Gr. 4.

DATA GENERAL
CORPORATION

Southboro,
Massachusetts 01772
(617) 485-9100

2381

PROGRAM

ABSOLUTE ASSEMBLER

TAPE

Absolute Binary: 091-000002

ABSTRACT

The Absolute Assembler translates the Nova-family absolute assembly language into machine language. The absolute binary output of the assembler is loaded with the Binary Loader.

093-000017-03

INTRODUCTION

Assembly language allows a programmer to write a source program in symbolic language. The assembler then translates this symbolic source program into a series of machine language instructions (an object program). The Data General Absolute Assembler uses a two-pass assembly process.

NOTATION CONVENTIONS USED IN THIS MANUAL

)	Represents a carriage return.
↓	Represents a form feed.
upper case letters	Parts of the format that are typed in upper case letters are literal parts of the Absolute Assembly Language, and must appear in context exactly as shown within the format.
lower case letters	Parts of the format that are written in lower case letters (and underscored) are variables indicating that the programmer substitutes an appropriate item (i.e., user symbol, expression, statement).
[ ]	Broken square brackets are used to indicate optional parts of a format.
△	A terminator (break) is one or more tabs, spaces, and/or commas.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION TO ASSEMBLY

THE ASSEMBLY PROCESS

The Absolute Assembler facilitates the coding of machine language programs. Basically, the Absolute Assembler is a program that assigns numeric values to symbols.

Certain symbols have preassigned values, e.g., the DGC instruction mnemonics. The instruction mnemonics are listed in Appendix G with their values in octal and are described in detail in How to Use the Nova Computers. The value assigned to an instruction mnemonic is the binary bit configuration recognized by the hardware for that instruction. For example, the following instruction mnemonics have the values given (in octal):

Mnemonic	Value
ADD	103000
SUBO	102440
ANDZR#	103630

Other symbols may be defined by the user. These symbols are assigned a numeric value, such as the current value of the location counter (LC), and can be referred to by other statements in the program. For example, the programmer might write the source program line:

```
LDA   0,  TEMP
```

which means "load accumulator 0 from symbolic address TEMP." LDA is an instruction mnemonic that the assembler translates into the 16-bit equivalent of 20000_8. Accumulator 0 has a value 0 that is ORed into bits 3 and 4. If TEMP is the symbolic address assigned by the user to absolute address 5, then the value 5 is ORed by the assembler into bits 13 through 15. The assembled line would be:

```
0010000000000101
```

and would be part of the object code output by the assembler. At a later time the object code could be loaded into core using the binary loader and executed.

THE ASSEMBLY PROCESS (Continued)

The assembly of a source program is done in two passes, i.e., the assembler goes through the entire source program character string twice. The first pass locates the entire program and determines definitions of all symbols. The second pass completes the evaluation of lines that could not be completely evaluated until after the first pass.

INPUT AND OUTPUT OF THE ASSEMBLER

Input to the assembler consists of one or more source programs, written in a subset of the ASCII character set. Output includes one or more of the following: a binary object program, a source program and error listing, and a symbol table listing.

The elementary scan of input by the assembler is a line-by-line read where each line includes all characters up to a carriage return or a form feed. Three characters that may appear in the input are unconditionally ignored. These characters are:

Character	ASCII Value
null	000
line feed	012
rubout	177

During translation, characters having incorrect parity are replaced by the ASCII character backslash (\ ). This character is transparent to higher level character processing, i.e., L\A is processed as LA.

The binary output program is a translation of the lines of the source program into a special blocked binary code. Most lines of source input translate into a single 16-bit (one-word) binary number for storage in core by the binary loader. Associated with each number is an absolute address. Every 16-bit word assembled is punched as two 8-bit characters on the object tape.

The listing provides a copy of the source input as well as the octal location and contents of every object word produced by the assembly process. The symbol table listing follows the program listing and is an alphabetic list of every user symbol and its value.

Any lines containing source program errors are printed on both pass 1 and pass 2 at the console with error codes. If a listing of the program is produced, the error codes will also appear on the listing. The listing contains information in the following order:

INPUT AND OUTPUT OF THE ASSEMBLER (Continued)

Column	Contents
1-3	Up to three error code letters indicating input errors in the line. The first error generates a letter code in column three, the second in column 2 and the first in column 1. Only 3 codes can be listed per line. If there are no input errors, these columns are blank.
4-8	The current value of the location counter, which is the absolute location in which the numeric output will be stored, if relevant. Otherwise, blank.
9	The column is left blank.
10-15	The numeric output line, if relevant. Otherwise, the column is blank.
16	The column is left blank.
17 . . .	The source line as written by the programmer.

