition.
data-type BLOCK size	Allocate block in data memory.
data-type DATA data element list	Allocate constant data in data memory
data-type PROG data element list	Allocate constant data in program memory.
DESC [limit ″,″ address]	Allocate descriptor
data-type ARRAY size	Allocate storage preceded by array descriptor.

STRING size	Same as BY ARRAY.
data-type ARRAYDATA data-element-list	Allocate constant data preceded by array descriptor.
STRINGDATA data-element-list	Same as BY ARRAYDATA.
ORG-P origin	Set absolute program origin.
ORG-D origin	Set absolute data origin.
BOUND-P base	Advance program location counter to next multiple of base.
BOUND-D base	Advance data location counter to next multiple of base.
MESSAGE	Output message string to object code

APPENDIX B

SUMMARY OF COMMANDS

$LIST listing-options	Enable listing options.
$NOLIST listing-options	Disable listing options.
listing-options: GLOBAL-SYMBOLS LOCAL-SYMBOLS LOCATION-COUNTER GENERATED-CODE CROSS-REFERENCE-TABLE	 Global symbols in symbol table. Local symbols in symbol table. Location counter field. Code fields. Cross-reference table.
$TITLE title-string	Define title string. Also performs page eject.
$EJECT	Page eject.
$IF expression	Conditional assembly.
$ELSIF expression	0 = FALSE, ><0 = TRUE.
$ELSE	Optional $ELSE command.
$ENDIF	End of conditional block.
$INCLUDE file-name ["," section-name]	Include source file.
$SECTION section-name	Defina section.
$MACRO macro-name ["(" parameters")"]	Start of macro definition.
$ENDMACRO [macro-name]	End of macro defintion.
$EXITMACRO	Immediate macro exit.
$PACK	Pack data elements.
$ALIGN	Align data elements.
$EOF	End-of-file.

APPENDIX C

RESERVED SYMBOLS

The symbols listing in this appendix are reserved symbols and may not be redefined by the user.

B	D1	ALT	ADDR	#DATE	#ZEROP
D	D2	AND	DESC	#DCLC	#ZEROD
F	D3	AUX	NARG	#LOG2	
H	D4	BI1	RETA	#NARG	
R	F1	BI2		#NCHR	
S	F2	BI3		#PCLC	
W	F3	BI4		#RCLC	
	F4	BY1		#SCLC	
	H1	BY2		PREVB	
	H2	BY3		SHIFT	
	H3	BY4			
	H4	IND			
	OR	MOD			
	R1	NOT			
	R2	XOR			
	R3				
	R4				
	SP				
	W1				
	W2				
	W3				
	W4				

APPENDIX D

INTRINSIC CONSTANTS AND FUNCTION SUMMARY

Constant	*Value*	*Description*
PREVB	0	Saved B-register
RETA	4	Saved return address
SP	8	Stack pointer
AUX	12	System cell
NARG	16	Number of arguments supplied in call
#ZEROP	0	Program address zero
#ZEROD	0	Data address zero

Function	*Description*
#PCLC	Program location counter
#DCLC	Data location counter
#SCLC	Stack location counter
#RCLC	Record location counter
#NCHR (string)	Number of characters in string
#NARG	Number of arguments in current macro call
#DATE	Current date and time (double word)
#LOG2 (integer)	Logarithm to base 2

APPENDIX E

MODULE EXAMPLE LISTING

This appendix shows the output listing from Example 1. The following options were enabled during the assembly.

