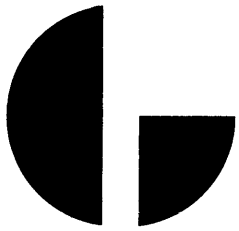


DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100

PROGRAM

Assembler

TAPES

Binary: 091-000002

093-000017-02

CONTENTS

INTRODUCTION

The NOVA assembly program allows the programmer to write a source program in a symbolic, mnemonic language using the English alphabet, numerals, and other common characters. The assembler takes the source program as input (or more precisely the ASCII codes for the characters that make up the source program) and assembles it into an object program. The output of the assembler is a complete listing of the source program on some output device and a tape of the object program ready for loading into memory by a binary loader.

1 THE ASSEMBLY LANGUAGE

The assembler recognizes the codes for all ASCII characters, but null, line feed, rubout and form feed are transparent. The first three can be used in any way in the input and the assembler responds to the input tape as though those characters were not there at all; hence blank frames on the tape have no effect, and mistakes can be overpunched with rubouts. A form feed is equally transparent with respect both to the assembly of the object program and to the characters that appear in the listing, but it can affect the format of the listing. Throughout this manual a reference to "any character" means any ASCII character except these four. Of the remaining characters many can be used only to refer to themselves, *ie* to supply their own codes to the assembler rather than being used as symbols to represent something else.

Hence the set of characters in the symbolic language is a subset of the ASCII character set, and this subset is listed in Appendix A. Basically the characters in the language are used as operators, as punctuation, as elements in numbers, and as elements in symbols that provide instructions

to the assembler. *Integers* and *numeric symbols* are character strings used as numbers. An integer, which specifies its own value, is a string of numerals. A *symbol* is a string of letters, numerals and periods whose value is either predefined (an *initial* symbol) or is defined by the source program. Symbols may be numeric and/or operational. An *operational symbol* tells the assembler to do something, purely operational symbols are called pseudo-ops. A symbolic address is an example of a purely numeric symbol. Many symbols, such as the instruction mnemonics, are numeric in that they represent numbers, but they are also operational as they provide information to the assembler concerning the characteristics of the statements in which they appear.

Integers and symbols are the basic units or *atoms* of the language. There are also several special atoms that do not satisfy the definitions for integers and symbols. A double quote combined with any character can be used just as though it were an integer whose value is the code for the characters. The special atoms also include the characters @ and # that are used for indirect addressing and to inhibit loading in an arithmetic and logical instruction. These characters can be used only in certain statements, but they are completely transparent in relation to grammatical structure the assembler responds to one of these in terms of assembling an instruction, but ignores it completely when determining the structure of the statement in which it appears.

Operators are characters that specify arithmetic and logical relations among numbers, *ie* integers and numeric symbols. A string of integers and symbols combined by operators is an *expression*. Some characters are used as *punctuation* to begin and end expressions, statements and comments, and to specify how parts of the source program are to be interpreted.

The language has a formal hierarchy. Certain characters can be combined into intergers and symbols; integers and numeric symbols can be combined with operators into expressions. Using punctuation, expressions and purely operational symbols can be combined into *statements*. These are the fundamental macroelements of the language: they provide instructions to the assembler, they define all symbols that are not initial, and they specify both the values of the words in the object program and the memory locations that will eventually receive them. Storage word statements and certain pseudo-op statements can generate output words for the object program. (Storage word statements are of two types, *data* statements and *instruction* statements.) Label statements, equivalence statements and the remaining pseudo-op statements are used to define symbols, locate the object program and control the assembly. Accompanying the statements are *comments* which provide commentary on the source program.

As mentioned above, the characters null, line feed and rubout are totally transparent: the assembler completely ignores them and responds to a character string containing them as though they simply were not there. A form feed is recognized by the assembler but only for format purposes (§2); in any other respect it is just as transparent as a rubout. The characters @ and # are also transparent with respect to grammatical structure even though they have a very definite effect on the generation of the object program; when one of these characters appears in a statement the assembler responds to its presence only after evaluating the entire statement as though the character were not there. Among the characters used for punctuation, comma, space and tab, which are used to separate expressions in a statement, are grammatically identical and can be used interchangeably. Moreover in a string of these characters, all after the first are redundant;

after encountering one such character, the assembler ignores them until some other character appears. Following a carriage return all commas, spaces, tabs and further carriage returns are ignored until some other character appears.

At the micro-level the source program is a long string of characters, but at the macro-level it is a string of lines separated by carriage returns. A single line may contain a comment, any number of label statements, but no more than one statement of another type. In other words a carriage return may be followed by a comment or any kind of statement; a label statement may be followed by a comment or any kind of statement including another label statement; but any statement other than a label statement must be followed either by a carriage return, which starts a new line, or by a comment, which in turn is terminated by a carriage return.

2 FORMAT

The NOVA assembly language is format free. However, the listing of a symbolic source program has a very definite format. A listing is the output produced when the characters that comprise the source program are printed. The teletype is the usual output medium, but a listing can be obtained on the line printer, and the string of characters in the listing can be punched in paper tape. The format of a listing is its visual appearance with respect to horizontal and vertical spacing, *ie* the use of spaces, tabs (tab settings) and carriage returns.

Some format is intrinsic to the language because these format characters are used as punctuation, and the source program is automatically formatted into lines by the requirement that sometime after any statement other than a label statement, a carriage return must appear before another statement

can be given. Furthermore these characters can be used expressly to format the listing: all redundant spaces, tabs and carriage returns are interpreted only with respect to the listing format. *Eg* a logically redundant carriage return produces a blank line in the listing (although line feeds are ignored in the input the assembler automatically follows every carriage return with a line feed in order to properly space paper in the listing device).

Within broad limits, the programmer is free to determine the format of the listing for his program. All of these lines are identical as far as the assembler is concerned, *ie* they differ only in format but are identical in grammatical structure.

```
LABEL:  ADD#    2,3,SZR          ;SKIP IF SUM ZERO
LABEL:ADD,2,3,SZR#;SKIP IF SUM ZERO
        LABEL:ADD 2 3 # SZR      ;SKIP IF SUM ZERO
LABEL:,,,,#ADD,                  ,2,     3       SZR;SKIP IF SUM ZERO
LABEL   :AD#D,2,3, SZR   ;SKIP IF SUM ZERO
```

A common practice is to divide each line into four columns by means of three tab settings, using the left column for labels, the second column for all other statements with the arguments of an instruction mnemonic starting at a second tab setting, and the right column for program comments. This is the format of the first example above. If the listing device does not have automatic tabbing (*eg* the ASR33), the assembler simulates tabs by spacing to the nearest assembler defined tab position (always leaving at least one space). These positions are every eight columns, *ie* columns 9, 17, 25, ...

Although the form feed character is completely transparent as far as the assembly is concerned, it does affect the listing format. The assembler puts a form feed (and hence starts a new page in the listing) before any line

in which that character is encountered. If the device is not equipped for form feeding, the function is simulated by line feeds, sixty lines per page.

In producing a listing the assembler actually prints out more than just the source program. In each line of the listing the assembler first prints one-letter mnemonics (flags) indicating errors that have been made by the programmer, then the address of the location that will contain the object word (if any) generated by a statement in the line, then the contents of that location (or if no storage word is generated, the value of whatever statement does appear in the line), and finally the line of the source program as formatted by the programmer. If the first instruction statement given above is assembled to be stored in location 3414, the line would appear in the listing as follows:

```
03414 157014 LABEL:  ADD#    2,3,SZR          ;SKIP IF SUM ZERO
```

Following the program the assembler lists the values of the symbols defined by the programmer.

