NORD

COMPUTER SYSTEMS

# ASSEMBLER for NORD-5

April 1972

A/S NORSK DATA-ELEKTRONIKK

Økernveien 145, Oslo 5

# ASSEMBLER for NORD-5

April 1972

CONTENTS

+++
+

# 1 GENERAL INFORMATION

## 1.1 Introduction

The NORD-5 assembler is a two-pass assembler. On the first pass all macroes are expanded and labels are recognized and stored in the label table together with their values. Certain pseudo opcodes that may change the assembly address will also be serviced.

During the second pass all macroes are expanded, the actual assembly of instructions is performed, all pseudo opcodes that will effect the assembly address are checked, assembler commands are acted upon and all output is done.

## 1.2 Language Characteristics

### 1.2.1 Definition

The NORD-5 will execute a program stored in its core memory. Each memory location will contain information that will direct the operation of the central processing unit or data used or generated during the execution of the stored program.

To set up the computer to perform a particular task the programmer may figure out the particular bit pattern required and insert it into the memory. For a program of any appreciable size this becomes a tedious task prone to introducing errors.

To aid the programmer in setting up his NORD-5, the current assembly program has been made available. The assembler allows the programmer to use easily remembered acronyms for the different tasks that the computer may perform. Locations and registers that are used may be given symbolic names. When this symbolic program is processed by the assembler program, the appropriate numerical values will be obtained and substituted for the symbolic program, and a binary program is obtained. Writing his program on a symbolic form will ease the programmers work, and the resulting program is readily modified.

This assembler is implemented as a two-pass assembler. Thus the source program has to be processed twice. The assembler contains tables for labels, macro prototype definitions, opcodes and pseudo opcodes. During the first pass the following takes place: All labels are picked up and saved in the label table together with their values. Each record is checked to see if it contains a pseudo opcode. If certain pseudo opcodes like ORG and BSS are detected, the current assembler address is updated. If the EQU pseudo opcode is used to define a label, all labels in the argument must have been defined in a previous record.

Each pass is terminated by the END pseudo opcode. If labels are defined as global labels (GLO), external references (EXT) or as being common area references, the appropriate flag bits are set in the label table.

Each record is checked to see if it contains a macro call, as all macroes have to be expanded during pass one.

During the second pass the following takes place: Each record is checked to see if it contains a pseudo opcode. If not, opcodes are checked for. If no legal opcode is found, a check is made to see if the record contains a macro call. Due to this search sequence of the tables, a mnemonic appearing in the opcode or pseudo opcode tables must not be used as a macro name. If a macro call was detected, a switch is set to "a macro to be expanded" and call sequence parameters are saved. If a pseudo opcode is detected, similar action as in pass one is taken. If an opcode is detected, its numeric value is obtained from the opcode table, arguments are evaluated and the numeric instruction is assembled.

If requested, a listing and binary data are output to the assigned files.

## 1.2.2 Symbols used

An argument may contain a constant, a symbolic label or an arithmetic combination of any number of these. Several special characters are used to identify constants and arithmetic operations. Special characters are used to specify constants as follows.

Octal number. A number preceded by an apostroph (') will be treated as an octal number by the assembler.

Decimal number. Any number not preceded by a special character will be treated as a decimal number.

ASCII character. A character preceded by a # will be treated as its 7 bits ASCII value.

The format of labels has been described in Section 3.11. The values of all labels have been determined and saved in the label table during pass one. When the assembler is evaluating an argument, it will obtain the value of labels from the label table. Constants will be evaluated by the appropriate subroutines. The values thus obtained may be combined by using the following arithmetic operators (+), (-), (*) or (/). By using these operators integer arithmetic may be performed as follows.

Addition. A (+) sign will add what is on the left of the (+) sign to the first entry to the right of the (+) sign.

Subtraction. The entry to the right of the (-) sign will be subtracted from what is one the left of the (-) sign.

Multiplication. A (*) sign will multiply what is on the left of the (*) sign by the first entry to the right of the (*) sign.

Division. A (/) sign will divide what is on the left of the (/) sign by the first entry to the right of the (/) sign.

Unary (+) and (-) are allowed.

It should be noted that the address arithmetic works from left to right. This is illustrated in the following examples:

```
2 + 3 * 4 = 20
2 * 3 + 4 = 10
```

Now constants and labels may be used in an argument when the above rules for address arithmetic are observed. The following gives examples of how to use the address arithmetic.

```
LABEL + 5
'10 * LABT + AB
LABEL * 2/3 + 5
etc.
```

The fact that the integer arithmetic works from left to right may often be used to great advantage. If it should be desired to perform address arithmetic requiring parenthesis are as in F = (A * B) + (C * D) this may be done as follows:

```
E EQU  C * D
F EQU  A * B+E
```

Current location. The (*) sign will be interpreted as current location when it is the first entry in an argument and when immediately followed by (+), (-), (*) or (/).

Literals. A literal is specified by using the (=) sign. Each time a literal is specified in a memory reference instruction, a new location containing the constant is generated. This constant is specified as if using the GCN pseudo opcode (see Section 2.3). The address field of the memory reference instruction will refer to this new location.

To specify a literal, the (=) sign should immediately precede the literal. The literal may contain a constant, a symbolic label or a combination of these.

Examples,

To load 10 into register 3:

```
LDR  3, = 10
```

To load register 3 with the address of ENTRY:

```
LDR  3, = ENTRY
```

Note however: No relocating of ENTRY!

The locations containing the literal constants will appear after the first LOR pseudo opcode. If a program contains more than one LOR, the constants appearing after a LOR will only be those requested since the last LOR.