LOCATION-COUNTER
GENERATED-CODE
GLOBAL-SYMBOLS
LOCAL-SYMBOLS
CROSS-REFERENCE-TABLE

```
NORD-500 ASSEMBLER 2.5      WEDNESDAY  26 MARCH 1980   1504:22      PAGE 1
MODULE EXAMPLE.

                                                     MODULE EXAMPLE        %  NAME OF MODULE
                                                     MAIN START            %  SPECIFIES MAIN ENTRY POINT

                                                     ROUTINE LNG
                                              %
                                              %      COMPUTE:  PAR3 = SQRT( PAR1**2 + PAR2**2 )
                                              %
  D 00000000000                               DSTK:  STACK FIXED           %  START OF STACK DEFINITION
  S 00000000024                               APAR1: W BLOCK 1             %  ADDRESS OF 1. PARAMETER
  S 00000000030                               APAR2: W BLOCK 1             %  ---- " --- 2. -- " --
  S 00000000034                               APAR3: W BLOCK 1             %  ---- " --- 3. -- " --
                                                     ENDSTACK
                                              %
  P 00000000000    335            00000000000 LNG:   ENTF DSTK             %  ENTER SUBROUTINE WITH
                                                                           %  FIXED DATA AREA BEGINNING
                                                                           %  AT 'DSTK'.

  P 00000000005    020     305 024                   F1 := IND(B.APAR1)    %  LOAD 1. PARAMETER
  P 00000000010    160     320                       F1 * F1               %  SQUARE
  P 00000000012    021     305 030                   F2 := IND(B.APAR2)    %  LOAD 2. PARAMETER
  P 00000000015 176361     321                       F2 MULAD F2,F1        %  SQUARE AND ADD
                           320
  P 00000000021 176325     321                       F2 SQRT F2            %  TAKE SQUAREROOT
  P 00000000024    045     305 034                   F2 =: IND(B.APAR3)    %  STORE IN 3. PARAMETER
  P 00000000027    200                               RET                   %  RETURN
                                                     ENDROUTINE
                                              %
                                              %      MAIN PROGRAM
                                              %
  D 00000000040                               STK:   STACK FIXED           %  START OF STACK DEFINITION
  S 00000000024                   10050000000 A:     F DATA 3.0            %  DEFINE A AS 3.0
  S 00000000030                   10060000000 BB:    F DATA 4.0            %  DEFINE BB AS 4.0
  S 00000000034                               C:     F BLOCK 1             %  DECLARE SPACE FOR
                                                                           %  ONE REAL VARIABLE.
                                                     ENDSTACK
                                              %
                                              %      INITIATE STACK AREA WITH MAIN PROGRAM STACK
```

```
37                                                    %      FRAME BEGINNING AT 'STK', LENGTH #SCLC, AND
38                                                    %      TOTAL STACK DEMAND OF 100.
39                                                    %         #SCLC IS AN INTRINSIC FUNCTION GIVING THE
40                                                    %      SIZE OF THE STACK FRAME IN THE LAST PRECEEDING
41                                                    %      DEFINITION.
42                                                    %
43 P 00000000030    334           00000000040 START: INIT STK,#SCLC,100
                              315 040
                              315 144
44                                                    %
45                                                    %      CALL ROUTINE WITH 3 LOCAL PARAMETERS A, BB
46                                                    %      AND C.
47                                                    %
48 P 00000000041    303           00000000000        CALL LNG,3,B.A,B.BB,B.C
                              003
                              105
                              106
                              107
49                                                    %
50                                                    %      "RETURN" FROM MAIN PROGRAM, I.E. STOP.
51                                                    %
52 P 00000000052    200                               RET
53
54 P 00000000053                                      ENDMODULE
```

```
NORD-500 ASSEMBLER 2.5       WEDNESDAY  26 MARCH 1980   1504:26       PAGE 2
SYMBOL TABLE

GLOBAL SYMBOLS

A                  W  A         00000000024
BB                 W  A         00000000030
C                  W  A         00000000034
LNG                W   P        00000000000
START              W   P M      00000000030
STK                W    D       00000000040


SYMBOLS LOCAL TO LNG

APAR1              W  A         00000000024
APAR2              W  A         00000000030
APAR3              W  A         00000000034
DSTK               W    D       00000000000

NORD-500 ASSEMBLER 2.5       WEDNESDAY  26 MARCH 1980   1504:26       PAGE 3
CROSS-REFERENCE TABLE

A                   30*   48
APAR1                9*   18
APAR2               10*   20
APAR3               11*   23
BB                  31*   48
C                   32*   48
DSTK                 8*   14
LNG                  4    14*   48
START                2    43*
STK                 29*   43

NORD-500 ASSEMBLER 2.5       WEDNESDAY  26 MARCH 1980   1504:27       PAGE 4


NO ERRORS DETECTED
```

APPENDIX F

ADDRESS CODES

NAME	SIZE	OPERATION	OCTAL LAYOUT	
LOCAL	:S	ea=(B)+d*4	1dd	
LOCAL	:B	ea=(B)+d	301	ddd
LOCAL	:H	ea=(B)+d	302	ddd ddd
LOCAL	:W	ea=(B)+d	303	ddd ddd ddd ddd
LOCAL P.I.	:B	ea=(B)+d+p*(Rn)	324+y	ddd
LOCAL P.I.	:H	ea=(B)+d+p*(Rn)	330+y	ddd ddd
LOCAL P.I.	:W	ea=(B)+d+p*(Rn)	334+y	ddd ddd ddd ddd
LOCAL INDIRECT	:B	ea=((B)+d)	305	ddd
LOCAL INDIRECT	:H	ea=((B)+d)	306	ddd ddd
LOCAL INDIRECT	:W	ea=((B)+d)	307	ddd ddd ddd ddd
LOCAL INDIRECT P.I.	:B	ea=((B)+d)+p*(Rn)	344+y	ddd
LOCAL INDIRECT P.I.	:H	ea=((B)+d)+p*(Rn)	350+y	ddd ddd
LOCAL INDIRECT P.I.	:W	ea=((B)+d)+p*(Rn)	354+y	ddd ddd ddd ddd
RECORD	:S	ea=(R)+d*4	2dd	
RECORD	:B	ea=(R)+d	311	ddd
RECORD	:H	ea=(R)+d	312	ddd ddd
RECORD	:W	ea=(R)+d	313	ddd ddd ddd ddd
PRE INDEXED	:B	ea=(Rn)+d	364+y	ddd
PRE INDEXED	:H	ea=(Rn)+d	370+y	ddd ddd
PRE INDEXED	:W	ea=(Rn)+d	374+y	ddd ddd ddd ddd
ABSOLUTE		ea=a	304	aaa aaa aaa aaa
ABSOLUTE P.I.		ea=a+(Rn)*p	340+y	aaa aaa aaa aaa
CONSTANT	:S	op=c	0cc	
CONSTANT	:B	op=c	315	ccc
CONSTANT	:H	op=c	316	ccc ccc
CONSTANT	:W	op=c	317	ccc ccc ccc ccc
CONSTANT	:F	op=c	317	ccc ccc ccc ccc
CONSTANT	:D	op=c	314	ccc ccc ccc ccc ccc ccc ccc ccc
REGISTER		op=(Rn)	320+y	
DESCRIPTOR		ea=A+p*(Rn)	360+y	<operand>
ALTERNATIVE			310	<operand>
NOT USED			300	