3 INTEGERS

An integer is a number computed in any radix from two to ten. The assembler converts each integer into one 16-bit unsigned number. The decimal integers 0 to 32767 yield the octal numbers 000000 to 077777, the decimal integers 32768 to 65535 convert to 100000 to 17777. Using twos complement conventions the program may treat the former words as positive numbers, the latter as negative. (The programmer can also generate signed numbers by using integers with operators as discussed in §6.)

An integer is any string of numerals that is preceded and followed by an operator or punctuation character and is neither in a program comment nor in a text string unless enclosed by angle brackets (§12). *Eg* the four strings

```
3      38      99      12345678
```

are all integers. (In all examples such as the above it is to be assumed that appropriate delimiting characters, such as commas, spaces or operators, precede and follow each example.) But the three character strings

31.27 66A A123

are not: the first two are illegal and would be flagged as number errors (N); the third is actually a symbol.

The assembler assumes that all integers are octal unless the programmer gives a radix pseudo-op to specify otherwise (§12). An integer that contains any numeral greater than or equal to the current radix is flagged as a number error. An integer greater than or equal to 2^{16} is also flagged and is evaluated modulo 2^{16}.

4 SYMBOLS

A symbol may have either or both of two properties: a numeric symbol represents a 16-bit number; an operational symbol conveys information to the assembler. Some symbols are already *defined* before the assembly starts; these are known as initial symbols and include the instruction mnemonics and pseudo-ops. Other symbols can be *defined* in the source program as labels (which represent addresses), as other purely numeric symbols, or as operational symbols that function like the instruction mnemonics. Operational symbols can be *used* to tell the assembler to do something; numeric symbols can be *used* as numbers in expressions. A symbol with both properties can be used to initiate an instruction statement. It is then used as a number in evaluating the statement as well as being used to tell the assembler how to evaluate it. The difference between a numeric symbol and an integer is that an integer specifies its own value, whereas the value of a numeric symbol must be looked up during assembly.

Any string that begins with a letter or period and is composed entirely of letters, numerals and periods is a symbol if it is preceded and followed

by an operator or punctuation character and is neither in a program comment nor in a text string unless enclosed by angle brackets (§12). A period that by itself obeys these conditions is a special single character symbol whose value, which is determined each time it is used, is equal to the current contents of the location counter (§7). The character strings

A .Z. .123 M12345678 . G.1 ABC

are symbols (the fifth is the special location symbol), but the strings

1.27 123 LA$B

are not: the first two do not begin with a letter or period (the second is actually an integer), and the last contains an illegal character. Although the assembler would flag the last string for a bad character ($), it would interpret the string as two separate symbols. But depending upon the type of statement in which the string occurs, this interpretation would usually lead to other errors as well.

Although a symbol can have any number of characters, the assembler uses only the first five to differentiate among them; in other words, all symbols whose first five characters are the same are indistinguishable to the assembler. Hence

BITMASK BITMA.7 BITMAQPRXJSK

are treated as the same symbol and can be used interchangeably. Long symbols are often used for clarity, but caution must be taken to ensure that symbols that are meant to be different actually differ in the first five characters.

The assembler will accept the codes for lower case letters as input, but in symbols it simply translates them into upper case. Hence all of these symbols as source program input

ABCD ABCd abcd AbcD abCd

are equivalent to ABCD, which is the only form that appears in the listing.

5 SPECIAL ATOMS

Atoms in the assembly language correspond to words in a natural language. They are the strings of characters that are combined using operators and punctuation into epxressions and statements. Besides integers and symbols, there are a few special atoms that have some of their properties but which contain characters that cannot be used in integers and symbols.

The character pair

"*x*

where *x* is any character other than rubout, line feed, form feed or null, is interpreted by the assembler as an integer whose value is the 7-bit ASCII code for the character *x*, provided the pair otherwise satisfies the conditions given for an integer. Hence giving the string

";

is the same as giving the octal integer 73, which is converted into the octal word 000073. The character *x* is recognized only to the extent of using its value as an integer, and the preceding double quote destroys whatever operational value it may otherwise have, *eg* as punctuation or as a user defined symbol.

The other two special atoms are the symbols @ and #: the former is used to place a 1 in the indirect bit of a memory reference instruction or address word, and can appear only in a statement that generates an output word of these types; the latter is used to place a 1 in the no-load bit of an instruction statement of that type. These atoms are completely transparent with respect to the overall structure of any statement in which either appears and with respect to the structure of any other atom in the statement. The appearance of either @ or # any number of times in a given statement

is equivalent to its appearance only once, and its effect is exactly the same no matter where it appears in the character string that makes up the statement. The assembler first evaluates the entire statement as though the special atom were not there at all, and then ORs a 1 into the appropriate bit of the 16-bit result as indicated by @ or #. Hence all of these character strings are interpreted by the assembler as being the same integer:

@4673 46@73 4673@ 4@67@3

and all of these are interpreted as the same symbol:

#ADDZL AD#DZL ADDZL# A#DD#ZL

6 OPERATORS AND EXPRESSIONS

Operators are characters that specify arithmetic and logical relations among integers and symbols; both types of relations can be intermixed in one expression. An expression is any series of integers and numeric symbols separated by operators. The term "expression" always includes the case of an integer or a symbol standing alone. As with all integers and numeric symbols, an expression has a 16-bit value, which the assembler computes by performing the indicated logical and arithmetic operations from left to right.

An operator specifies an operation to be performed on the operands at either side of it. The operand at the left is all of the expression at the left, *ie* that part of the whole expression from the beginning to the preceding integer or symbol, the operand at the right is the next integer or symbol. Logical operators work bitwise on pairs of operands; arithmetic operators treat operands as numbers. Note that operands are intrinsically neither arithmetic nor logical: they are simply 16-bit numbers that are treated in different ways.

The assembler interprets the following six characters as operators to specify two logical and four arithmetic operations with *no* check for overflow.

Operator	*Operation*	*Interpretation of Operands*
+	Addition	Unsigned 16-bit integers
-	Subtraction	Unsigned 16-bit integers
*	Multiplication	Signed twos complement integers; result is low order word
/	Division	Signed twos complement integers; result is one word, unrounded
&	Logical AND	16-bit logical words
!	Logical OR	16-bit logical words

The plus and minus sign are additive operators, the others are product operators. An additive operator may take either one or two operands, but in the former case the operator must be at the left in order to be meaningful. Actually the assembler assumes a zero operand at the beginning of any expression that begins with an operator and at the end of any expression that ends with an operator, but this can cause difficulty only with product operators--it has no effect on additive operators. Consequently

```
+A
```

being equivalent to

```
0+A
```

is alright; the operator in

```
A-
```

is meaningless but not illegal, and the expression is equivalent to A. Note that an integer that is used to produce a negative number must have a magnitude less than or equal to 2^{15}: *eg* the expression

```
-100001
```

is not evaluated correctly but is not flagged as an error since there is no overflow check. The expression $-x$ where x is greater than 2^{15} is evaluated as $2^{16}-x$, which results in a positive number less than 2^{15}. In the example given, the evaluation is 077777.