### 1.2.3 Types of Statements

When writing an assembly program, the programmer has the choice of three major types of statements,

Machine oriented statements

Process oriented statements

Data definition statements

A machine oriented statement will normally occupy one location in the object program. The contents of this location will direct the NORD-5 to perform one specific task when the assembly program is being executed. The task may be specified by any of the instructions (opcodes) listed in Appendix A. A machine oriented statement is specified by an opcode followed by no more than five arguments depending on the instruction.

A process oriented statement is used to give the assembler information concerning the assembly. Pseudo opcodes may give the start of a program (ORG), end of program (END), room for data storage (BSS) etc. It is seen that pseudo opcodes do not generate any data that become part of object program. But a process oriented statement may determine the load or assembly location of a machine oriented statement and its actual assembled value. A process oriented statement is specified by a pseudo opcode followed by one or more arguments. A macro call directs the assembler to fetch one or more statements to be inserted after the macro call.

A data definition statement is used to introduce data into the assembly program. Examples of data are decimal constants, floating point constants and alphanumeric data. The data defined may require one or more locations of core storage. Data is introduced by a pseudo opcode followed by one operand giving the data to be introduced.

The above statements are described in detail in Section 2.

## 1.3 Language Environment

The assembler is written in the NORD-1 assembly language. Thus it must be executed on a NORD-1. The assembler is a part of the NORD-OPS operating system. Thus it must initially be called through the operating system. Once an assembly is started, all input and output is through assigned files.

# 2 LANGUAGE STATEMENTS

## 2.1 Machine oriented Statements

The NORD-5 will accept the two following major types of executable instructions,

Memory Reference Statements

Register Instructions.

### 2.1.1 Symbolic Formats and Object Translations

All machine instructions are written on symbolic form by the programmer and translated to the machine instruction format by the assembler.

Generally the programmer will specify:

1) An operation to be performed,

2) one or more registers to be operated upon, and

3) further specification of operation.

The operation in 1) is given as the operation code (opcode). Examples are add and shift operations. A summary of all opcodes may be found in Appendix A. Operations in 2) and 3) are given as operands. There may be from one to five operands depending on the operation to be performed. Operands are separated by a comma (,). The opcode is separated from operands by one or more blanks as in the following example:

```
OPC  OP1, OP2, OP3
```

### 2.1.2 Memory Reference Instructions

A memory reference instruction is specified by the following general statement:

```
OPC  R,D,B,X,I
```

The opcode is given as OPC and may be any of the memory reference opcodes given in Appendix A.

The register to be operated upon is given as R, and may be any of the 64 registers available.

The memory location it is desired to reference is given as D.

The remaining three parameters are not necessarily required. Thus a memory reference instruction may contain only OPC, R and D. If one of the remaining parameters are required, any preceding parameter has to be specified. Thus if it is desired to specify X register, a B register must also be specified. However, if (,,) is used, the assumed base register is inserted for B and index register 0 for X.

If a base register is required it is specified by B. As base register may be used any of the 15 base registers available. Each time a memory reference instruction is specified, an assumed base register is inserted into the machine instruction being assembled, unless a base register has been specified by the programmer. The assumed base register is set to zero at the start of each assembly pass and may be changed by the BAS pseudo opcode.

If it is desired to use an index register for address modification, any of the 15 index registers may be specified in the X position.

If it is desired to specify an indirect operation, I should be specified as a non-zero value.

The values substituted for R, D, B, X and I may be any decimal or octal constants, label or a valid arithmetic combination of constants and labels. Literals may be used in the D field.

## 2.1.3 Register Operations

A register operation is specified by the following general statement:

```
          OPC  DR,SR,B
```

The opcode is given as OPC and may be any of the register operations given in Appendix A.

The register to be operated upon is given as DR and may be any of the 64 available registers.

The source register is given as SR, and may be any of the 64 available registers. A source register is not required for the SZR and SON opcodes.

Parameter B will contain information depending on the opcode according to the following table

| Operation | B field contents |
|---|---|
| Register I/O | External register contents |
| Shift | Shift count |
| Bit | Bit number |
| Logical register | Second source register |
| Register | Second source register |
| Skip | Second source register |

### 2.1.4 Argument Instructions

An argument instruction is specified by the following general statement:

```
OPC R,A
```

The opcode is given as OPC may be any of the argument instructions specified in Appendix A.

The register to be operated upon is given as R, and may be any of the 64 available registers.

The argument is given as A. The size of the argument is limited to 16 bits. The argument may be a constant, label or any valid arithmetic combination of these.

## 2.2 Process oriented Statements

A process oriented statement will give a specific directive instruction to the assembler. Thus the information conveyed will be acted upon by the assembler at assembly time and used to control the assembly process. Process oriented statements may be used to specify that a binary load tape is desired, the next statement should be listed on the top of the next page, the end of the assembly has been reached, etc.

### 2.2.1 Symbolic Format

A process oriented statement will be of the form:

```
POC A,B,C
```

where POC is a pseudo opcode specifying the directive instruction. The pseudo opcode will normally contain three alphabetic characters. The pseudo opcode is followed by one or more arguments. Each argument will normally be separated by a comma. An argument may be any valid arithmetic combination.

### 2.2.2 Available Directive Instructions

#### 2.2.2.1 Assumed Base Register

One or more assumed base registers may be specified as,

```
BAS  LABEL,B
```

where LABEL is a label appearing in the source program and B specifies a base register. B may be a numeric value, symbolic reference or any valid arithmetic combination of numeric values and references which will specify any of the 15 available base registers.