An example of listing of source and object codes, error codes, and a symbol table is shown on the following page.

```
---
                  ;  SAMPLE ASSEMBLY LISTING
                  ;
   00000 024002 STRT:   LDA 1,.+2
   00001 050000         STA 2,.-1
   00002 157000         ADD 2,3
   00003 014020         DSZ 20
   00004 170400         NEG 3,2
   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
A  00013 010717         ISZ .+317,1
UB 00014 024023 LASL: LDA 1,23
MC              A+2:
UUE             REG= 3+B
F  00015 143000         ADD 2
I               ; PARITY ERROR I\ THIS COMMENT
L               .LOC -1
MP 00016 000003 A:3
M  00017 000005 A:5
N  00020 000007 C77: 7A
O  00021 020016         LDA 4,.-3
R               .RDX 12
T               2*3+.DUSR
U  00022 030015 LDA 2,B
X               3+.TXT+2
                ;
                ; SEE TABLE 8-1 FOR AN EXPLANATION
                ;    OF THE ABOVE ERRORS
                ;
O               .+.END

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

CHAPTER 2

INPUT

Assembly language source programs are made up of a series of lines. A line is all characters scanned by the assembler up to a carriage return or form feed. The assembler recognizes several types of lines; each source line must conform to a given structure, depending on its type. In addition, each line must contain characters specified as part of the Absolute Assembler character set.

CHARACTER SET

The Absolute Assembler accepts the following characters within a source program

1. Alphabetics A through Z
2. Numerals 0 through 9
3. Special Characters: ! " # & * + , - . / ; : = @ < >
4. Format Control and Line Terminators:

   horizontal tab, form feed, carriage return, space

Appendix B contains a table listing the character set with ASCII equivalents. If at any time, a lower case alphabetic is used, the assembler will automatically convert it to its upper case equivalent. The Absolute Assembler unconditionally ignores the three characters: null, line feed, and rubout. The assembler will respond to their input as if they were not present.

Any character not part of the character set will be flagged with an error on the assembly listing with the bad character code (B).

SOURCE LINES

Members of the character set are combined to form source lines. The majority of source lines cause the generation of a 16-bit value that is to occupy a memory location at execution time. Any line of this type is said to produce a storage word. The storage word has a value, usually defined by an expression or instruction, and an address. At assembly time the address assigned is the current contents of the location counter (LC). The generation of each 16-bit storage word causes the location counter to be incremented by one. Thus, in general, storage words are assigned to consecutive increasing LC values.

Several types of source lines produce storage words. Others are used to define symbols, control the assembly process, and provide instructions to the assembler.

Assembly source lines must be one of the following types:

1. Data
2. Instruction
3. Pseudo-op
4. Equivalence

Data Lines

A data line is one of the simplest in the assembly language. It consists of a single numeric expression that evaluates to an integer that is stored in a 16-bit storage word. The special atom @ (explained later in the chapter) can be used anywhere in the data line to cause the assembler to place a 1 in bit 0 of the storage word. Thus, for example, all of the following data lines have the same value.

```
102644
102644@
2644@
@1322*2        (where * is the multiply sign)
```

Instruction Lines

An instruction line is an instruction mnemonic followed by any required or optional fields. All instruction lines generate a 16-bit storage word, which provides an instruction to the assembler, such as to load an accumulator, add two accumulators, or increment an accumulator. Instructions are described in Chapter 3.

Pseudo-op Lines

A pseudo-op line must begin with a permanent symbol (except the symbol .) and may be followed by one or more required or optional arguments. Some pseudo-op lines (such as .LOC and .END) are merely commands to the assembler and do not generate either a storage word or 16-bit value. Others (such as .RDX) generate a 16-bit value but do not increment the location counter. Pseudo-ops are discussed in Chapter 4.

Equivalence Lines

One means of assigning a symbolic name to a numeric value is by equivalence. An equivalence line associates a value with a symbol; that symbol can then be used any time the value is required. An equivalence line has the form:

<u>symbol</u> = <u>expression</u>

Equivalence Lines (Continued)

where symbol is a user symbol conforming to the rules given in the section on atoms. The expression following the equals sign must be capable of evaluation to an integer on pass one. It can, for example, be an instruction since this is evaluable at pass one. The user symbol on the left must be previously undefined in pass one. Some equivalence lines are:

```
000037  A = 3 + 5 * 4 - 1
000040  ACNST = 40
000010  B = 10
020005  S = LDA 0  5
```

In an equivalence line, predefined symbols used on the right will assemble correctly whether or not they are used in a syntactically correct instruction, for example:

```
E = ADDZ - SNC
```

where the expression on the right contains symbols used in instructions but is not a true instruction. It will assemble correctly, i.e., the value of SNC (3) would be subtracted from the value of ADDZ (103020) and the resultant value (103015) assigned to E. If the expression on the right contains an undefined symbol, an equivalence (E) error results.

Labels

Any source program line can contain one or more labels. A label allows the programmer to name a storage word symbolically. Using the label, the programmer can then reference the storage word without regard for its numeric address.

A label is a user symbol, previously undefined, that must appear at the beginning of a source line and must be followed by a colon (:). Like other symbols, a label has a value; its value is that of the location counter, i.e., the address of the next storage word assembled. Since some source lines do not generate storage words, this definition is not necessarily associated with the line in which the label appears. The following source line is given the label LOOP.