Expressions are evaluated from left to right taking one operand at a time; in evaluating

```
A+B/C
```

the assembler adds A to B and then divides the sum by C. If two operators are contiguous, the assembler assumes a zero operand between them. For a string of additive operators this means that only the final one is significant:

```
A+-+-B
```

is interpreted as

```
A+0-0+0-B
```

which is equivalent to

```
A-B
```

But with product operators you lose:

```
A*-B
```

is interpreted as

```
A*0-B
```

which is simply -B. To multiply A by -B the programmer must either give

```
-B*A
```

or define some symbol C as equal to -B and then use the expression

```
A*C
```

7 LOCATION COUNTER

As the assembler translates the source program into an object program, it not only generates the object words, but also generates information as to where they will be stored; for this purpose the assembler keeps a location count. Whenever a storage word is generated, it is assigned to the location addressed by the current contents of the location counter.

At the start of an assembly, the assembler initializes the counter to location 0. During assembly the contents of the counter can be altered in several ways:

The source program can set the counter to any desired 15-bit address by means of a location statement (§12).

Every time a storage word is generated in the object program, the counter is incremented by one. Hence unless something else changes the counter, words are assigned to consecutive memory locations. (The location following 77777 is 00000.)

At the appearance of the pseudo-op .BLK the counter is incremented by the value of the argument of the pseudo-op (§12).

The period, when used alone, is a special symbol whose value is equal to the current contents of the location counter. Thus

```
LDA   3,.+6
```

is equivalent to

```
LDA   3,6,1
```

If the instruction is assembled at location 1215, it is also equivalent simply to

```
LDA   3,1223
```

8 COMMENTS AND STATEMENTS

As previously mentioned, a source program can be regarded as one long character string. Except for redundant carriage returns, tabs, spaces and commas, every character in the string either is part of a comment or statement, or terminates a comment or statement.

A comment is not really part of the source program because it cannot affect the generation of the object program. Its only function is in conjunction with the source program listing--a comment presumably explains something related to the portion of the program where it appears. A semicolon as a statement terminator or as the first significant character

following a statement or comment terminator indicates the beginning of a comment; the comment terminates with the next carriage return. Any character except carriage return, but including semicolon, can be used in a comment. Of course a control character produces no printable output--it has its given effect (if any) on the listing device at the point that it appears. (Remember, a form feed is executed prior to the line in which it appears and cannot be part of a comment).

Statements in the assembly language correspond to the statements or sentences in a natural language. A statement either defines a symbol, generates a word in the object program, or supplies information to the assembler. The next three sections describe the four types of statements: label statements, equivalence statements, storage word statements, and pseudo-op statements.

A statement terminator is a character that ends a statement but is not itself part of the statement. No character is used to indicate the beginning of a statement. Instead a statement is assumed to begin with the first significant character that follows a statement or comment terminator, provided this character is not a semicolon (which indicates the beginning of a comment). A statement that contains a single undefined symbol terminated by a colon is a label statement. Every other statement is terminated by a semicolon or a carriage return. An equivalence statement begins with an undefined symbol followed by an equal sign; a pseudo-op statement begins with a pseudo-op. A statement that is none of the above is taken to be a storage word statement, and the assembler inspects the first nontransparent atom in it to determine the type. If it begins with an integer or a purely numeric symbol, it is a data statement and can contain only one expression; if it begins with an instruction mnemonic or equivalent,

it is an instruction statement and the number of expressions it may contain depends upon the instruction class to which it belongs. In determining the structure of a statement or evaluating it, the assembler ignores all spaces, tabs and commas that immediately precede the statement or its terminator, or precede or follow an equal sign.

9 SYMBOL DEFINITION

A symbol is said to be defined if the assembler has a value for it. The value of a numeric symbol is the 16-bit number it represents; the value of an operational symbol is its meaning. Some symbols, such as the instruction mnemonics, have both numeric and operational properties. for such a symbol to be defined the assembler must both have a numeric value for it and also know its meaning. All symbols that appear in a program must be defined. The initial symbols are predefined and hence already have values at the start of the assembly. The source program can define a symbol as a symbolic address by means of a label statement, as a numeric symbol by means of an equivalence statement, or as a symbol that may have both numeric and operational properties by means of certain pseudo-op statements.

The assembly of a source program is done in two passes, *ie* the assembler goes through the entire character string twice. The first pass locates the entire program and determines the definitions of all symbols. Hence the assembler must be able to evaluate all symbol-defining statements in the first pass. This means the source program cannot use a pseudo-op or equivalence statement to define A as a function of the symbol B unless the statement that defines B appears earlier in the source program. In order to define all symbols and locate the program, the assembler must also be able on the first pass to evaluate all statements that indicate how integers

are to be interpreted or that alter the normal consecutive sequence of the location counter. Hence the assembler must be able to evaluate any expression that appears in a radix, location or block pseudo-op statement. If two or more statements define (*ie* assign values to) the same symbol, every occurrence of the symbol is flagged as multiply-defined (M).

As part of the assembler's initialization, it determines the memory size of the configuration in which it is running. This enables one version of the Assembler to run in all memory sizes efficiently, building its symbol table upward until the memory capacity is reached. An attempt by the program to define more symbols than the assembler can accomodate in the area of core set aside for them results in a symbol table error (S), and the assembler will accept no more symbol definitions.

The assembler evaluates all other statements in the second pass. Any symbol whose value is not known to the assembler when it is encountered in the second pass or in an expression that must be evaluated in the first pass is flagged as an undefined symbol (U). A symbol whose value in the second pass differs from its value in the first pass is flagged as a phase error (P).

10 LABEL AND EQUIVALENCE STATEMENTS

Only numeric values can be assigned to symbols by label and equivalence statements. These statements are evaluated in the first pass and must be used to assign values to symbols that are not defined elsewhere.

A label statement follows a carriage return or colon, consists of one symbol that has not been defined previously in pass 1, and is terminated by a colon. The statement defines the symbol, and its value is taken from the current contents of the location counter. Ordinarily a label statement is used in conjunction with a storage word statement. If the latter

immediately follows the former, the label provides a (symbolic) address for the memory location that will receive the storage word when the object program is loaded. If the storage word statement

```
LDA   2,30
```

is immediately preceded by a label statement, say

```
LOOP:
```

ie if the coding is

```
LOOP:   LDA   2,30
```

or equivalent, then the storage word statement

```
JMP   LOOP
```

is assembled to produce a jump to the same location that receives the storage word LDA 2,30 (provided of course that location LOOP is in page zero or within range of the location containing the JMP LOOP (see §11)). A previously defined symbol terminated by a colon is recognized as a label statement, and the symbol is redefined and flagged (M). A label-type statement containing other atoms besides an undefined symbol is flagged as a colon error (C).

An equivalence statement follows a carriage return or colon and uses an equal sign to define the symbol at its left by assigning to it the value of the storage word statement at its right. These are all legal equivalence statements.

```
A = 342
B =A/2
C= SZC+17*A/11-B
D = LDA 2,35@,3
E=ADDZ-SNC
```

The symbol at the left must be previously undefined in pass 1, and the statement at the right must be capable of evaluation in pass 1, *ie* any

symbols in it must already have been defined (an undefined symbol is flagged as a equivalence error (E)). The statement at the right of the equal sign is not really a storage word statement in that the assembler does not actually generate a storage word from it, but it must be recognized by the assembler as equivalent to such a statement. Note that in the last example, the statement at the right is recognized as a storage word statement for a format error (F) but would assemble it correctly, *ie* would assign the actual value of the expression at the right to the symbol at the left. An equivalence statement terminates with the first semicolon or carriage return, but any expression following a complete storage word statement after the equal sign is ignored. Any number of tabs, spaces or commas at either side of the equal sign are also ignored.