A source program may contain several BAS pseudo opcodes associating base registers to several entry labels.

When a memory reference instruction or address constant (ACN) is being assembled, the evaluated address will be compared to the value given to labels referenced by BAS pseudo opcodes, and the one giving the smallest displacement from the address referenced is selected. Next the base register associated with this label is inserted into the instruction or constant being assembled.

A maximum of 8 BAS pseudo opcodes may be specified in a program. If more than 8 BAS pseudo opcodes are specified, the first assumed base register specified will be replaced by the new one, etc. Thus the list for storing assumed base registers are of a circular nature.

### 2.2.2.2 Reserve Data Block

A part of memory may be reserved as

```
BSS  A
```

where the parameter A gives the number of words to be reserved. A may be any valid arithmetic expression giving a positive number when evaluated by the assembler. A negative BSS is not valid and will not reserve any room. The value of a BSS will be listed in column 2 of the assembly listing. If a label is specified at the same time as the BSS the label will be giving the value of the location of the first storage word reserved by the BSS.

### 2.2.2.3 Clear

The pseudo opcode CLR will clear local labels, global labels and macro prototype tables. This pseudo opcode should be inserted as the first instruction in an assembly that does not require any information left over from a previous assembly.

### 2.2.2.4 Set Common Pointer

The pseudo COM sets a pointer to the program counter for the common area. Thus, each time the assembler modifies its program counter (assembly address) the program counter for the common area will be updated. All labels defined after a COM pseudo opcode will be flagged as being common labels in the label table. This is reset by the PRG pseudo opcode.

### 2.2.2.5 Conditional Assembly

Conditional assembly may be specified by using the following pair of pseudo opcodes

```
SCA  A,B
ECA
```

The SCA pseudo opcode gives the start of the conditional assembly, and ECA the end of the conditional assembly. If the two parameters A and B are not equal, the source statements appearing between the SCA and ECA statements will be assembled. If A and B are equal, the source statements between SCA and ECA will be listed as comments in the object listing. The comparison between the two parameters is arithmetic. The parameters A and B may be any valid arithmetic expression. Conditional assemblies may be nested as

```
SCA  A,B
.
.          a
SCA  C,D
.
.          b
ECA
.
.          c
ECA
```

Depending on the parameters A,B,C and D sections a,c or b or a,b,c may be assembled. Nesting rules are similar to FORTRAN DO statement nesting rules.

### 2.2.2.6 Program End

The end of a program is given by the pseudo opcode END. The END pseudo opcode will terminate assembly pass 1 and 2. When END is read at the end of pass 2, all local labels will be erased. Global labels will survive.

### 2.2.2.7 Equivalence

A label may be given a specific value as in

```
A  EQU  B
```

B may be any valid arithmetic expression. The assembler will evaluate B and assign this value to A. The value assigned to A will be listed in column 2 of the assembly listing.

#### 2.2.2.8 External Reference

The loader may be given information about external references by using the EXT pseudo opcode as

```
          EXT A,B,C
```

A,B,C are external labels that the current program wants to reference. Each time a reference is made to the label A in the program being assembled, information about this is made a part of the binary output. This information is thus made available to the loader which will update the locations in question as soon as information about the label is made available to the loader.

#### 2.2.2.9 Specify formatted Data Fields

The FORM pseudo opcode is used to specify data fields for formatted data. This pseudo opcode is described under FDAT in Section 2.3.

#### 2.2.2.10 Generate

If it is desired to repeat or generate a source statement several times, this may be done

```
          GEN A
```

Then the next source statement will be repeated A times. A may be any valid arithmetic statement giving a positive value when evaluated by the assembler.

If A is zero or negative, the next source statement will appear once. Any opcode, pseudo opcode or macro may be generated with the exception of a GEN pseudo opcode, a floating point constant or a string constant. However, floating point and string constants may appear inside a macro that is GENed. If a label appears on the same line as the GEN pseudo opcode, it will be assigned the value of the location given to the first of the GENed statements.

#### 2.2.2.11 Global Labels

Labels may be declared to be global as

```
          GLO A,B,C
```

A,B and C are labels defined in the program. As many labels as can be accomodated in a 80 column card image may be included following the GLO pseudo opcode.

### 2.2.2.12 Literal Orgin

If any literals have been used in the program, one or more locations have to be generated. If a LOR pseudo opcode is inserted into the program, all literals up to that point will be inserted immediately following the LOR pseudo opcode.

### 2.2.2.13 Program Name

The name of a program may be saved as part of the object load module by using the following pseudo opcode,

```
MAIN  A
```

A is a label defined in the program. This label and the value assigned to it will be saved in the load module.

### 2.2.2.14 Assembly Options

Assembly options are specified as,

```
OPT  A,B,C,D,E,F,G
```

where

| | |
|---|---|
| A = 1 | selects no listing |
| B = 1 | selects listing of errors only |
| C = 1 | selects binary output |
| D | selects FDN for source program |
| E | selects FDN for listing of assembly |
| F | selects FDN for binary output |
| G | selects FDN for intermediate storage |

Parameters A,B and C must be 0 or 1 or a symbolic expression giving that value when evaluated. Trailing parameters may be omitted. Thus if it is desired to select binary output, only parameters A,B and C have to be specified. If a file device should not be changed, its parameter may be set equal to zero.

```
OPT      0,0,0,27
OPT      0,0,1,0,47
```

After the two above pseudo opcodes have been assembled, the source program is read from file No. 27 and the assembly listing will be saved on file No. 47. File device numbers should not be changed during one assembly. Options should be selected as early as possible in the assembly.