```
LOOP:  ADD#  0  1  SKP
```

Then the storage word line:

Labels (Continued)

```
JMP  LOOP
```

is assembled to produce a jump to the same location that receives the storage word ADD# 0 1 SKP.

If a previously defined symbol terminated by a colon is redefined, it will be flagged with an error code M. A label containing other atoms besides an undefined symbol is flagged as a colon error, C.

Some examples of labels are:

```
00000 063511 PUTC: SKPBZ TTO
00462 063077 ERROR: HALT
00515 024420 INIT: LDA 1, ASTK
```

where the value of ERROR is 462, the value of INIT is 515, and the value of PUTC is 0.

Comments

An assembly language program can include comments to facilitate program checkout, maintenance, and documentation. A comment is not interpreted in any way by the assembler and will not affect the generation of the object program. All comments must be preceded by a semicolon (;) and terminate with a carriage return.

Upon encountering a semicolon, the assembler will ignore that which follows it until encountering a carriage return or form feed. However, a listing of the source program will generate all comments as input by the user. The following source program lines illustrate the use of comments.

```
; THIS PROGRAM CALCULATES THE ABSOLUTE VALUE OF A NUMBER IN AC0.
;
     MOVL#  0, 0, SZC ;TEST SIGN
     NEC  0, 0         ;NEGATE IF NEGATIVE
     .END              ;END OF ABSOLUTE VALUE PROGRAM
```

Source Line Formatting

Within broad limits the programmer is free to determine the format of the source lines of a program. For example, all of the following lines are identical in meaning to the assembler; they differ only in format.

```
   INC:  ADD#  2, 3, SZR             ;SKIP IF SUM = ZERO
   INC:ADD, 2,3,SZR#;SKIP IF SUM = ZERO
         INC:     ADD   2   3   SZR # ;  SKIP IF SUM = ZERO
```

(The special atom # can appear anywhere in a source line. See section on atoms.)

A common practice in writing source programs is to divide each line into four columns by means of three tab settings, using the leftmost column for labels, the second column for the beginning of the source line, the third for argument fields, and the rightmost for comments. If the listing device is not equipped with automatic tabbing (such as the ASR 33), the Absolute Assembler simulates tabs by spacing to the nearest assembler-defined tab position (and always leaving at least one space between fields). Assembler-defined tab positions are at every eight columns, that is, at columns 9, 17, 25, etc.

LOCATION COUNTER

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

> Every time a storage word is generated in the object program, the counter is incremented by one. Therefore, unless something else changes the counter, words are assigned to consecutive memory locations. (The location following 77777 is 00000.)
>
> The programmer can set the counter to any desired 15-bit address by means of a location pseudo-op, . LOC. (See page 4-2.)
>
> At the appearance of the pseudo-op . BLK, the counter is incremented by the value of the argument of the pseudo-op. (See page 4-3.)

The symbol . when used alone is a special symbol whose value is equal to the current contents of the location counter. For example:

```
01077 177771 M7:     -7
01100 001103 TABAD: .+3
```

ATOMS

An atom, the basic unit of the assembly language, is a group of characters having special meaning to the assembler. All characters, except those in comments or text strings, are interpreted by the assembler as an atom or a part of an atom. The general classes of atoms are:

- terminators
- integers
- symbols
- special atoms

These are described on this and following pages.

TERMINATORS

A terminator separates numbers and symbols from other numbers and symbols. They can be used either as operators or breaks.

Operators

Operators are a set of characters which are used with integers and symbols to form expressions that specify arithmetic and logical relations among integers and numeric symbols. Expressions are described in Chapter 3. The operators are:

Arithmetic	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
Logical	&	Logical AND
	!	Logical OR

Breaks

The terminators that are used primarily as separators, to begin and end expressions and comments, and to specify how parts of the source program are to be interpreted are:

Δ	Δ represents the class of spaces - a space, a comma, a tabulation, or any number or combination of spaces, commas, and tabulations.
:	A colon (:) is one means used to define the symbol which precedes it, for example: <u>user - symbol:</u>
=	An equals sign (=) is another means of defining the symbol which precedes it. For example: <u>user-symbol=</u>
( )	Parentheses may enclose an expression.
;	A semicolon indicates the beginning of a comment.
)	A carriage return terminates a line of source code.
↓	A form feed terminates a line of source code.

INTEGERS

Normal Format of Integers

An integer is a number computed in any radix from two to ten. The Absolute Assembler converts each integer into a 16-bit unsigned number. The decimal integers 0 to 32767 yield the octal numbers 000000 to 077777. The decimal integers 32767 to 65535 convert to 100000 to 177777. Using two's complement notation, the program may treat the former words as positive numbers, the latter as negative. (The user can generate signed numbers by using integers with operators + or - listed on page 2-6.)

An integer is any string of digits that is preceded and followed by an operator or break character and is neither in a program comment nor in a text string unless enclosed within angle brackets. The four strings:

```
3
38
99
123456789
```

are all integers. But the three character strings:

```
31.27
66A
A123
```

are not; the first two are illegal and would be flagged as number errors (N), and 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. (.RDX is described on page 4-18.) An integer that contains any numeric 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}.

Special Format of Integers

There is a special input format that is converted to the single precision 7-bit octal value for the single ASCII character following. The input format is:

"$\underline{x}$

where: $\underline{x}$ represents any ASCII character except line feed (012 octal), rubout (177 octal), or null (000).

Special Format of Integers (Continued)

Only the single ASCII character immediately following the quotation mark is interpreted. The ASCII characters null, rubout, and line feed, which are invisible to the Absolute Assembler, cannot be input using this format. However, the other ASCII characters can be represented as single precision integers in this manner.

```
000101   "A

000065   "5
```

The format can be used as an operand within an expression as shown below.

```
000103   "A+2

000026   "B/3

177751   "*-"A
```

Note that "↲ assembles as octal 15 and also terminates a line.

SYMBOLS

A primary function of the Absolute Assembler is the recognition and interpretation of symbols. Symbols are used both to direct the action of the assembler and to represent numeric values. The various types of symbols will be discussed in Chapter 3. Their source representation is given below.

a[b . . . b] break

where: a is . or a letter (A-Z).
b is ., a letter (A-Z), or a decimal digit (0-9).
break is any character that is not an alphanumeric or a period.
If more than five characters precede break, only the first five are regarded as significant.

Some examples of symbols are:

```
.ABS
ABS
A12
```

SPECIAL ATOMS

There are two atoms that are transparent during the assembly scan. The effect of these atoms upon a line occurs after the entire line has been scanned.

@ An at sign (@), or any number of at signs, appearing anywhere in a source program line of a memory reference (MR) instruction or before an expression has the following effect.

1. When the rest of the MR has been evaluated, presence of the @ sign or a series of @ signs anywhere in the instruction causes a 1 to be stored in bit 5. In the format of a memory reference instruction, bit 5 is the indirect addressing bit.

```
024020  LDA  1,20

026020  LDA  1,@20
```

2. In the format of a data word, bit 0 is the indirect addressing bit. When the expression has been evaluated, presence of the @ sign or a series of @ signs causes bit zero of the word to be set to a 1.

```
000025  25

100025  @25
```

A pound sign (#) or any number of pound signs appearing anywhere in a source program line of an arithmetic and logical (ALC) instruction has the following effect.

When the rest of the ALC has been evaluated, a 1 is stored in bit 12. (Bit 12 in the format of the ALC is the no load bit.)

```
133102    ADDL   1,2,SZC

133112    ADDL#  1,2,SZC
```

CHAPTER 3

SYNTAX

EXPRESSIONS

An expression, expression, has the format:

[operand$_1$] operator operand$_2$

where: operator is an Absolute Assembler operator.

operand$_1$ and operand$_2$ are operands which may be integers or symbols or expressions evaluating to integers. An operand preceding the operator is necessary for each operator, except for unary operators signifying plus and minus. Either unary operator may follow an operator or precede an expression.

An expression is any series of integers and numeric symbols, separated by operators, as specified in the format above. The term "expression" always includes the case of an integer or 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. Both arithmetic and logical operators may appear in an expression.

An operator specifies an operation to be performed on the operands at either side of it. 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 two's complement integers, result is the low order word
/	Division	Signed two's complement integers, result is one word, unrounded
&	Logical AND	16-bit logical words
!	Logical OR	16-bit logical words

Expressions are evaluated left to right with no implied precedence, i.e., A+B/C is evaluated as (A+B)/C. The unary operator - is permitted. Multiplication is signed, single precision. Division is signed integer, unrounded. Overflow is not checked for, and it is therefore the user's responsibility to keep his results in the range of numbers which can be handled by DGC's Nova-family computers. Note that an integer that is used to produce a negative number must have a magnitude less than or equal to 2^{15}, e.g., the expression:

```
-100001
```

is not evaluated correctly but will not be 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, resulting in a positive number less than 2^{15}. In the example given the evaluation is 077777.

The use of "@" and "#" within expressions will change the resulting value. These characters are initially screened from the expression. This screened expression is evaluated, and finally the bits corresponding to "@" and "#" are logically ORed into the value. For example:

```
@0+@0  is  100000  (not 000000)

ANDZ#-SZC  is 103416  (not 103426)
```

After the following statements have been assembled:

```
01000  000003  LABEL:  3
       000101  ANUM = "A
01001  177000  LOOP:   ADD 3,3
```

some typical expressions and their values are:

Expression	Value
LABEL+7	001007
ANUM-077	000002
"B+ANUM*4 /LABEL	000001
LOOP+ADD	104001
LABEL/4	000200
LABEL*LABEL	000000 (overflow not checked)
"0+ANDZ-SZC	103476
LABEL&LOOP	001000
LABEL!LOOP	001001

When an expression begins with an operator or when two operators appear in succession in an expression, the Absolute Assembler assumes an operand of zero, e.g.,

+A	is equivalent to	0+A
A+ -B	is equivalent to	A + 0 -B or A -B
*A	is equivalent to	0 * A or 0
A * -B	is equivalent to	A * 0 -B or -B

To multiply or divide by some negative integer or symbol, the integer or symbol must be equated to a symbol first, e.g.,

```
C= -B

A/C
```

SYMBOLS

Symbols recognized by the Absolute Assembler are classified as:

1. Permanent
2. Semi-permanent
3. User

Symbol Definition

A symbol is defined if the assembler has a value for it. The value of a symbol may be either numeric or operational. The value of a numeric symbol is the 16-bit number it represents; the value of an operational symbol is its meaning. Some symbols have both numeric and operational properties. For such a symbol to be defined, the assembler must know both the numeric value for it and also know its meaning. All symbols which appear within a program must be defined.

Permanent Symbols

Permanent symbols are defined by the assembler and cannot be altered in any way. These symbols are used for two purposes: 1) they are used to direct the assembly process; and 2) they are used to represent numeric values of internal assembler variables.

Symbols used to direct the assembly process are called pseudo-ops. Pseudo-ops are used for such purposes as setting the input radix for numeric conversions, setting the location counter, assembling ASCII text, etc. They are discussed in detail in Chapter 4, and alphabetically listed in Appendix C.

There is one permanent symbol which represents a numeric value of an internal assembler variable. This permanent symbol is . (period). The symbol period, when used alone, is a special symbol whose value is equal to the current contents of the location counter. (See page 2-5.)

Permanent symbols will not be printed as part of the user symbol table.

Semi-Permanent Symbols

Semi-permanent symbols form an important class of symbols, containing the instruction mnemonics. An instruction mnemonic signals the start of an instruction. Using appropriate pseudo-ops, symbols may be defined as semi-permanent, and as such their future use will imply further syntax analysis. For example, a symbol may be defined as requiring an accumulator. Use of this symbol causes the assembler to scan for an expression following the symbol. If not found, a format (F) error results. If found, the value of the expression determines the value of the accumulator field bit positions within the 16-bit instruction value. Instructions are discussed in the next section.

Semi-Permanent Symbols (Continued)

Semi-permanent symbols can be saved and used, without redefinition, for all subsequent assemblies. The assembler supplied by DGC contains predefined semi-permanent symbols that conform to the Nova family instruction mnemonics. A list of these is given in Appendix G. The user may eliminate these symbols and define his own symbols or he can add to the given set. (See Appendix A.)

In addition to instruction mnemonics, several other types of semi-permanent symbols are provided by DGC. These include skip mnemonics, device codes, and floating-point instruction mnemonics.

Semi-permanent symbols will, by default, not be printed as part of the user symbol table.

User Symbols

The user can define any symbol that does not conflict with the permanent or semi-permanent symbols. Symbolic definitions are used to name a location, to assign a numeric parameter to a symbol, etc. These symbols are maintained for the duration of any assembly in a symbol table that is printed after the assembly source listing as shown in Chapter 1.

INSTRUCTIONS

Instructions consist of one or more fields. The first field is an instruction mnemonic that is a semi-permanent symbol, defined by one of the pseudo-ops .DALC, .DIAC, .DIO, .DIOA, .DMR, or .DMRA as described in Chapter 4. Other fields may follow the mnemonic. Fields in an instruction can be separated by a space, comma, or tab and must conform to the field requirements for the class of semi-permanent symbol in number and type of fields.

The Nova family of computers recognizes six basic classes of instructions. Each class has a corresponding pseudo-op enabling definition of semi-permanent symbols within the class. Instructions fall into one of the six classes:

1.	Arithmetic and logical instructions.	Mnemonic defined by .DALC
2.	Input/output without accumulator.	Mnemonic defined by .DIO
3.	Input/output with accumulator.	Mnemonic defined by .DIOA
4.	Memory reference without accumulator.	Mnemonic defined by .DMR
5.	Memory reference with accumulator.	Mnemonic defined by .DMRA
6.	Instructions requiring an accumulator.	Mnemonic defined by .DIAC

An instruction is assembled in the following order:

1. The mnemonic instruction field is assembled as a 16 -bit word.

   For example, DGC-defined mnemonic ANDZL would be evaluated as 103520_8. If extra characters were appended to a mnemonic, e.g., ANDZL+1, the mnemonic would be evaluated correctly but the statement would be flagged with a format error (F).

2. Each subsequent field is evaluated and ORed into the appropriate bits of the 16-bit instruction word.

   Each field is evaluated as a 16-bit word but all bits except the low-order bits required for the field are masked. For example, an accumulator field is masked except for the low-order two bits. Although the assembler masks out unnecessary bits in fields, a field overflow error (O) will be given for such a field, e.g., accumulator field greater than 3, skip field greater than 7, etc.

3. Special atoms # and @ when present in the instruction are ORed into their appropriate bit position.

   The special atoms are not assembled until all fields have been assembled. The assembler flags as a format error (F) any instruction containing a special atom when none is allowed.

If a field requires the assembler to place a non-zero number in any bit field of the instruction that is already non-zero, the assembler will flag the statement with an overflow error (O). For example, except for the incorrect formatting of the second instruction, the following two instructions are equivalent.