Neither a label statement nore an equivalence statement has any effect on the location counter. Beginning at location 1322,

```
        LDA 1,.
        A=.
        LDA 2,.
B:      C:
D:      LDA 3,.
        A
```

is assembled as equivalent to

```
        LDA 1,1322
        A=1323
        LDA 2,1323
B:      C:
D.      LDA 3,1324
        1323
```

when B, C and D are all assigned the value 1324.

11 STORAGE WORD STATEMENTS

A storage word statement generates the output for one word to be stored as part of the object program. Except for a text or end pseudo-op statement, only this type of statement actually produces output, although other types can affect the value of the 16-bit word produced. The following are typical storage word statements.

```
135602
ISZ     @A+C,2
STA     3,D
COM#    1,1,SZR
A+B/C*D
@3720+E
DIAS    2,PTR
HALT
```

The current contents of the location counter designate the memory location that is to receive the word when the object program is loaded. The counter is incremented every time a storage word statement is processed, so the words generally are assigned to consecutive locations unless the counter is changed by a location or block pseudo-op statement.

A statement that is not a label statement and does not contain an equal sign or a pseudo-op is assumed to be a storage word statement that is terminated by the first semicolon or carriage return. The assembler examines the first nontransparent atom in the statement to determine the type, and hence the maximum number of expressions or *fields* the statement can contain and the minimum number it must contain. A statement with fewer than the minimum or more than the maximum is flagged for a format error (F), but the assembler ignores any expressions beyond the maximum allowed in it.

A transparent atom can appear anywhere from the first character to the last before the terminator, when one is used, the assembler first evaluates the statement without it, and then adjusts the result for the special atom by ORing a 1 into the appropriate bit (hence it has no effect if the bit is already 1).

Data

If the first nontransparent atom in a storage word statement is an integer or a purely numeric symbol, the assembler takes it as a data statement containing a single expression. In the above examples the first, fifth and sixth are data statements. In such a statement the special atom @ can be used in generating a full word indirect address. Since it has its effect after the statement is evaluated, all of these data statements have the same value,

```
102644
102644@
2644@
1322*2      3@
```

although the last one is flagged for a format error. Remember that the special atom is neither an integer, a symbol, nor an operator, and therefore cannot be part of an expression. Hence either of these,

```
100000@
@0
```

is a data statement whose value is 100000, but this,

```
@
```

is not. In other words a storage word statement must contain at least one expression--@ by itself does not suffice.

A statement being taken as a data statement does not mean that the object word generated by it is necessarily an operand in the program. A data statement is simply a way of representing a 16-bit value; it need not be used as an operand anymore than a number generated by an instruction mnemonic need be executed as an instruction. *Eg* the first example of a storage word statement given at the beginning of this section,

135602

if executed as an instruction would be equivalent to

INC2 1,3,SZC

Instructions

If the first nontransparent atom in a storage word statement is an instruction mnemonic (or equivalent), the assembler takes it as an instruction statement, determines the class to which it belongs (memory reference, with or without accumulator; arithmetic and logic; input-output, with or without accumulator) and therefore the number of fields in it. In general an instruction statement is made up of a mnemonic field followed by a number of argument fields, the standard procedure is to separate the argument fields from the mnemonic field by a tab and the argument fields from each other by commas, but since tab, space and comma are equivalent, they can be used arbitrarily as field separators. Every field is an expression which is evaluated in the normal way. The mnemonic field should be simply the mnemonic, which specifies only certain bits in the instruction word, but results in a 16-bit value (*eg* ADDZL is evaluated as 103530). A mnemonic field containing more than just a mnemonic is flagged as a format error but is evaluated correctly. In any event the expression must begin with a mnemonic; *eg* one could give

ADDZL+400

which (although flagged with an F) is equivalent to

103130+400

ie to ANDZL. The argument fields represent other parts of the instruction word, such as an accumulator address or a skip function; and their effects are limited to those parts of the word--each argument field is evaluated as a 16-bit number, but the assembler evaluates the total statement by taking only the necessary low order bits from each argument value and ORing them into the appropriate bits for the total statement value. If the statement contains a transparent atom (which is not regarded as a field even though it represents a specific part of the instruction word), its value is ORed into the result after evaluating all fields.

Although the assembler masks out unnecessary bits in the values of field expressions, it flags as a field overflow error (O) any AC address or index field whose value is greater than 3, any skip field whose value is greater than 7, and any device field whose value is greater than 63. The assembler flags as a format error (F) any instruction statement that contains a transparent atom when none is allowed. An overflow error also results if any argument field requires the assembler to place a nonzero number in any field of the storage word that has already received a nonzero number due to the evaluation of the mnemonic field; *eg* except for incorrect format in the second, these two statements are equivalent:

```
AND     0,2,SKP
AND+1   0,2
```

so this statement results in both format and overflow errors:

```
AND+1   0,2,SKP
```

Memory Reference. A statement for an instruction that references memory has one or the other of these forms depending upon whether it requires an accumulator.

Mnemonic Address, Index (optional)

Mnemonic Accumulator, Address, Index (optional)

The mnemonics for these two forms are as follows.

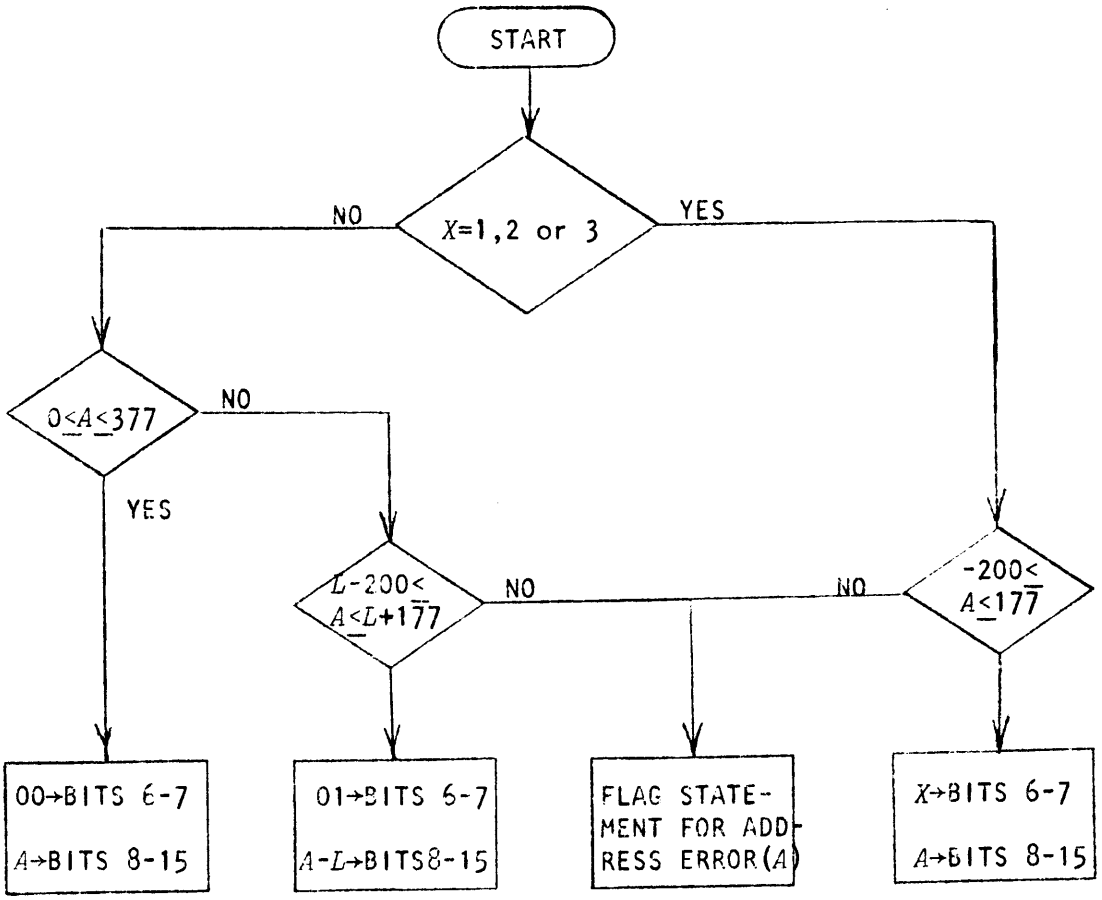
Without Accumulator	*With Accumulator*
JMP	LDA
JSR	STA
ISZ	
DSZ	

The transparent atom @ may be used to indicate indirect addressing; it is usually placed immediately before the address field.