### 2.2.2.15 Program Start

The start address of a program is given as,

```
ORG  A
```

where the parameter A gives the start location of the program. A may be any valid arithmetic expression. If one of the parameters in A is undefined, it will be assumed to be zero for the purpose of computing the starting address. If the ORG pseudo opcode has been omitted, the start address is assumed to be zero.

### 2.2.2.16 Set Program Pointer

The pseudo opcode PRG will set a pointer to the program counter for the program being assembled. Thus, each time the assembler modifies its program counter (assembly address) the program counter for the program being assembled will be updated. Also see the COM pseudo opcode.

### 2.2.2.17 Program Entry Point

The loader may be given information about entry points by using the REF pseudo opcode as,

```
REF  A,B,C
```

A,B,C are labels defined in the program. As many labels as can be in a 80 column card may be included following REF pseudo opcode. Each label and the value assigned to it will be saved as part of the object load module. This information will be picked up and stored by the loader which will use the information to link programs.

### 2.2.2.18 Print Cross Reference Table

If the XRE pseudo opcode is made part of a program, a cross reference table will be printed out at the end of the assembly. All labels, their assigned value and all locations where the label is referenced will be printed out. The labels will appear in alphabetical order. Symbols defined inside macroes will not be listed. Only references made subsequent to the XRE pseudo opcodes will appear in the listing.

## 2.3 Definition of Data

When it is desirable to insert a constant into a given location, this is achieved by using a pseudo opcode. This pseudo opcode will direct the assembler to interpret its argument as a constant to be converted and included as part of the object program. The pseudo opcode itself specifies the type of constant for the assembler. The following data definition statements are available.

### 2.3.1 General Constant

A general constant is specified by the following statement,

```
GCN  A
```

The assembler will evaluate the operand (A) as a single precision value. The operand may be any combination of numeric values, labels and arithmetic operators as described in Section 1.2.2.

### 2.3.2 Floating Point Constant

A floating point constant is specified by the following statement,

```
FCN  A
```

The assembler will evaluate the operand (A) as a floating point constant. The operand should be specified as in the FORTRAN E or F format statement. The mantissa and exponent may contain any number of characters consistent with the accuracy of the NORD-5 floating point format.

### 2.3.3 String Constant

A string constant is specified by the following statement,

```
SCN  'STRING'
```

The string constant is found between the two apostrophs ('). The string may contain any character except apostroph. The characters in the string will be packed four to a word with the first character in the most significant position in the data word. If only part of the last word is required for storing characters, the unused part will be filled with zeroes. Only the characters between the apostrophs will be stored, not the apostrophs. The characters are stored without parity. The maximum number of characters is only limited by the 80 character source record length.

## 2.3.4 Address Constant

An address constant is specified by the following statements,

```
ACN  LABEL,B,X,I
```

The assembler will evaluate the operand LABEL as a single precision value. The loader will add the program base to the value to get an absolute address.

B,X and I specify base, index and indirect modification of the address constant.

Thus the address will be relocated at load time, but otherwise similar to a memory reference instruction with the destination register omitted.

## 2.3.5 Formatted Data

It is possible to insert data into selected parts of a word by using the FORM and FDAT pseudo opcodes. The FORM pseudo opcode will divide a word into as many as 64 subfields. The FDAT pseudo opcode will be used to insert data according to the specification given by the last FORM pseudo opcode. The FORM pseudo opcode may be used as in

```
FORM  A,B,C
```

where only three fields are specified. Their lengths are A, B and C respectively. We may select actual numbers for the field lengths

```
FORM  10, 10, 11, 7
```

where the word is divided into four fields.

The following FDAT will specify data according to the format given by the last FORM,

```
FDAT  R+10, LABEL*3, 7, '10
```

When the assembler is evaluating the data given by a FDAT pseudo opcode, it will go through the following steps.

The data that is to go into each field is evaluated separately as a 32 bit constant.

The absolute value of the constant is checked to see if it will fit in its field. This may result in an error condition (operand flag).

## 2.4 Macro Extensions

In its simplest form a macro is an abbreviation for a sequence of instructions.

Often a sequence of instructions is to be repeated several times. It is then desirable to form abbreviations, for example we would like to "attach" a name to the sequence of instructions and use the name wherever we want the instruction sequence to occur. We attach the name to the sequence by means of a macro prototype definition.

### 2.4.1 Defining a Macro

A macro is defined as a macro prototype. This macro may then later be inserted into the program sequence one or more times by using a macro call. The macro prototype may contain any form of coding. It may contain executable instructions, assembler directive statements, macro calls and data definitions. This is subject to a few exceptions that will be listed below. It is noted that a prototype should not contain another prototype definition.

The prototype is stored in a separate table during the assembly. Thus the programmer should attempt to write the prototype as compact as possible in order to conserve storage space. Thus labels should be kept short and comments avoided.

The start of a macro prototype definition is specified by the MACR pseudo opcode, and the end of the definition by the EMAC pseudo opcode. There should be a label associated with the MACR pseudo opcode. This label specifies the name of the prototype. The macro name is given as one to five alphanumeric characters. A macro name should not be the same as one of the opcodes or pseudo opcodes found in Appendix A og B.

The MACR pseudo opcode may have one or more parameters. These parameters specify which labels the prototype should fetch from the call sequence. There are no label or argument associated with the EMAC pseudo opcode. Three types of labels may be referenced inside a macro prototype;

1) Labels defined external to the prototype except internal labels of another prototype.

2) Labels internal to the macro prototype.