()	- Contents of
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
y	- 0,1,2 or 3 specifies the registers R1 to R4.
p	- p= 1/8 (bit), 1 (byte), 2 (half word), 4 (word), 4 (float), 8 (double float) operations. Post index scaling factor.
Rn	- Used to reference a register, n=1,2,3,4
B	- Base register
R	- Record register

APPENDIX G

ADDRESS CODE TABLE

	:S	:B	:H	:W	:F	:D	PREFIX
LOCAL	1dd	301	302	303			
LOCAL P.I.		324+	330+	334+			
LOCAL INDIRECT		305	306	307			
LOCAL INDIRECT P.I.		344+	350+	354+			
RECORD	2dd	311	312	313			
PRE INDEXED		364+	370+	374+			
ABSOLUTE				304			
ABSOLUTE P.I.				340+			
CONSTANT	0cc	315	316	317	317	314	
REGISTER	320+						
ADDRESS CODE PREFIXES:							
DESCRIPTOR							360+
ALTERNATIVE							310

APPENDIX H

INSTRUCTION LIST

ARITHMETICAL, LOGICAL, and DATA TRANSFER INSTRUCTIONS

Instruction Code octal value	assembly notation	name
176004+(n-1)	BIn :=	load bit
004+(n-1)	BYn :=	load byte
010+(n-1)	Hn :=	load halfword
014+(n-1)	Wn :=	load word
020+(n-1)	Fn :=	load float
024+(n-1)	Dn :=	load double float
176010	B:=	load local base
030	R:=	load record base
176014+(n-1)	BIn =:	store bit
034+(n-1)	BYn =:	store byte
176020+(n-1)	Hn =:	store halfword
040+(n-1)	Wn =:	store word
044+(n-1)	Fn =:	store float
050+(n-1)	Dn =:	store double float
176012	B=:	local base store
176011	R=:	record base store
176013	BI MOVE	move bit
031	BY MOVE	move byte
176024	H MOVE	move halfword
032	W MOVE	move word
033	F MOVE	move float
054	D MOVE	move double float
176030+(n-1)	BIn COMP	register bit compare
060+(n-1)	BYn COMP	register byte compare
176034+(n-1)	Hn COMP	register halfword compare
064+(n-1)	Wn COMP	register word compare
070+(n-1)	Fn COMP	register float compare
074+(n-1)	Dn COMP	register float compare
176025	BI COMP2	bit compare
055	BY COMP2	byte compare
176026	H COMP2	halfword compare
056	W COMP2	word compare
057	F COMP2	float compare
100	D COMP2	double float compare
101	BI TEST	bit test against zero
102	BY TEST	byte test against zero
103	H TEST	halfword test against zero
104	W TEST	word test against zero
105	F TEST	float test against zero
106	D TEST	double float test against zero

177010+(n-1)	BYn	NEG	byte register negate
177014+(n-1)	Hn	NEG	halfword register negate
220+(n-1)	Wn	NEG	word register negate
224+(n-1)	Fn	NEG	float register negate
224+(n-1)	Dn	NEG	double float register negate
177020+(n-1)	BIn	INV	bit invert register
177024+(n-1)	BYn	INV	byte invert register
177030+(n-1)	Hn	INV	halfword invert register
230+(n-1)	Wn	INV	word invert register
177420+(n-1)	Wn	INVC	word invert register with carry
177400+(n-1)	BYn	ABS	byte absolute value
177404+(n-1)	Hn	ABS	halfword absolute value
177410+(n-1)	Wn	ABS	word absolute value
177414+(n-1)	Fn	ABS	float absolute value
177414+(n-1)	Dn	ABS	double float absolute value
176064+(n-1)	BYn	+	byte add
176070+(n-1)	Hn	+	halfword add
124+(n-1)	Wn	+	word add
130+(n-1)	Fn	+	floating add
134+(n-1)	Dn	+	double float add
176074+(n-1)	BYn	-	byte subtract
176100+(n-1)	Hn	-	halfword subtract
140+(n-1)	Wn	-	word subtract
144+(n-1)	Fn	-	float subtract
150+(n-1)	Dn	-	double float subtract
176104+(n-1)	BYn	*	byte multiply
176110+(-1)	Hn	*	halfword multiply
154+(n-1)	Wn	*	word multiply
160+(n-1)	Fn	*	floating multiply
164+(n-1)	Dn	*	double float multiply
176114+(n-1)	BYn	/	byte divide
176120+(n-1)	Hn	/	halfword divide
170+(n-1)	Wn	/	word divide
174+(n-1)	Fn	/	float divide
350+(n-1)	Dn	/	double float divide
176027	BY	ADD2	byte add two arguments
176124	H	ADD2	halfword add two arguments
123	W	ADD2	word add two arguments
176126	F	ADD2	float add two arguments
176127	D	ADD2	double float add two arguments
176130	BY	SUB2	byte subtract two arguments
176131	H	SUB2	halfword subtract two arguments
340	W	SUB2	word subtract two arguments
176133	F	SUB2	float subtract two arguments
176134	D	SUB2	double float subtract two arguments
176135	BY	MUL2	byte multiply two operands
176136	H	MUL2	halfword multiply two operands
176137	W	MUL2	word multiply two operands
176140	F	MUL2	float multiply two operands