The index field determines the action the assembler takes with respect to the address field. Let L be the current value of the location counter, A the value of the address field, and X the value of the index field.

Index	*Action*
0 *or* Blank (*ie no expression*)	If $A \leq 377$, place 00 in bits 6-7 and A in bits 8-15 (page zero addressing). If $L - 200 \leq A \leq L + 177$, place 01 in bits 6-7 and $A - L$ in bits 8-15 (relative addressing).
1, 2 *or* 3	If $-200 \leq A \leq 177$, place X in bits 6-7 and A in bits 8-15.

If the condition associated with a given index value is not satisfied, the assembler flags the statement for an address error (A) and places the low order bits in the displacement and index parts of the instruction word as shown in this flow chart.



Arithmetic and Logic. A statement for an instruction in the arithmetic and logical class has this form

Mnemonic Source AC, Destination AC, Skip Function (optional)

The instruction and skip mnemonics for this class are as follows.

Instruction			*Skip*
COM NEG MOV INC ADC SUB ADD AND	L R S Z O C	L R S	SKP SZC SNC SZR SNR SEZ SBN

The transparent atom # may be used to inhibit the processor from loading the instruction result into the destination accumulator; it is usually placed immediately after the instruction mnemonic.

Input-output. A statement for an instruction in the in-out class has one or the other of these forms depending upon whether it requires an accumulator.

Mnemonic *Device*

Mnemonic *Accumulator, Device*

The mnemonics for these two forms are as follows.

Without Accumulator	*With Accumulator*
NIO{S, C, P}	DIA, DOA, DIB, DOB, DIC, DOC {S, C, P}
SKPBN	
SKPBZ	
SKPDN	
SKPDZ	

Special Mnemonics. The assembler also recognizes some special instruction mnemonics which combine a basic in-out code with the central processor device code. This eliminates the need for a device field, and two of them even eliminate the usually required accumulator field.

Special Mnemonic	*Equivalent*		*Required Arguments*
READS	DIA	0,CPU	Accumulator
IORST	DICC	0,CPU	None
HALT	DOC	0,CPU	None
INTEN	NIOS	CPU	None
INTDS	NIOC	CPU	None
INTA	DIB	0,CPU	Accumulator
MSKO	DOB	0,CPU	Accumulator

Hence the assembler recognizes

```
READS   3
```

as a storage word statement equivalent to

```
DIA     3,CPU
```

and ignores anything after one argument field in a statement beginning with READS.

The pseudo-ops discussed in §13 allow the programmer to define symbols that will then be accepted by the assembler as equivalent to instruction mnemonics.

Floating Point Instructions. The programmer must remember that it is illegal for him to define symbols that are identical to the initial symbols. The initial symbols include the instruction mnemonics listed in Appendix D of How to Use the Nova and also the floating point instruction mnemonics that are assembled like ordinary instructions but for execution by an interpretive routine. The floating point mnemonics are explained in the writeup of the Floating Point Interpreter program and are listed below.

Instruction Mnemonics		*Option Mnemonics*
FADD	FINI	FSGE
FALG	FISZ	FSGT
FATN	FJMP	FSKP
FCOS	FJSR	FSLE
FDFC	FLD3	FSLT
FDFCI	FLDA	FSNR
FDIV	FMOV	FSZR
FDSZ	FMNS	
FETR	FMPY	
FEXP	FNEG	
FEXT	FPOS	
FFDC	FRND	
FFDFF	FSIN	
FFIX	FSQR	
FFLO	FST3	
FHLV	FSTA	
FIC2	FSUB	
FIC3	FTAN	

12 BASIC PSEUDO-OPS

A pseudo-op is a purely operational symbol. Such symbols are commands to the assembler rather than symbolic representations of numbers; they can affect the internal operation of the assembler, generate portions of the object program, and define symbols (the last type is treated in the next section). Most pseudo-op statements have the form of a pseudo-op followed by one argument.

Radix. At the beginning of each pass the assembler starts by interpreting integers as octal. The source program can change the radix by giving a statement of the form

.RDX *Expression*

where integers in the expression are *always* interpreted as decimal. The value of the expression becomes the new radix for integer evaluation. If the expression cannot be evaluated in pass 1 or its value is less than two or greater than ten, the assembler flags the statement for a radix error (D) and continues to use the previous radix.

This example of source coding illustrates the effect of the radix pseudo-op on the *octal* values of expressions.

Location	*Value*	*Statement*	
	000002	.RDX 2	
00000	000037	101!11011	;5 ORED WITH 33
	000003	.RDX 3	
00001	000013	21+11	;7+4
00002	000006	12*12/11	;5x5÷4
	000012	.RDX 10	
00003	000115	77	
00004	000077	63	
00005	000037	9*8/3+7	

Location. Whenever the assembler encounters a statement of the form

.LOC *Expression*

it sets the location counter to the value of the expression. If the expression cannot be evaluated in pass 1 or its value exceeds $32,767_{10}$ (*ie* produces an address of more than fifteen bits), the assembler flags it for a location error (L) and *ignores* the statement. In other words an erroneous location statement has no effect on the location counter.

When resetting the counter, the programmer should be careful not to produce an overlap, as in

```
Location
00214                   NIOKO   0
00215                   LDA     0,TTSAV
00216                   INTEN
00217                   JMP     @TTSAV+1
00220           TTSAV:  0
00221                   0
00222                   0
                .LOC    220
00220           PTFIN:  STA     0,PPSAV
00221                   STA     1,PPSAV+1
00222                   STA     2,PPSAV+2
```

The assembler would assign the zero words to locations 220-222. But resetting the counter to 220 causes the next three instructions to be assigned to the same three locations with no error diagnostic. When the object program is loaded the zero data words are loaded first, but are replaced by the STAs when they are loaded. Furthermore, as soon as the program saves something in the TTSAV locations, the STAs are destroyed.

A location statement can be used to reserve a block of storage. The following example allocates a block of twenty locations for a table wherein the first location in the table is labeled TAB20, and the first location after the table is labeled TEND.

```
TAB20:  .LOC    .+24
TEND:
```

Block. This pseudo-op is used explicitly to allocate blocks of storage. At the appearance of a statement of the form

```
.BLK    Expression
```

the assembler increments the location counter by an amount equal to the value of the expression. A location error (L) results if the expression cannot be evaluated in pass 1 or its value when added to the current value of the location counter exceeds $2^{15} - 1$.

This line of source coding reserves a block of six words starting at location BLK6.

```
BLK6.   .BLK    2*3
```

End of Tape. It is sometimes necessary to continue a program onto another source tape. Upon encountering the pseudo-op

```
.EOT
```

the assembler stops the source input device and halts with 00006 in the address lights. The assembly can be continued by loading a new tape and and pressing the console continue switch.