3) Labels given as a parameter in the macro call sequence. If a label is referenced in the prototype and the same label appears as a MACR parameter, this label will be treated as a call sequence parameter. When the macro is called, the parameter in the corresponding location in the call sequence will be substituted for the label.
This is illustrated in the following example,

```
ARNA    MACR BAKER
        LDR  5,ABLE
        MPY  5,$BAKER
        STR  5,CHARLY
        RTJ  0,0,3
CHARLY  GCN  0
        EMAC
```

This macro prototype defines a macro called ARNA. The external label ABLE is referenced. When the macro is called, one parameter will be expected in the call sequence. This parameter will be substituted for BAKER. The internal label CHARLY is defined. Although the macro may be called several times, the internal label will not become multiply defined.

Regular labels may also be defined in a macro prototype. This would however, defy the purpose of the macro as the macro may be called only once. But it would be appropriate to define an entire program as a macro prototype. This prototype and a single call to it would then be read during pass one of an assembly. During the second pass only the macro call should be read. This way the source would be read only once. The macro prototype must appear in the source before it is being called the first time. The prototype is saved during pass one. If the prototype is read during pass two, it will be treated as a comment.

When defining a macro prototype the programmer should be aware of the following,

1) A macro may contain a call to iself or a call to a second macro that will call the first macro. This recursivity is limited to a level of 10.

2) A macro prototype should not be placed within another macro prototype.

3) A macro is global.

4) A prototype should not contain the GEN pseudo opcode if the macro is going to be GENed.

5) A maximum of 100 internal labels may be defined in any prototype.

6) The maximum number of prototypes that may be defined is 100. This is an assembly parameter that may be changed by reassembly.

7) A macro name should not be an opcode or pseudo opcode.

8) All prototypes should be defined before any label is defined.

## 2.4.2 Calling a Macro

A previously defined macro prototype may be called by using a macro call. This will cause the macro to be inserted after the macro call. The macro specified in 2.4.1 may be called as,

```
ARNA  DOG
```

Here the macro is called by placing the macro name (ARNA) in the opcode field. This particular macro requires one parameter in the call sequence (DOG). The above macro call will produce the following coding to be inserted immediately after the macro call,

```
LDR  5,ABLE
MPY  5,DOG
STR  5,CHARLY
RTJ   0,0,3
CHARLY GCN  0
```

It may be noted that the parameter DOG has been inserted into the MPY instruction.

If the macro call contains too many parameters the extra parameters will be ignored. If the macro call contains too few parameters, blanks will be substituted for the parameter.

No program should make more than 1156 macro calls.

# 3 USING THE LANGUAGE

## 3.1 How to write a Program

This section will contain information required by the programmer when he is going to write his program.

### 3.1.1 Source Program Format

The assembler is record oriented. Thus one record will be read into a buffer at a time for processing. The source will be read from a disc file or any other input device supported by the I/O system being used.

The source program may consist of machine oriented statements, directive statements to the assembler etc. One such statement will be contained in each record.

A record contains as many as 80 characters. The record is divided into four different fields,

1) The label field
2) The opcode field
3) The operand field
4) The comments field

A semi-free record is utilized. The record format is the same as the record format for the NORD-1 assembly language.

The label field starts in column one.

The opcode field is to the right of the label field (at least one space ahead of it).

The operand field is to the right of the opcode field (at least one space ahead of it).

The comments field is to the right of the operand field (at least two spaces ahead of it).

#### 3.1.1.1 The Label Field

The label, if any, will have from one to six alphanumeric characters. The first character must be alphabetic and appears in column 1. The first space or non-alphanumeric character found after column 1 indicated the end of the label. The period character (.) is treated like a digit.

#### 3.1.1.2 The Opcode Field

In this field may appear any of the opcodes or pseudo opcodes found in Appendix A and B and macro names.

## 3.1.1.3 The Operand Field

Arguments in the operand are left justified within its field. No space are allowed between arguments. The first space found indicates the end of the operand.

## 3.1.1.4 The Comments Field

When an (*) is found in column one the whole record is treated as a comment. If a comment is to appear on the same line as a statement to be assembled, it may be placed after the last operand. Then there should be at least one space separating the comment and the operand. It is suggested that comments start in column thirty-one. A blank record is ignored.

Examples showing the format used are shown in Appendix C.

The result of the assembly is listed in three major octal fields where the third field is broken down into several subfields. Field 1 contains the address against which the source statement is assembled.

Field 2 contains the result of the assembly. Only information that will actually be loaded into core during execution will appear in this field. All information in this field will appear in a binary load module. The field will never contain assembler or loader information.

The complete instruction in field 2 has been broken down and appears in the remaining subfields. This will make it easier for the programmer to determine which registers have been used, what locations have been referenced etc. Three different formats may be found depending on whether the assembled instruction is a memory reference, register or argument instruction. The contents of the different columns are summarized in the following table.

| Field | | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| Memory | Contents | I | X | B | OP | R | D | |
| Reference | Bit No. | 31 | 30-27 | 26-23 | 22-18 | 17-12 | 11-0 | |
| Register | Contents | 0 | R | mix | 0 | DR | SRA | SRB |
| | Bit No. | 31 | 30-27 | 26-23 | 22-18 | 17-12 | 11-6 | 6-0 |
| Argument | Contents | I | | R | 0 | | ARG | |
| | Bit No. | 31 | 30-29 | 28-23 | 22-18 | 17-16 | 15-0 | |

When a memory reference instruction has been assembled, the letters X or C may appear between fields 7 and 8. This indicates that an external label (X) or a label defined to be in the common area (C) has been referenced in the instruction. If both an external and a common label have been referenced, the letter D will appear.