176141	D	MUL2	double float multiply two operands
176142	BY	DIV2	byte divide two arguments
176143	H	DIV2	halfword divide two arguments
176144	W	DIV2	word divide two arguments
176145	F	DIV2	float divide two arguments
176146	D	DIV2	double float divide two arguments
176147	BY	ADD3	byte add three arguments
176150	H	ADD3	halfword add three arguments
176151	W	ADD3	word add three arguments
176152	F	ADD3	float add three arguments
176153	D	ADD3	double float add three arguments
176154	BY	SUB3	byte sutract three operands
176155	H	SUB3	halfword subtract three operands
176156	W	SUB3	word subtract three operands
176157	F	SUB3	float subtract three operands
176160	D	SUB3	double float subtract three operands
176161	BY	MUL3	byte multiply three arguments
176162	H	MUL3	halfword multiply three arguments
176163	W	MUL3	word multiply three arguments
176164	F	MUL3	float multiply three arguments
176165	D	MUL3	double float multiply three arguments
176166	BY	DIV3	byte divide three arguments
176167	H	DIV3	halfword divide three arguments
176170	W	DIV3	word divide three arguments
176171	F	DIV3	float divide three arguments
176172	D	DIV3	double float divide three arguments
176040+(n-1)	BYn	MUL4	byte multiply with overflow
176044+(n-1)	Hn	MUL4	halfword multiply with overflow
176050+(n-1)	Wn	MUL4	word multiply with overflow
176054+(n-1)	BYn	DIV4	byte divide with remainder
176060+(n-1)	Hn	DIV4	halfword divide with remainder
176174+(n-1)	Wn	DIV4	word divide with remainder
176200+(n-1)	Wn	UMUL	word unsigned multiplication
177110+(n-1)	Wn	UDIV	word unsigned divide
177100+(n-1)	Wn	ADDC	word add with carry
177104+(n-1)	Wn	SUBC	word subtract with carry
204+(n-1)	BIn	CLR	bit register clear
204+(n-1)	BYn	CLR	byte register clear
204+(n-1)	Hn	CLR	halfword register clear
204+(n-1)	Wn	CLR	word register clear
210+(n-1)	Fn	CLR	float register clear
214+(n-1)	Dn	CLR	double float register clear
176205	BI	STZ	bit store zero
110	BY	STZ	byte store zero
111	H	STZ	halfword store zero
112	W	STZ	word store zero
113	F	STZ	float store zero
114	D	STZ	double float store zero

176206	BI	SET1	bit set to one
176207	BY	SET1	byte set to one
176210	H	SET1	halfword set to one
115	W	SET1	word set to one
107	F	SET1	float set to one
176211	D	SET1	double float set to one
176212	BY	INCR	byte increment
116	H	INCR	halfword increment
117	W	INCR	word increment
120	F	INCR	float increment
176213	D	INCR	double float increment
176214	BY	DECR	byte decrement
176215	H	DECR	halfword decrement
121	W	DECR	word decrement
176216	F	DECR	float decrement
176217	D	DECR	double float decrement
176714+(n-1)	BIn	AND	bit and register
176220+(n-1)	BYn	AND	byte and register
176224+(n-1)	Hn	AND	halfword and register
344+(n-1)	Wn	AND	word and register
176770+(n-1)	BIn	OR	bit or register
176230+(n-1)	BYn	OR	byte or register
176234+(n-1)	Hn	OR	halfword or register
240+(n-1)	Wn	OR	word or register
176774+(n-1)	BIn	XOR	bit exclusive or register
176240+(n-1)	BYn	XOR	byte exclusive or register
176244+(n-1)	Hn	XOR	halfword exclusive or register
244+(n-1)	Wn	XOR	word exclusive or register
176250	BY	SHL	byte shift logical
176251	H	SHL	halfword shift logical
176252	W	SHL	word shift logical
176253	BY	SHA	byte shift arithmetical
176254	H	SHA	halfword shift arithmetical
176255	W	SHA	word shift arithmetical
176256	BY	SHR	byte shift rotational
176257	H	SHR	halfword shift rotational
176260	W	SHR	word shift rotational
176264+(n-1)	BYn	GETBI	byte get bit
176270+(n-1)	Hn	GETBI	halfword get bit
176720+(n-1)	Wn	GETBI	word get bit
176724+(n-1)	BYn	PUTBI	byte put bit
176730+(n-1)	Hn	PUTBI	halfword put bit
176734+(n-1)	Wn	PUTBI	word put bit
177175	BY	CLEBI	byte clear bit
177176	H	CLEBI	halfword clear bit
177177	W	CLEBI	word clear bit
177200	BY	SETBI	byte set bit
177201	H	SETBI	halfword set bit