End. The *final statement* in a source program must be one of these,

```
.END    Expression
.END
```

and the line in which it appears (including a comment if any) must be terminated by a carriage return. If the pseudo-op has an argument, its

value is taken as the starting address of the program just assembled: after reading in the object tape, the loader automatically starts the execution of the program at the location given. If there is no argument, the loader halts after loading the object program.

Caution

An end statement *must be followed by a carriage return.* Omission of this character causes the assembler to act as though it had encountered an end of tape statement instead. It will thus halt and wait for further action by the operator.

Text. To store the octal codes for a string of characters packed two to a word, the programmer can use the text pseudo-ops. The basic text statement is of the form

.TXT ξ*text string*ξ

where ξ is any character other than carriage return, space, tab, comma, null, line feed, form feed or rubout, and which does not appear in the text string. Upon encountering the pseudo-op .TXT, the assembler takes the next significant character other than a carriage return as the text delimiter, and then assigns succeeding pairs of characters to consecutive memory locations until it again encounters the delimiter. If the string contains an odd number of characters, the final one is paired with a null character; if an even number, a null word is assigned to the location following the string. This provides a convenient means for an output routine to detect the end of the string.

As usual, null, line feed, form feed and rubout are not regarded as elements in the statement; but from the time the assembler encounters the first delimiter until its second occurrence, carriage returns are also ignored (of course a carriage return preceding the first occurrence

terminates the statement). Hence when the programmer is preparing an input tape on an ASR, he can continue the text string onto additional lines on the teletype paper without introducing spurious carriage returns into the statement.

The programmer can introduce *any* character, even a rubout, into the text string by enclosing an expression for it in angle brackets; hence angle brackets cannot themselves appear as characters in the string. Upon encountering a left bracket, the assembler evaluates the expression contained between it and the next right bracket, and takes the low order seven bits of that value as the ASCII code for the next byte to be packed. Thus to store the sentence

GO TO <I'>

the programmer can give a text statement of the form

.TXT @GO TO <74>I'<76>@

or, if he cannot remember the codes,

.TXT @GO TO <''<>I'<''>>@

The example just given would appear in the listing this way.

```
.TXT @GO
 T
O
<''<>I
'<''>>
@
```

The assembler generates 8-bit bytes, made up of the 7-bit ASCII code and a leftmost bit of 0, and packs them from right to left in the storage words. Our example would thus produce the words

```
OG      047507
T       052040
 O      020117
I<      044474
>M      037116
OO      000000
```

Altogether there are four forms of the text pseudo-op that vary the disposition of the leftmost bit in the 8-bit bytes generated from the text string.

Pseudo-op	*Effect on Left Bit*
.TXT	Left bit is 0.
.TXTO	Left bit is odd parity for the byte.
.TXTE	Left bit is even parity for the byte.
.TXTF	Forces left bit to be 1.

The assembler initially packs text bytes from right to left unless the programmer gives a text mode pseudo-op. After the appearance of the statement

.TXTM *Expression*

with a *nonzero* expression, the assembler uses left-right packing for any text string it encounters. The programmer can switch back to right-left packing by giving .TXTM with a *zero* argument. *Eg* in this sequence,

```
.TXTM   0
.TXT    /A/
.TXTM   1
.TXT    /A/
```

the second statement generates the storage word 000101, the fourth generates 040400.

13 SYMBOL TABLE PSEUDO-OPS

By using certain pseudo-ops the programmer can define symbols like the special instruction mnemonics (such as ITEN and MSKO), *ie* symbols that include an instruction mnemonic and other fields of an instruction statement as well. The general form of a symbol-defining pseudo-op statement is

Pseudo-op Equivalence Statement

ie

Pseudo-op Undefined Symbol = Storage Word Statement

The simplest of these pseudo-ops, .DUSR, defines symbols which retain no operational properties; in other words it acts just like a simple equivalence statement insofar as the value of the symbol is concerned. Defining CNT by

```
.DUSR   CNT = 24
```

means that

```
STA   2,CNT
```

is equivalent to

```
STA   2,24
```

Similarly

```
.DUSR  RDR = DIAS 0,PTR
```

allows us to give

```
RDR
```

to bring in a character from the reader to AC0 and start the reader again. But we cannot give

```
RDR   1
```

A symbol defined by .DUSR has no operational properties and therefore can take no arguments; the last example would be flagged for a format error (F).

The other six pseudo-ops of this type define symbols with operational properties. Typically each pseudo-op allows the programmer to define a symbol as equivalent to an instruction statement in which certain fields are zero. The symbol is then used with arguments corresponding to the zero fields in the definition. Suppose we often have to compare the magnitudes of unsigned numbers in the accumulators. We could use .DALC for these definitions.

```
.DALC SL = SUBZ# 0,0,SZC
.DALC SLE = ADCZ# 0,0,SZC
```

Then

```
SL    x,y
```

skips if ACy < ACx, and

```
SLE   x,y
```

skips if ACy $\leq$ ACx. In other words these newly defined symbols act just like instruction mnemonics in the arithmetic and logical class. The number of arguments given with the symbol plus the number absorbed in it is equal to the number the original mnemonic takes. With a symbol defined by .DALC, a skip field is optional if none was given in the definition. The effect of a transparent atom can accompany every use of a symbol by giving it in the definition, or it can be given at the programmer's discretion when the symbol is used.

Now even though a symbol defined by .DALC has certain operational properties (specifically taking certain arguments), the storage word statement in the definition need not have any. Hence SL could just as well be defined this way:

```
.DALC   SL = 102432
```

and it would still require two accumulator arguments in use. Also, the

argument fields required in use need not be zero in the definition--the restriction is that the programmer must not attempt to put a nonzero quantity in the same storage word field twice. Thus to have a convenient symbol for testing whether AC1 is less than some other accumulator, we could define TEST by

```
.DALC   TEST = SUBZ# 0,1,SZC
```

or equivalently

```
.DALC   TEST = 106432
```

Then

```
TEST   2,0
```

skips if AC1 < AC2. Note that the required arguments must be given even if zero.

The table on page 37 lists all of these pseudo-ops, the types of symbols they can define, and for each type, the arguments that must accompany the symbol when it is used. Optional elements are indicated by square brackets.

Caution

The programmer can specify certain parts of ALC and IO instruction words by appending letters to the basic three-letter mnemonics for these instructions. This property is retained by the equivalent symbol types defined by the pseudo-ops discussed here. *Eg* if the programmer uses .DALC to define a symbol whose fourth character is S, whenever the assembler encounters a statement in which that symbol is used, it will place 1s in bits 8 and 9 of the storage word generated from the statement (just as it would if the programmer used ADDS or NEGS) regardless of the value assigned to the symbol by

the .DALC statement. Hence unless the programmer actually wishes to use this function of the assembler in generating storage words, he should avoid the following:

> Using L, R, S, Z, 0 or C as the fourth character in a symbol defined by .DALC;
>
> Using L, R or S as the fifth character in a symbol defined by .DALC and whose fourth character is Z, 0 or C,
>
> Using S, C or P as the fourth character in a symbol defined by .DIO or .DIOA.

Conversely if the programmer limits his symbols of these types to three characters,he can append the above letters to them to produce the same effects as with ALC and IO mnemonics. (In fact the instruction mnemonics are not built into the assembler--they are defined by pseudo-ops.)