If the source statement is a pseudo opcode like ORG, BSS, EQU or GEN the value of the argument will appear in column 3.

If the source statement is an address constant I, X, B and the displacement will appear in sub columns 3 through 6.

## 3.2 How to prepare for Assembly

When the programmer is ready to assemble his program, the source 'deck' should contain the following,

1) ORG pseudo opcode giving the start of the program.
2) CLR pseudo opcode to clear tables if this assembly does not require information from any previous assembly.
3) The source program.
4) END pseudo opcode giving the end of the assembly.

The source program should appear in the sequence indicated above.

Control command:

```
$N5ASM ('options')
```

where 'options' has the same format as the operand field for the pseudo opcode OPT.

The program may be punched on paper tape or cards or be stored on a file on a mass storage device or any other peripheral device supported by the I/O system to be used with the assembler.

## 3.3 Assembly Output

### 3.3.1 The Assembly Listing

When the appropriate options are selected, the assembler will give an assembly listing. This listing contains the result of the assembly, information on assembly errors and a listing of the source.

An example of an assembly is given in Appendix C. Columns 1 through 40 contain several fields of octal information giving the result of the assembly. Starting in column 45 the source program is listed. Error flags will appear between the assembly result and the source listing.

If any assembly errors occurred during the assembly, error flags will appear right justified in columns 41 through 43. If a system error occurred, the appropriate message will be listed starting in column 1. The different error codes are explained in Section 3.3.2.

Starting in column 45 the source program is listed. The following assembler commands will not appear in the listing, HLT, NOLS and LIST.

## 3.3.2 Diagnostic Messages

When the assembler detects an error, a message to that effect will appear in the assembly listing. Errors may be introduced due to programmer errors or due to limitations imposed by the assembler.

### 3.3.2.1 Programmer Errors

When the programmer has made an error, one or more error flags will appear as described in 3.3.1. The sample in Appendix C should also be consulted as it shows the error flags as used for the different instructions. The different error flags are,

| | |
|---|---|
| O | Operand error |
| B | Illegal base register |
| R | Illegal destination register |
| A | Illegal opcode |
| X | Illegal index register |
| M | Label multiple defined |
| U | Label undefined |
| Q | Possible error |

When one of these errors except M and Q has been detected, a halt (STOP) instruction is substituted as the result of the assembly.

### 3.3.2.2 System Errors

When one of the limitations of the assembler has been exceeded, a system error will result. Then a message will appear in the assembly listing.

System errors are as follows:

1) Label table full.
2) Macro prototype table full.
3) Too many macroes expanded.
4) Cross reference table full.
5) Too many recursive macroes called.
6) Too many macro prototypes stored.

System errors are not recoverable and the assembly will be terminated.

## APPENDIX A

### SUMMARY OF INSTRUCTIONS

### A.1 Memory Reference Instructions

| Mnemonic | Action |
|---|---|
| RTJ | Return jump |
| EXC | Remote execute |
| MIN | Memory increment |
| CRG | Skip if $(R) \geq (Ea)$ |
| CRL | Skip if $(R) < (Ea)$ |
| CRE | Skip if $(R) = (Ea)$ |
| CRD | Skip if $(R) \neq (Ea)$ |
| JRP | Jump if $(R) \geq 0$ |
| JRN | Jump if $(R) < 0$ |
| JRZ | Jump if $(R) = 0$ |
| JRF | Jump if $(R) \neq 0$ |
| JPM | Modify (R) and jump if $(R) \geq 0$ |
| JNM | Modify (R) and jump if $(R) < 0$ |
| JZM | Modify (R) and jump if $(R) = 0$ |
| JFM | Modify (R) and jump if $(R) \neq 0$ |
| ADD | Add (Ea) to (R) |
| SUB | Subtract (Ea) from (R) |
| AND | Logical AND between (Ea) and (R) |
| LDR | Load (R) with (Ea) |
| ADM | Add (R) to (Ea) |
| XMR | Exchange (Ea) and (R) |
| STR | Store (R) in (Ea) |
| MPY | Multiply (R) by (Ea) |
| DIV | Divide (R) by (Ea) |
| LDF | Load (F) with (Ea, Ea + 1) |
| STF | Store (F) in (Ea, Ea + 1) |
| FAD | Add (Ea, Ea + 1) to (F) |
| FSB | Subtract (Ea, Ea + 1) from (F) |
| FMU | Multiply (F) by (Ea, Ea + 1) |
| FDV | Divide (F) by (Ea, Ea + 1) |

## A.2 Inter Register Operations

### A.2.1 Shift Instructions

| Mnemonic | Action |
|---|---|
| SLR | Left rotational shift |
| SRR | Right rotational shift |
| SLA | Left arithmetical shift |
| SRA | Right arithmetical shift |
| SLL | Left logical shift |
| SRL | Right logical shift |
| SLRD | Left rotational floating register shift |
| SRRD | Right rotational floating register shift |
| SLAD | Left arithmetical floating register shift |
| SRAD | Right arithmetical floating register shift |
| SLLD | Left logical floating register shift |
| SRLD | Right logical floating register shift |

### A.2.2 Miscellaneous Operations

| | |
|---|---|
| BST | Bit set |
| BCL | Bit clear |
| BSZ | Bit skip on zero |
| BSO | Bit skip on one |
| FIX | Convert floating to integer |
| FLO | Convert integer to floating |

### A.2.3 Arithmetic Operations

| | |
|---|---|
| RAD | Register add |
| RSB | Register subtract |
| RMU | Register multiply |
| RDV | Register divide |
| RAF | Floating register add |
| RSF | Floating register subtract |
| RMF | Floating register multiply |
| RDF | Floating register divide |