177202	W	SETBI	word set bit
176740+(n-1)	BYn	GETBF	byte get bit field
176744+(n-1)	Hn	GETBF	halfword get bit field
176750+(n-1)	Wn	GETBF	word get bit field
176754+(n-1)	BYn	PUTBF	byte put bit field
176760+(n-1)	Hn	PUTBF	halfword put bit field
176764+(n-1)	Wn	PUTBF	word put bit field
176300+(n-1)	Fn	AXI	register float argument to the <I>'th power
176304+(n-1)	Dn	AXI	register double float argument to the <I>'th power
176310+(n-1)	BYn	IXI	register byte I to the <J>'th power
176314+(n-1)	Hn	IXI	register halfword I to the <J>'th power
176320+(n-1)	Wn	IXI	register word I to the <J>'th power
176324+(n-1)	Fn	SQRT	register float square root
176330+(n-1)	Dn	SQRT	register double float square root
176275	BI	SWAP	bit swap
176276	BY	SWAP	byte swap
176277	H	SWAP	halfword swap
122	W	SWAP	word swap
176334	F	SWAP	float swap
176335	D	SWAP	double float swap
176340+(n-1)	Fn	POLY	floating polynomial
176344+(n-1)	Dn	POLY	double float polynomial
177130+(n-1)	Fn	REM	float divide with remainder
177134+(n-1)	Dn	REM	double float divide with remainder
177140+(n-1)	Fn	INT	float integer part
177144+(n-1)	Dn	INT	double float integer part
177150+(n-1)	Fn	INTR	float integer part with rounding
177154+(n-1)	Dn	INTR	double float integer part with rounding
176350+(n-1)	BYn	MULAD	byte multiply and add
176354+(n-1)	Hn	MULAD	halfword multiply and add
250+(n-1)	Wn	MULAD	word multiply and add
176360+(n-1)	Fn	MULAD	float multiply and add
176364+(n-1)	Dn	MULAD	double float multiply and add
176370+(n-1)	BYn	PSUM	byte add and multiply
176374+(n-1)	Hn	PSUM	halfword add and multiply
176400+(n-1)	Wn	PSUM	word add and multiply
176404+(n-1)	Fn	PSUM	float add and multiply
176410+(n-1)	Dn	PSUM	double float add and multiply
176414+(n-1)	BYn	LIND	byte load index
176420+(n-1)	Hn	LIND	halfword load index
254+(n-1)	Wn	LIND	word load index
176424+(n-1)	BYn	CIND	byte calculate index
176430+(n-1)	Hn	CIND	halfword calculate index

260+(n-1)	Wn CIND	word calculate index

CONTROL INSTRUCTIONS

Instruction Codes octal value	assembly notation	
300	GO:B	jump byte
301	GO:H	jump halfword
302	GO:W	jump word
264	JUMPG	jump general

Instruction Codes octal value	assembly notation	condition	name
	IF=GO	Z=1	equal
	IF Z GO		(alt. assembly notation)
304	IF=GO:B		byte displacement
305	IF=GO:H		halfword displacement
	IF><GO	Z=0	unequal
	IF -Z GO		(alt. assembly notation)
306	IF><GO:B		byte displacement
307	IF><GO:H		halfword displacement
	IF>GO	S=0 and Z=0	greater signed
310	IF>GO:B		
311	IF>GO:H		
	IF<GO	S=1	less signed
	IF S GO		(alt. assembly notation)
312	IF<GO:B		
313	IF<GO:H		
	IF>=GO	S=0	greater or equal signed
	IF -S GO		(alt. assembly notation)
314	IF>=GO:B		
315	IF>=GO:H		
	IF<=GO	S=1 or Z=1	less or equal signed
316	IF<=GO:B		
317	IF<=GO:H		
	IF K GO	K=1	flag
320	IF K GO:B		
321	IF K GO:H		
	IF -K GO	K=0	not flag
322	IF -K GO:B		
323	IF -K GO:H		
	IF>>GO	C=1 and Z=0	greater magnitude
324	IF>>GO:B		
325	IF>>GO:H		
	IF>>=GO	C=1	greater or equal magnitude

	IF C GO		(alt. assembly notation)
326	IF>>=GO:B		
327	IF>>=GO:H		
	IF<<GO	C=0	less magnitude
	IF -C GO		(alt. assembly notation)
330	IF<<GO:B		
331	IF<<GO:H		
	IF<<=GO	C=0 or Z=1	less or equal magnitude
332	IF<<=GO:B		
333	IF<<=GO:H		
	IF ST GO		specified bit in status register set
176173	IF ST GO:B		
176544	IF ST GO:H		
	IF -ST GO		specified bit in status register not set
176545	IF -ST GO:B		
176204	IF -ST GO:H		