Although symbols defined by .DUSR take no arguments, there is one property that all symbols defined by these pseudo-ops have in common and that differentiates them from symbols defined by label and equivalence statements. All symbols defined by pseudo-ops become initial symbols, *ie* they become initial entries in the symbol table and can be used without definition by programs that are assembled after they are defined. (This also means that a later program cannot use the same character string for some other purpose, *eg* as a label.) These symbols remain in the symbol table until the assembler is reloaded or the programmer expunges the table. Reloading the assembler reduces the table to its initial state, in which it contains only the instruction mnemonics and the *permanent* symbols, *ie* the special location counter symbol (.) and the pseudo-ops. Giving the pseudo-op

.XPNG

undefines all but the permanent symbols and recovers the space used by the expunged definitions. After expunging the table, the programmer can even define instruction mnemonics such as ADD and JMP in any way he wishes.

SYMBOL DEFINING PSEUDO-OPS

Pseudo-op	*Symbol Type Defined*	*Arguments in Use*
.DUSR	User (purely numeric)	None (can be used in any expression)
.DMR	Memory reference	[@]Address [,Index]
.DMRA	Memory reference with AC	AC, [@]Address [,Index]
.DALC	Arithmetic and logical class	[#]ACS, ACD [,Skip]
.DIO	In-out	Device
.DIOA	In-out with AC	AC, Device
.DIAC	Instruction with AC	AC (in bits 3 and 4)

14 OPERATING PROCEDURE

To assemble a source program it is first necessary to load the object tape of the assembler (a tape is included in the standard NOVA software package). Once loaded, the assembler automatically takes control and prints requests for various parameters on the teletype. The programmer supplies the necessary information by typing numerals back.

The assembler first types

IN:

in response to which the programmer identifies the source input device by typing one of the following numerals.

1	Teletype reader without parity checking
2	Teletype reader with parity checking
3	Paper tape reader without parity checking
4	Paper tape reader with parity checking
5	Teletype keyboard without parity checking

When parity is checked the assembler substitutes a backward slash (\) for any incorrect character and flags the line containing it for an input error (I).

Next

LIST:

requests the programmer to select the device on which the assembler is to list the source program.

1	Teletype ASR33 (tabs and form feeds simulated)
2	Teletype KSR35
3	Line printer
4	Paper tape punch with tape prepared for later listing on an ASR33
5	Paper tape punch with tape prepared for later listing on an ASR35

After

BIN:

is typed, select the output device on which the object (binary) tape is to be punched.

1 Teletype punch

2 Paper tape punch

The above responses identify the IO devices to be used during assembly, and at this time the source tape should be mounted on the selected input device. The assembler types out

MODE:

to determine what function to perform during the upcoming pass.

1 Pass 1 (all symbols are defined)

2 Pass 2 - Output an object tape

3 Pass 2 - Output a listing (including an alphabetical symbol list)

4 Pass 2 - Output both an object tape and a listing

5 Output an alphabetical symbol list

Note that 4 is illegal if the programmer selected the same device in response to both BIN: and LIST:. When a pass is completed, the assembler again types .

MODE:

to request the next function to be performed, if any.

If it is necessary at any time to select a new IO device, do the following:

1. Press RESET
2. Set 000002 into the data switches
3. Press START

To reassign the mode, do this:

1. Press RESET
2. Set 000003 into the data switches
3. Press START

To save the symbol table (*eg* because new initial symbols have been defined), punch a new object tape of the assembler itself after pass 1 as follows.

1. Perform pass 1 on the defining tape.
2. When the assembler finishes pass 1 it types out "MODE". Respond by typing in "1". This causes the assembler to eliminate noninitial entries from the symbol table, and it then stops since there is no source tape in the reader.
3. Using the Binary Punch Program (*qv*), punch the tape from location 000002 to the location addressed by the contents of location 000004 (location 000004 addresses the last location in the symbol table).
4. Specify 000002 as the assembler start address to be punched in the start block at the end of the tape.

APPENDIX A

CHARACTERS

Character	7 Bit ASCII	Character	7 Bit ASCII	Character	7 Bit ASCII
Null	000	4	064	I	111
Horizontal Tab	011	5	065	J	112
Line Feed	012	6	066	K	113
Form Feed	014	7	067	L	114
Carriage Return	015	8	070	M	115
Space	040	9	071	N	116
!	041	:	072	O	117
"	042	;	073	P	120
#	043	<	074	Q	121
&	046	=	075	R	122
*	052	>	076	S	123
+	053	@	100	T	124
,	054	A	101	U	125
-	055	B	102	V	126
.	056	C	103	W	127
/	057	D	104	X	130
0	060	E	105	Y	131
1	061	F	106	Z	132
2	062	G	107	Rub Out	177
3	063	H	110		

APPENDIX B

PSEUDO-OPS

Mnemonic	Effect
.BLK	Assign a block of storage
.DALC	Define an arithmetic and logical instruction
.DIAC	Define an instruction requiring an accumulator
.DIO	Define an input/output instruction
.DIOA	Define an input/output instruction requiring an accumulator
.DMR	Define a memory reference instruction
.DMRA	Define a memory reference instruction requiring an accumulator
.DUSR	Define a user symbol
.END	End of source input
.EOT	End of tape
.LOC	Assign a location counter value
.RDX	Change the number radix
.TXT	Define packed test strings in octal--force parity to 0
.TXTE	Define packed text strings in octal--compute even parity
.TXTF	Define packed text strings in octal--force parity to 1
.TXTM	Define text packing mode
.TXTO	Define packed text strings in octal--compute odd parity
.XPNG	Expunge all but the permanent symbols from the symbol table

APPENDIX C

SYMBOL TABLE

All predefined and user defined symbols are entered in a table called the symbol table. This table is origined at the end of the assembler and is upward expandable until the memory capacity of the machine being used is exhausted. Each entry in the table occupies three 16-bit words. The maximum length of a stored symbol is five characters and is represented in radix 50_8 form. This method uses the first word to store the first three characters of the symbol and eleven bits of the second word to store the last two characters of the symbol. The five remaining bits of the second word are used to define attributes of the symbol, *eg*, a memory reference instruction symbol. The third word is used to store the numeric value of the symbol.

Symbol table capacity for a 4K system is approximately 400 symbols.

Radix 50 representation is used to condense symbols of five characters into two words of storage using only 27 bits. Assume a symbol of the form:

	α_4 α_3 α_2 α_1 α_0	
α_i may be	A - Z	(26)
	0 - 9	(10)
	or .	(1)

All symbols are padded (if necessary) with nulls. Therefore, there are $38_{10} = 46_8$ possible characters. Each character can be translated as follows:

character (α_i)	translation (β_i)
Null	00
0 to 9	1 to 12
A to Z	13 to 44
.	45

If α_i translates to β_i, we can compute the following numbers:

$$N_1 = ((\beta_4 * 50)+\beta_3)*50+\beta_2$$

$$N_2 = (\beta_1 * 50)+\beta_0$$

N_1 maximum is $(50)^3$ -1 which equals 174777 and will take a maximum of 16 bits to represent. N_2 maximum is $(50)^2$ -1 which equals 3077 and will take 11 bits to represent. The symbol is thus represented by N_1 and N_2 which take 27 bits of storage.

A number of symbols exist which are permanently defined in the assembler. They cannct be eliminated by the .XPNG pseudo-op. These symbols are:

.BLK	.DMRA	.TXT
.DALC	.DUSR	.TXTE
.DIAC	.END	.TXTF
.DIO	.EOT	.TXTM
.DIOA	.LOC	.TXTO
.DMR	.RDX	.XPNG

These symbols will never appear in the symbol list following an assembly listing. Note that a second class of symbols exists (initial symbols) which have been entered in the symbol table by the operator defining pseudo-ops (§12). All of the NOVA instruction mnemonics are in this category. They are never printed in the symbol list following an assembly listing. They can be eliminated, however, by using the .XPNG pseudo-op. Care must be taken not to confuse this second class of symbols with permanent symbols when using the .XPNG pseudo-op.