## A.2.4 Test and Skip

| Mnemonic | Action |
|---|---|
| SGR | Subtract registers and skip if result $\geqslant 0$ |
| ASG | Add " " " " " $\geqslant 0$ |
| SLE | Subtract " " " " " $< 0$ |
| ASL | Add " " " " " $< 0$ |
| SEQ | Subtract " " " " " $= 0$ |
| ASE | Add " " " " " $= 0$ |
| SUE | Subtract " " " " " $\neq 0$ |
| ASU | Add " " " " " $\neq 0$ |
| SGF | Subtract floating registers and skip if result $\geqslant 0$ |
| ASGF | Add " " " " " $\geqslant 0$ |
| SLF | Subtract " " " " " $< 0$ |
| ASLF | Add " " " " " $< 0$ |
| SEF | Subtract " " " " " $= 0$ |
| ASEF | Add " " " " " $= 0$ |
| SUF | Subtract " " " " " $\neq 0$ |
| ASUF | Add " " " " " $\neq 0$ |

## A.2.5 Logical Operations

| Mnemonic | Action |
|---|---|
| RND | Register AND |
| RNDA | Register AND, use complement of (SRA) |
| RNDB | Register AND, use complement of (SRB) |
| RXO | Register exclusive OR |
| RXOA | Register exclusive OR, use complement of (SRA) |
| RXOB | Register exclusive OR, use complement of (SRB) |
| ROR | Register OR |
| RORA | Register OR, use complement of (SRA) |
| RORB | Register OR, use complement of (SRB) |
| SZR | Set all zeroes |

## A.2.6 Argument Instructions

| | |
|---|---|
| XORA | Exclusive OR |
| ANDA | AND |
| ORA | OR |
| SETA | Register set |
| SECA | Set register to complement |
| ADDA | Add |
| ADCA | Add complement |
| DDP | Skip if (R) $\geq$ A |
| DDN | " " " $<$ A |
| DDZ | " " " $=$ A |
| DDF | " " " $\neq$ A |
| DSP | " " " $\geq$ -A |
| DSN | " " " $<$ -A |
| DSZ | " " " $=$ -A |
| DSF | " " " $\neq$ -A |

# APPENDIX B

## SUMMARY OF PSEUDO OPCODES

| | |
|---|---|
| BAS LABEL,B | The parameter B specifies a base register associated with LABEL to be used in memory reference instructions if a base register has not been specified. |
| BSS A | The parameter specifies the number of locations that is to be reserved. |
| CLR | Clear label tables. |
| COM | Start assembling into common area. |
| ECA | End of conditional assembly. Regular assembly is resumed after a previous SCA. |
| END | Program end. Will terminate pass one and two and erase local labels after end of pass two. |
| EMAC | End of macro prototype definition. |
| EQU A | The label is given the value specified by the argument. |
| EXT A,B,C.... | The parameters give the name of labels that are external to the current program. |
| FORM A,B,C.... | The parameters specify fields for later use by FDAT. |
| GEN A | The contents of the next source statement are repeated the number of times given by the parameter. |
| GLO A,B,C.... | The parameters give the name of labels that are to be declared as global labels. |

HLT

The assembly is temporarily stopped.

LIST

If listing of assembly is specified, listing will be resumed (see NOLS).

LOR

All literals requested after the last LOR will be defined following LOR.

MAIN A

The parameter gives the name of the program being assembled.

MACR A,B,C....

Start macro prototype definition. The label gives the name of the macro. The parameters give call sequence parameters.

NOLS

The assembly will not be listed (see LIST).

OPT A,B,C,D,E.F,G

The three first parameters give the desired assembly options (no listing, list error only, binary output if = 1). The four last parameters give the FDN of the files used.

ORG A

The selected program counter is set to the value given by the parameter.

PRG

Start assembling into the program areas.

REF A,B,C....

The parameters give the names of program labels that are required as external reference points.

SCA A,B

Start conditional assembly. If the two parameters are equal, the following source statements will not be assembled (see ECA).

XRE

Save data for a cross reference table to be printed at the end of assembly.

The following pseudo opcodes are used to specify data:

| | | |
|---|---|---|
| FDAT | A,B,C.... | Formatted data (see FORM) |
| GCN | A | General constant |
| FCN | E or F | Floating point constant |
| SCN | 'STRING' | String constant |
| ACN | LABEL,B,X,I | Address constant. |

## APPENDIX C

### SAMPLE LISTING

```
                                                   OPT    0,0,1,4,1,3
                                                   CLR
                                            *  SAMPLE LISTING
                                                   XRE
                            0000000620             ORG    400
00620 00027000642 0 00 00 27 00   0642             STR    0,OLE
00621 00027000643 0 00 00 27 00   0643             STR    0,HANS
00622 24100000000 1 05 00 00 02 000000             SETA   2,0
00623 00001000637 0 00 00 01 00   0637             RTJ    0,NILS
MONS                                               REF    MONS
00624 00023010643 0 00 00 23 01   0643      MONS   LDR    1,HANS
00625 00023020642 0 00 00 23 02   0642             LDR    2,OLE
00626 14000010102 0 14 00 00 01  01 02             RAD    1,1,2
00627 00020010642 0 00 00 20 01   0642             ADD    1,OLE
00630 24040600001 1 05 00 00 01 000001             ADCA   1,1
00631 00023030644 0 00 00 23 03   0644             LDR    3,TALL
00632 16040000103 0 16 01 00 00  01 03             SGR    0,1,3
00633 00001000635 0 00 00 01 00   0635             RTJ    0,*+2
00634 00000000000 0 00 00 00 00 000000             STOP   0
00635 00027010643 0 00 00 27 01   0643             STR    1,HANS
00636 07042010001 0 07 01 02 01   0001             EXC    1,1,1,7
00637 24100400001 1 05 00 00 02 000001      NILS   ADDA   2,1
00640 00027020642 0 00 00 27 02   0642             STR    2,OLE
00641 00001040624 0 00 00 01 04   0624             RTJ    4,MONS
00642 00000000000                           OLE    GCN    0
00643 00000000000                           HANS   GCN    0
00644                       0000000001      TALL   BSS    1
                                                   EXT    TRULS
                            0000000002             GEN    2
00645 00000000000                                  GCN    TRULS


00646 00000000000                                  GCN    TRULS
                                                   END
HANS     000643 000621 000624 000635
MONS      000624 000624 000641
NILS      000637 000623
OLE       000642 000620 000625 000627 000640
TALL      000644 000631
TRULS     000000 000645 000646
```