Instruction Codes

octal value	assembly notation	name
176336	BY LOOPI:B	byte loop increment
176436	BY LOOPI:H	byte loop increment
176337	H LOOPI:B	halfword loop increment
176437	H LOOPI:H	halfword loop increment
277	W LOOPI:B	word loop increment
341	W LOOPI:H	word loop increment
176434	F LOOPI:B	float loop increment
176441	F LOOPI:H	float loop increment
176435	D LOOPI:B	double float loop increment
176442	D LOOPI:H	double float loop increment
176443	BY LOOPD:B	byte loop decrement
176450	BY LOOPD:H	byte loop decrement
176444	H LOOPD:B	halfword loop decrement
176451	H LOOPD:H	halfword loop decrement
176445	W LOOPD:B	word loop decrement
176452	W LOOPD:H	word loop decrement
176446	F LOOPD:B	float loop decrement
176453	F LOOPD:H	float loop decrement
176447	D LOOPD:B	double float decrement
176454	D LOOPD:H	double float decrement
176455	BY LOOP:B	byte loop general step
176462	BY LOOP:H	byte loop general step
176456	H LOOP:B	halfword loop general step
176463	H LOOP:H	halfword loop general step
176457	W LOOP:B	word loop general step
176464	W LOOP:H	word loop general step
17646C	F LOOP:B	float loop general step

176465	F LOOP:H	float loop general step
176461	D LOOP:B	double float loop general step
176466	D LOOP:H	double float loop general step
303	CALL	call subroutine absolute
265	CALLG	call subroutine general
334	INIT	initialize stack
337	ENTM	enter module
234	ENTD	enter subroutine directly
270	ENTS	enter stack subroutine
335	ENTF	enter subroutine
272	ENTSN	enter max argument stack subroutine
336	ENTFN	enter max argument subroutine
274	ENTT	enter trap handler
275	ENTB	enter buddy subroutine
200	RET	clear flag return from subroutine
201	RETK	set flag return from subroutine
202	RETD	return from direct subroutine
203	RETT	trap handler return
235	IF K RET	if flag set subroutine return
177034	RETB	buddy subroutine return
177035	RETBK	set flag buddy subroutine return

SPECIAL INSTRUCTIONS

Instruction Codes octal value	assembly notation	name
177000	SOLO	disable process switch
177001	TUTTI	enable process switch
176471	SETE	set bit in local trap enable register
176472	CLTE	clear bit in local trap enable register
176500	Wn STIFZ	compare and store if zero
176504	BI BYCONV	bit to byte convert
176505	BI HCONV	bit to halfword convert
176506	BI WCONV	bit to word convert
176507	BI FCONV	bit to float convert
176510	BI DCONV	bit to double float convert
176511	BY BICONV	byte to bit convert
176512	BY HCONV	byte to halfword convert
176513	BY WCONV	byte to word convert
176514	BY FCONV	byte to float convert
176515	BY DCONV	byte to double float convert
176516	H BICONV	halfword to bit convert
176517	H BYCONV	halfword to byte convert
176520	H WCONV	halfword to word convert
176521	H FCONV	halfword to float convert
176522	H DCONV	halfword to double float convert

176523	W	BICONV	word to bit convert
176524	W	BYCONV	word to byte convert
176525	W	HCONV	word to halfword convert
176526	W	FCONV	word to float convert
176527	W	DCONV	word to double float convert
176530	F	BICONV	float to bit convert
176531	F	BYCONV	float to byte convert
176532	F	HCONV	float to halfword convert
176533	F	WCONV	float to word convert
176534	F	DCONV	float to double float convert
176535	D	BICONV	double float to bit convert
176536	D	BYCONV	double float to byte convert
176537	D	HCONV	double float to halfword convert
176540	D	WCONV	double float to word convert
176541	D	FCONV	double float to float convert
177160	F	BYCONR	float to byte convert with rounding
177161	D	BYCONR	double float to byte convert with rounding
177162	F	HCONR	float to halfword convert with rounding
177163	D	HCONR	double float to halfword convert with rounding
177164	F	WCONR	float to word convert with rounding
177165	D	WCONR	double float to word convert with rounding
177203	W	FCONR	word to float convert with rounding
177204	D	FCONR	double float to float convert with rounding
177040+(n-1)	BIn	LADDR	bit load address
177044+(n-1)	BYn	LADDR	byte load address
177050+(n-1)	Hn	LADDR	halfword load address
176474+(n-1)	Wn	LADDR	word load address
176474+(n-1)	Fn	LADDR	float load address
177054+(n-1)	Dn	LADDR	double float load address
176125	BI	RLADDR	bit load address record
176132	BY	RLADDR	byte load address record
176261	H	RLADDR	halfword load address record
276	W	RLADDR	word load address record
276	F	RLADDR	float load address record
176262	D	RLADDR	double float load address record
176263	BI	BLADDR	bit load address local
176274	BY	BLADDR	byte load address local
176467	H	BLADDR	halfword load address local
176543	W	BLADDR	word load address local
176543	F	BLADDR	float load address local
176470	D	BLADDR	double float load address local
002	BP		break point instruction
003	NOOP		no operation

177002	SETK	set flag
177003	CLRK	clear flag
177114+(n-1)	Wn GETB	get buddy
176666	FREEB	free buddy
275	ENTB	enter buddy subroutine
177034	RETB	buddy subroutine return
177035	RETBK	buddy subroutine error return