APPENDIX D

ERROR MNEMONICS

Extensive examination of statement syntax takes place during both passes of the assembly in order to detect syntactic errors in the input. A statement found to be in error will be flagged with from one to three letters indicating general classes in which the error(s) fall. Statements in error during pass 1 will be printed (with flags) on the teletype. After pass 1 the user may decide whether to continue to pass 2 or to correct any errors which have occurred thus far. Statements in error during pass 2 will be printed on the teletype as well as flags appearing opposite the statements on the listing device (if any).

An alphabetical list of error codes along with examples of statements causing such errors is given on the next page.

ERROR FLAG	GENERAL CLASS OF PROBLEM	EXAMPLES - COMMENTS
A	Address error	LDA 0,400 ISZ .+317
B	Bad character	LA$L: LDA 1,23 ; $ NOT PERMITTED
C	Colon error	A+2: ; NO EXPRESSION PERMITTED BEFORE A COLON
D	Radix error	.RDX 12 ; RADIX 12 NOT PERMITTED
E	Equal error	REG= 3+B ; WHERE B IS UNDEFINED
F	Format error	ADD 2 ; NEED AT LEAST 2 OPERANDS
I	Input error	; PARITY CHECKED ON INPUT AND SOME CHARACTER WAS IN ERROR
L	.LOC error	.LOC -1 ; BIT 0 SET
M	Multiply defined symbol	A: 3 ; SYMBOL MAY APPEAR ONLY A: 5 ; ONCE IN LABEL FIELD
N	Number error	C77: 7A ; NO LETTERS PERMITTED IN A NUMBER
O	Field overflow	LDA 4.LOC ; NO REGISTER 4
P	Phase error	; VALUE OF A SYMBOL IN PASS 1 DIFFERS FROM THAT OF PASS 2
Q	Questionable line	.+.END
S	Symbol table overflow	; MEMORY CAPACITY FOR A GIVEN MACHINE HAS BEEN REACHED
T	Error in table pseudo-op	14+.XPNG ; NO EXPRESSION BEFORE A TABLE PSEUDO-OP
U	Undefined Symbol	; A SYMBOL IN OPERAND FIELD WAS NEVER DEFINED
X	Text error	LET. "CB ; ONLY ONE CHARACTER IN " ATOM 3+.TXT ; NO EXPRESSION PERMISSIBLE BEFORE .TXT

APPENDIX E

LISTING FORMAT

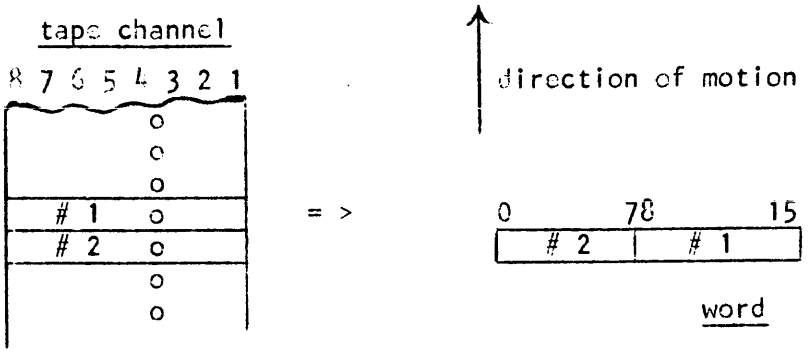
```
                    ; SAMPLE ASSEMBLY LISTING


    00000 024002 STRT:   LDA      1,.+2
    00001 050000         STA      2,.-1
    00002 157000         ADD      2,3
    00003 014020         DSZ      20
    00004 170401         NEG      3,2,SKP
    00005 042524         .TXT     *TE
    00006 052130 XT
    00007 005015 <15><12>
    00010 000000 *



          000040 ACNST=  40
          000002         .RDX     2
          000005 BCNST=  101
    00011 000135 CNST:   1011101
          000010         .RDX     8
                .

A   00012 020766         LDA      0,400    ; ADDRESS ERROR
A   00013 010717         ISZ      317,1    ; ADDRESS ERROR
UB  00014 024023 LASL:   LDA      1,23     ; BAD CHARACTER IN LABEL
MC                A+2:                    ; MULTIPLY DEFINED AND
                                          ; COLON ERROR
UUF               REG=   3+B              ; EQUIVALENCE ERROR
F   00015 143000         ADD      2        ; FORMAT ERROR
L                        .LOC     -1       ; LOCATION ERROR

MP  00016 000003 A:3                      ; PHASE ERROR
M   00017 000005 A:5                      ; MULTIPLY DEFINED
N   00020 000007 C77:    7A               ; NUMBER ERROR
O   00021 020016         LDA      4,.-3    ; FIELD OVERFLOW

R                        .RDX     20       ; RADIX ERROR
T                 2*3+.DUSR                ; SYMBOL TABLE ERRO
U   00022 030015         LDA      2,B      ; B IS UNDEFINED
X                 3+.TXT+2                ; TEXT ERROR

Q                        .+.END           ; QUESTIONABLE
```

```
---
A       000017
ACNST   000040
B       000015
BCNST   000005
C77     000020
CNST    000011
LA      000014
LAL     000014
REG     000015
STRT    000000
```

APPENDIX F

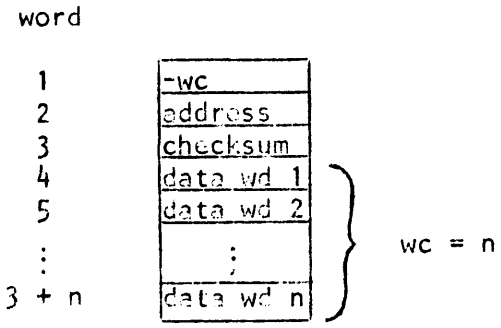
OBJECT TAPE FORMAT

The output of the assembler is an object tape. Its format is acceptable as input to the binary loader. The tape is punched in blocks separated by null (all 0) characters. There are three block types: data, start and error. The Loader reads two tape characters to form a 16-bit word. The format is as follows:



In other words, the first tape character forms bits 8-15 of the data word (Channel 8 to bit 8, etc.) and the second tape character forms bits 0-7 of the data word (Channel 8 to bit 0, etc.). The first non-null tape character signifies that start of a block. The block type is determined by the first word read. A description of each block type follows:

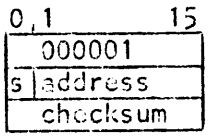
Data Block - Bit 0 of first word is a 1.



The two's complement of the number of data words in the block is given in the first word (therefore bit 0 is a 1). Normally 16 data words will be punched per data block. However, the .END and .LOC pseudo-ops may cause

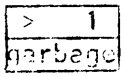
short blocks to be punched. The second word contains the address at which the first data word is to be loaded. Subsequent data words are loaded in sequentially ascending locations. The third word contains a checksum. This number is such that the binary sum of all words in the block should give a zero result. The remaining words are the data to be loaded.

Start Block - First word is 000001.



The first word contains 1. The second word uses the sign bit as a flag. If S=0, the loader will transfer to the address in bits 1-15 of the word. If S=1, the loader will halt. The third word checksum is the same as that for a data block.

Error Block - First word > 1.



The first word is greater than +1.

An error block is ignored in its entirety by the loader. All error blocks are terminated by a rubout.