# APPENDIX D

## BRF IN NORD-5 ASSEMBLER

### D.1 General

| | |
|---|---|
| H-Group | means two consecutive frames. |
| W-Group | means four consecutive frames (one N-5 word). |
| S-Group | means eight consecutive frames and are used for symbols only. |

Now to the different control numbers:

### D.2 Feed

| | |
|---|---|
| Octal value | : 0 |
| Comparison with NORD-1 BRF | : FEED |
| Consists of | : <FEED> |
| Explanation | : Ignored |

### D.3 Increase LOC Counter

| | |
|---|---|
| Octal value | : 1 |
| Comparison with NORD-1 BRF | : AFL |
| Consists of | : <AFL> <H-GROUP> |
| Explanation | : $H_1$ + (CLC) → (CLC)   NB! No zero fill $H_1$ may be negative |

### D.4 Load one N-5 Word

| | |
|---|---|
| Ocatal value | : 2 |
| Comparison with NORD-1 BRF | : LF |
| Consists of | : <LF><W-GROUP> |
| Explanation | : - If 'add flag' is OFF (see below), then $W_1$ → ((CLC)), (CLC) +1 → (CLC)<br>- If 'add flag' is ON, then $W_1$ + ((CLC)) → ((CLC)), (CLC) +1 → (CLC) and 'add flag' is turned OFF. |

## D.5 EXT

| | |
|---|---|
| Octal value | : 3 |
| Comparison with NORD-1 BRF | : REF |
| Consists of | : <REF><S-GROUP> |
| Explanation | : - If SYMBOL is <u>not</u> defined, then add SYMBOL to UNDEFINED symbol table with a notification that it is used in loc. (CLC).<br><br>- If SYMBOL <u>is</u> defined, then<br>    - if 'add flag' is OFF, then value (SYMBOL) $\rightarrow$ ((CLC)) and 'add flag' is turned ON;<br>    - if 'add flag' is ON, then value (SYMBOL)+((CLC)) $\rightarrow$ ((CLC)) |
| Comment | : The expression<br>    OLE+5<br>where OLE is an external symbol is output as<br>    <REF><S-GROUP><LF> <W-GROUP><br>Here the S-GROUP contains the symbol OLE and the W-GROUP contains the value 5. |

## D.6 REF

| | |
|---|---|
| Octal value | : 4 |
| Comparison with NORD-1 BRF | : ENTR |
| Consists of | : <ENTR><S-GROUP><H-GROUP> |
| Explanation | : SYMBOL is entered into DEFINED SYMBOLS TABLE with a value equal to<br>    $H_1+(PB)$<br>The UNDEFINED SYMBOL TABLE is then scanned, and for each occurrence of SYMBOL in this table, the following steps are performed:<br>    - value of SYMBOL is added into location referenced;<br>    - the entry is erased from the U.S.T. |

## D.7 LIB

| | |
|---|---|
| Octal value | : 5 |
| Comparison with NORD-1 BRF | : LIBR |
| Consists of | : <LIBR><S-GROUP><H-GROUP > |
| Explanation | : Identical with ENTR |
| Comment | : LIBR denotes the entry point of a library routine. |

## D.8 END

| | |
|---|---|
| Octal value | : 6 |
| Comparison with NORD-1 BRF | : END |
| Consists of | : <END > |
| Explanation | : $(CLC) \rightarrow (PB)$; end of loading |
| Comment | : No checksum is provided! |

## D.9 Set Location Counter

| | |
|---|---|
| Octal value | : 7 |
| Comparison with NORD-1 BRF | : SFL |
| Consists of | : <SFL><W-GROUP > |
| Explanation | : $W_1 \rightarrow (CLC)$ |
| Comment | : Not produced by the assembler, but implemented to ease the production of memory dumps. |

## D.10 Load a Sequence of N-5 Words

| | |
|---|---|
| Octal value | : 10 |
| Comparison with NORD-1 BRF | : LNF |
| Consists of | : $\text{<LNF> <H-GROUP>}\underbrace{\text{<W-GROUP> --- <W-GROUP >}}_{\text{Numbered by H-Group!}}$ |
| Explanation | : $W_i \rightarrow ((CLC))$, $(CLC)+1 \rightarrow (CLC)$ $\quad i = 1, \ldots, H.$ |
| Comment | : See SFL above! |

## D.11 Load one N-5 Word and relocate it

| | |
|---|---|
| Octal value | : 11 |
| Comparison with NORD-1 BRF | : LR |
| Consists of | : <LR> <W-GROUP> |
| Explanation | : As for LF, except $W_1$+(Program Base) → ((CLC)) |