REGISTER COMMUNICATION INSTRUCTIONS

Instruction Codes octal value	assembly notation	name
176473	L:=	load link register
176667	HL:=	load upper limit register
176670	LL:=	load lower limit register
176671	ST1:=	load first status register
176673	TE1:=	load first local trap enable register
176674	TE2:=	load second local trap enable register
176675	TOS:=	load top of stack register
176712	THA:=	load trap handler register
176700	L=:	store link register
176701	HL=:	store upper limit register
176702	LL=:	store lower limit register
176703	ST1=:	store first status register
176705	TE1=:	store first local trap enable register
176706	TE2=:	store second local trap enable register
176707	SE1=:	store first system trap enable register
176710	SE2=:	store second system trap enable register
176711	TOS=:	store top of stack register
176713	THA=:	store trap handler register
176542	P=:	store program counter
177060+(n-1)	An:=	load most significant part of double float register
177064+(n-1)	En:=	load least significant part of double float register
177070+(n-1)	An=:	store most significant part of double float register
177074+(n-1)	En=:	store least significant part of double float register
176440	BY BMOVE	byte block move
177170	H BMOVE	halfword block move
177171	W BMOVE	word block move
177172	F BMOVE	float block move
177173	D BMOVE	double float block move

STRING INSTRUCTIONS

Instruction Codes octal value	assembly notation	name
176546	BI SMOVE	bit string move
176547	BY SMOVE	byte string move
176550	H SMOVE	halfword string move
176551	W SMOVE	word string move
176552	F SMOVE	float string move
176553	D SMOVE	double float string move
176562	BY SMVWH	byte move string while
176563	BY SMVUN	byte move string until
176564	BY SMVTR	move translated string
176565	BY SMVTU	move string translated until
176566	BI SMOVN	string move n bits
176567	BY SMOVN	string move n bytes
176570	H SMOVN	string move n halfwords
176571	W SMOVN	string move n words
176572	F SMOVN	string move n floats
176573	D SMOVN	string move n double floats
176574+(n-1)	BIn SFILL	bit string fill
176600+(n-1)	Bn SFILL	byte string fill
176604+(n-1)	Hn SFILL	halfword string fill
176610+(n-1)	Wn SFILL	word string fill
176614+(n-1)	Fn SFILL	float string fill
176620+(n-1)	Dn SFILL	double float string fill
176624+(n-1)	BIn SFILLN	string fill n bits
176630+(n-1)	BYn SFILLN	string fill n bytes
176634+(n-1)	Hn SFILLN	string fill n halfwords
176640+(n-1)	Wn SFILLN	string fill n words
176644+(n-1)	Fn SFILLN	string fill n floats
176650+(n-1)	Dn SFILLN	string fill n double floats
176654	BY SCOMP	string compare
176655	BY SCOTR	string compare translated
176676	BY SCOPA	string compare with pad
176677	BY SCOPT	string compare translated with pad
176656	BY SSKIP	skip elements
176657	BI SLOCA	string locate bit
176660	BY SLOCA	string locate byte
176661	BY SSCAN	string scan
176662	BY SSPAN	string span
176663	BY SMATCH	string match
176664	BY SSPAR	set parity in string
176665	BY SCHPAR	check parity in string

APPENDIX I

INSTRUCTION CODE TABLE

	BI	BY	H	W	F	D
tn :=	176004	004	010	014	020	024
R :=				030		
B :=				176010		
tn =:	176014	034	176020	040	044	050
R =:				176011		
B =:				176012		
t MOVE	176013	031	176024	032	033	054
tn COMP	176030	060	176034	064	070	074
t COMP2	176025	055	176026	056	057	100
t TEST	101	102	103	104	105	106
tn NEG		177010	177014	220	224	224
tn INV	177020	177024	177030	230		
tn INVC				177420		
tn ABS		177400	177404	177410	177414	177414
tn +		176064	176070	124	130	134
tn -		176074	176100	140	144	150
tn *		176104	176110	154	160	164
tn /		176114	176120	170	174	350
t ADD2		176027	176124	123	176126	176127
t SUB2		176130	176131	340	176133	176134
t MUL2		176135	176136	176137	176140	176141
t DIV2		176142	176143	176144	176145	176146
t ADD3		176147	176150	176151	176152	176153
t SUB3		176154	176155	176156	176157	176160
t MUL3		176161	176162	176163	176164	176165
t DIV3		176166	176167	176170	176171	176172
tn MUL4		176040	176044	176050		
tn DIV4		176054	176060	176174		
tn UMUL				176200		
tn UDIV				177110		
tn ADDC				177100		
tn SUBC				177104		
tn CLR	204	204	204	204	210	214
t STZ	176205	110	111	112	113	114
t SET1	176206	176207	176210	115	107	176211
t INCR		176212	116	117	120	176213
t DECR		176214	176215	121	176216	176217
tn AND	176714	176220	176224	344		
tn OR	176770	176230	176234	240		
tn XOR	176774	176240	176244	244		
t SHL		176250	176251	176252		
t SHA		176253	176254	176255		
t SHR		176256	176257	176260		
tn GETBI		176264	176270	176720		
tn PUTBI		176724	176730	176734		
t CLEBI		177175	177176	177177		
t SETBI		177200	177201	177202		
tn GETBF		176740	176744	176750		
tn PUTBF		176754	176760	176764		
tn AXI					176300	176304
tn IXI		176310	176314	176320		
tn SQRT					176324	176330
t SWAP	176275	176276	176277	122	176334	176335
tn POLY					176340	176344
tn REM					177130	177134
tn INT					177140	177144
tn INTR					177150	177154
tn MULAD		176350	176354	250	176360	176364
tn PSUM		176370	176374	176400	176404	176410
tn LIND		176414	176420	254		
tn CIND		176424	176430	260		
:B GO				300		
:H GO				301		

	BI	BY	H	W	F	D
:W GO				302		
JUMPG				264		
:B IF = GO				304		
:H IF = GO				305		
:B IF >< GO				306		
:H IF >< GO				307		
:B IF > GO				310		
:H IF > GO				311		
:B IF < GO				312		
:H IF < GO				313		
:B IF >= GO				314		
:H IF >= GO				315		
:B IF <= GO				316		
:H IF <= GO				317		
:B IF K GO				320		
:H IF K GO				321		
:B IF -K GO				322		
:H IF -K GO				323		
:B IF >> GO				324		
:H IF >> GO				325		
:B IF >>= GO				326		
:H IF >>= GO				327		
:B IF << GO				330		
:H IF << GO				331		
:B IF <<= GO				332		
:H IF <<= GO				333		
:B IF ST GO				176173		
:H IF ST GO				176544		
:B IF -ST GO				176545		
:H IF -ST GO				176204		
:B t LOOPI		176336	176337	277	176434	176435
:H t LOOPI		176436	176437	341	176441	176442
:B t LOOPD		176443	176444	176445	176446	176447
:H t LOOPD		176450	176451	176452	176453	176454
:B t LOOP		176455	176456	176457	176460	176461
:H t LOOP		176462	176463	176464	176465	176466
CALL				303		
CALLG				265		
INIT				334		
ENTM				337		
ENTD				234		
ENTS				270		
ENTF				335		
ENTSN				272		
ENTFN				336		
ENTT				274		
ENTB				275		
RET				200		
RETK				201		
RETB				177034		
RETBK				177035		
RETD				202		
RETT				203		
IF K RET				235		
SOLO				177000		
TUTTI				177001		
SETE				176471		
CLTE				176472		
tn STIFZ				176500		
t BICONV		176511	176516	176523	176530	176535
t BYCONV	176504		176517	176524	176531	176536
t HCONV	176505	176512		176525	176532	176537
t WCONV	176506	176513	176520		176533	176540

		BI	BY	H	W	F	D
t	FCONV	176507	176514	176521	176526		176541
t	DCONV	176510	176515	176522	176527	176534	
t	BYCONR					177160	177161
t	HCONR					177162	177163
t	WCONR					177164	177165
t	FCONR				177203		177204
tn	LADDR	177040	177044	177050	176474	176474	177054
t	RLADDR	176125	176132	176261	276	276	176262
t	BLADDR	176263	176274	176467	176543	176543	176470
	BP				002		
	NOOP				003		
	illeg.1				000		
	illeg.2				001		
	SETK				177002		
	CLRK				177003		
Wn	GETB				177114		
	FREEB				176666		
L	:=				176473		
HL	:=				176667		
LL	:=				176670		
ST1:=					176671		
TE1:=					176673		
TE2:=					176674		
TOS:=					176675		
THA:=					176712		
L	=:				176700		
HL	=:				176701		
LL	=:				176702		
ST1=:					176703		
TE1=:					176705		
TE2=:					176706		
SE1=:					176707		
SE2=:					176710		
TOS=:					176711		
THA=:					176713		
P	=:				176542		
An	:=				177060		
En	:=				177064		
An	=:				177070		
En	=:				177074		
t	BMOVE		176440	177170	177171	177172	177173
t	SMOVE	176546	176547	176550	176551	176552	176553
t	SMVWH		176562				
t	SMVUN		176563				
t	SMVTR		176564				
t	SMVTU		176565				
t	SMOVN	176566	176567	176570	176571	176572	176573
tn	SFILL	176574	176600	176604	176610	176614	176620
tn	SFILLN	176624	176630	176634	176640	176644	176650
t	SCOMP		176654				
t	SCOTR		176655				
t	SCOPA		176676				
t	SCOPT		176677				
t	SSKIP		176656				
t	SLOCA	176657	176660				
t	SSCAN		176661				
t	SSPAN		176662				
t	SMATCH		176663				
t	SSPAR		176664				
t	SCHPAR		176665				
n	exten.				374		

INDEX

Page:

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

COMMENT AND EVALUATION SHEET

NORD-500 ASSEMBLER Reference Manual ND-60.113.02
MAY 1980

In order for this manual to develop to the point where it best suits your needs, we must have your comments, corrections, suggestions for additions, etc. Please write down your comments on this preaddressed form and mail it. Please be specific wherever possible.

FROM ....
....
....

– we make bits for the future

NORSK DATA A.S BOX 4 LINDEBERG GÅRD OSLO 10 NORWAY PHONE: 30 90 30 TELEX: 18661