```
AND    0,2,SKP
AND+1  0,2                        ;INSTRUCTION WOULD BE FLAGGED F
```

Therefore, the following instruction would result in both a format (F) and overflow (O) error·

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

The DGC-defined instructions corresponding to the different classes of instructions are described fully in How to Use the Nova Computers. The syntax required for each instruction class is given on the pages following. The semi-permanent symbols (instruction mnemonics) listed for each instruction are those defined by DGC.

Arithmetic and Logical (ALC) Instructions

An arithmetic and logical (ALC) instruction is implied when the mnemonic field is one of the following:

ADC	AND	INC	NEG
ADD	COM	MOV	SUB

The format of the source program instruction is:

```
alc-mnemonic{carry}{shift}△source-ac△destination-ac{△skip}
```

where:

alc-mnemonic	is one of the eight semi-permanent symbols listed above.	
carry	is an optional Carry bit mnemonic:	Z sets Carry to 0. O sets Carry to 1. C complements the current state of Carry.
shift	is an optional rotate/shift mnemonic:	L shifts left one bit. R shifts right one bit. S swaps bytes.
source-ac	is any legal expression evaluating to an accumulator (0, 1, 2, 3) to be used as the source accumulator.	
destination-ac	is any legal expression evaluating to an accumulator (0, 1, 2, 3) to be used as the destination accumulator.	
skip	is an optional skip mnemonic:	

SBN	skip on zero Carry and result.	SNR	skip on nonzero result.
SEZ	skip on zero Carry or result.	SZC	skip on zero Carry.
SKP	skip unconditionally.	SZR	skip on zero result.
SNC	skip on nonzero Carry.		

The atom # can be specified anywhere as a break character. If used, a 1 is assembled at bit 12 (no load bit).

An ALC instruction is represented in memory as:

1	source ac		dest. ac		alc mnemonic			shift		carry		no load	skip		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Some examples of ALC instructions are:

```
MOVL   1,1,SNC
NEGZL  1,0,SBN
ADC#   0,1
```

I/O Instructions without Accumulator

An input/output instruction without an accumulator is implied when the mnemonic field is one of the following:

NIO	SKPBN	SKPDN
	SKPBZ	SKPDZ

The format of the source program instruction is:

```
io-mnemonic△device-code
```

where:

io-mnemonic	is one of the four semi-permanent symbols SKPBN, SKPBZ, SKPDN, SKPDZ, or is NIO{pulse} where:

pulse is an optional pulse specification:

C	Clear Busy and Done bits, idling device.
S	Clear Done and set Busy, starting device.
P	Pulse the special in-out bus control line (set Busy and Done).

When device-code is the CPU (77), the meanings of S, C, and P are:

C	Clear the Interrupt On flag.
S	Set the Interrupt On flag.
P	Has no effect.

device-code	is any legal expression evaluating to an integer that specifies a device.

An I/O instruction without an accumulator is represented in memory as:

0	1	1	0	0	io-mnemonic					device-code					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Some examples of I/O instructions without an accumulator are:

```
NIOC    CPU
SKPBZ   36
SKPDN   TTR
```

I/O Instructions with an Accumulator

An input/output instruction with accumulator is implied when the mnemonic field is one of the following:

DIA	DIB	DIC
DOA	DOB	DOC

The format of the source program instruction is:

```
ioa-mnemonic{pulse}△ accumulator △ device-code
```

where:

ioa-mnemonic	is one of the six semi-permanent symbols listed above.
pulse	is an optional pulse specification:

C	Clear Busy and Done bits idling device.
S	Clear Done and set Busy, starting device.
P	Pulse special in-out bus control line (set Busy and Done).

When the device-code is the CPU (77), C, S, and P have special meanings:

C	Clear the Interrupt On flag
S	Set the Interrupt On flag
P	Has no effect

accumulator	is any legal expression evaluating to an accumulator (0, 1, 2, or 3).
device-code	is any legal expression evaluating to an integer that specifies a device.

An I/O instruction with an accumulator is represented in memory as:

0	1	1	ac		ioa-mnemonic and {pulse}					device-code					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Some examples of I/O instructions with an accumulator are:

```
DOAS   0,PTP
DIA    2,CPU
```

Memory Reference (MR) Instructions without Accumulator

A memory reference instruction without an accumulator is implied when the mnemonic field is one of the following:

DSZ	JMP
ISZ	JSR

The format of the source program instruction is:

```
mr-mnemonic△displacement△mode
              or
mr-mnemonic△address
```

where:

mr-mnemonic	is one of the four semi-permanent symbols listed above.
displacement	is any legal expression evaluating to an 8-bit integer within the range set for the particular mode as shown below.
mode	is any legal expression evaluating to an integer in the range 0-3, which determines an explicit mode of forming an effective address as follows:

Mode	Formation of Effective Address
0	Page zero or direct addressing. ($0 \leq$ displacement $\leq 377_8$) and (effective address = displacement).
1	Addressing relative to the contents of the location counter (LC). ($-200_8 \leq$ displacement $\leq +177_8$) and $[(LC) - 200] \leq$ effective address $\leq [(LC) + 177]$
2 or 3	Addressing based on contents of accumulator AC2 or accumulator AC3. ($-200_8 \leq$ displacement $\leq +177_8$) and $\left[\begin{Bmatrix} AC2 \\ AC3 \end{Bmatrix} - 200\right] \leq \text{effective address} \leq \left[\begin{Bmatrix} AC2 \\ AC3 \end{Bmatrix} + 177\right]$

address	is any legal expression evaluating to an address within the range of either mode 0 or mode 1.

Normally, modes 0 and 1 are not explicitly given to determine the mode. Instead, if only address is given, the assembler determines if this address is in page zero (mode 0) or within 256_{10} words of the location counter (mode 1). When using explicit modes 1, 2, or 3, displacement is considered a signed integer.

Memory Reference (MR) Instructions without Accumulator (Continued)

(Explicit mode 1 can be used occasionally to force LC-relative addressing; explicit mode 0 is never used.)

If address or the evaluation of displacement to an address does not produce an effective address within the appropriate range, an address (A) error is set. If mode does not evaluate to 0-3, a field overflow (O) error is given.

An MR instruction is represented in memory as:

					indirect bit										
0	0	0	mr-mnemonic		↓	mode		displacement							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Use of the special atom @ anywhere in an MR instruction causes bit 5 of MR instruction to be set. If set to 1, indirect addressing is specified, and the effective address contained in the instruction is only a pointer to another memory cell, which may contain an effective address. The memory cell pointed to by the instruction may, in turn, also be indirect, etc.

Page 3-12 contains a flow chart showing how effective addresses are formed.

Some examples of MR instructions without accumulators are:

```
DSZ  COUNT
JMP  LOOP+3
JSR  @SAVE
JMP  .+5
```

Memory Reference (MR) Instructions with Accumulator

A memory reference instruction with an accumulator is implied when the mnemonic is one of the following:

LDA STA

The format of the source program instruction is:

```
mra-mnemonicΔaccumulatorΔdisplacementΔmode
                    or
mra-mnemonicΔaccumulatorΔaddress
```

Memory Reference (MR) Instructions with Accumulator (Continued)

where:

mra-mnemonic	is one of the semi-permanent symbols LDA or STA.
accumulator	is any legal expression evaluating to 0, 1, 2, or 3.

displacement, mode, and address are the same as for MR instructions without an accumulator.

The special atom @ is used as it is for MR instructions without an accumulator.

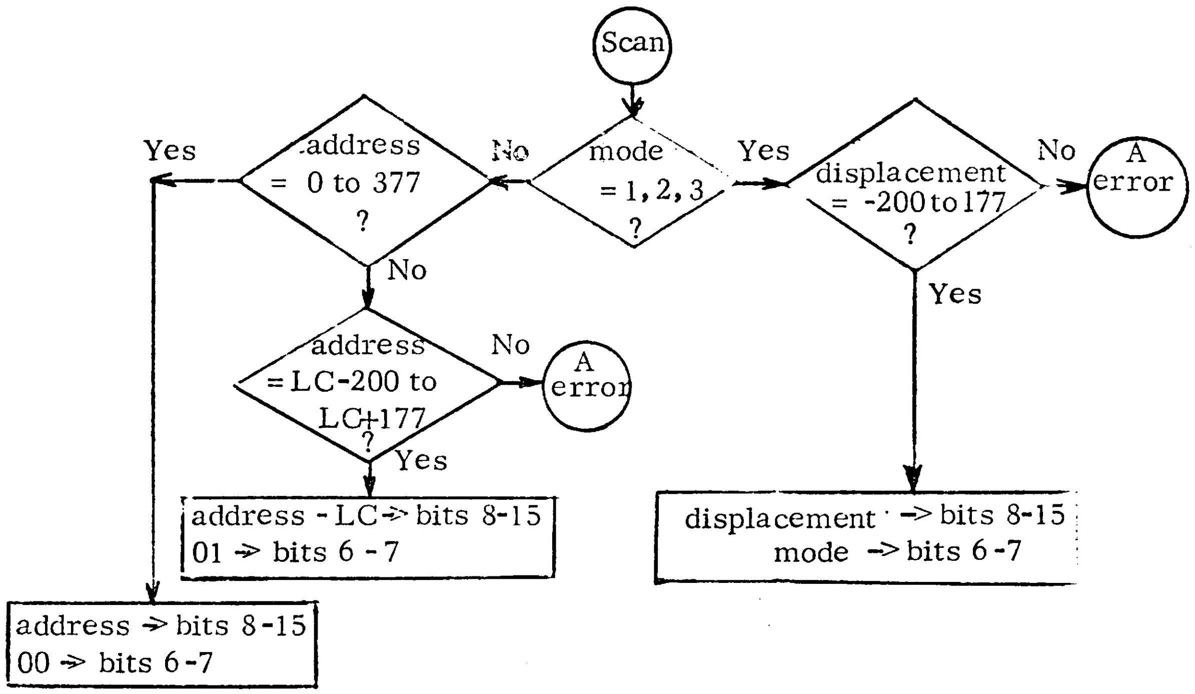
An MR instruction with accumulator is represented in memory as:

					indirect bit										
0	mr-mnemonic		ac		↓	mode		displacement							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Some examples of MR instructions with an accumulator are:

```
STA  3, SAVE
LDA  1,0,3
LDA  3, COUNT@
```

Formation of an Effective Address



Instructions Requiring an Accumulator

Certain commonly used I/O instructions have been defined with the device-code field, CPU. The accumulator field is filled by the programmer. The mnemonic fields and their equivalent I/O instruction counterparts are:

Mnemonic	Equivalent
READS	DIA accumulator, CPU
INTA	DIB accumulator, CPU
MSKO	DOB accumulator, CPU

The format of the source program instruction is:

```
iac-mnemonic △ accumulator
```

where:

iac-mnemonic is one of the three semi-permanent symbols listed above.

accumulator is any legal expression evaluating to an accumulator 0, 1, 2, or 3.

Some examples of instructions requiring an accumulator are:

```
READS 0
MSKO  3
```

Instructions with No Argument Fields

Certain. commonly used I/O instructions have been defined as semi-permanent symbols using the pseudo-op .DUSR. .DUSR defines a simple symbol that does not take any argument fields. The mnemonics and their equivalent I/O instructions are:

Mnemonic	Equivalent
IORST	DICC 0, CPU
HALT	DOC 0, CPU
INTEN	NIOS CPU
INTDS	NIOC CPU

CHAPTER 4

PERMANENT SYMBOLS

Pseudo-ops are permanent symbols which direct the assembly process. The Absolute Assembler pseudo-ops can be grouped into five logical categories, according to the functions they perform. The categories are:

1. Location counter pseudo-ops
2. Symbol table pseudo-ops
3. Terminating pseudo-ops
4. Number radix pseudo-op
5. Text pseudo-ops

Each pseudo-op is explained within this chapter according to the above categories, in the order specified. Appendix C gives a list of all pseudo-ops in alphabetical order, along with their function and basic syntax. In general, pseudo-op lines may appear anywhere within a source program.

In addition to the pseudo-ops, the permanent symbol . ,which has as its value the current contents of the location counter, is described.

LOCATION COUNTER PSEUDO-OPS

Pseudo-op: . LOC

Syntax: . LOC △ expression

Purpose: The . LOC pseudo-op sets the location counter equal to the value of expression. If this pseudo-op does not appear in the symbolic program, the program will be assembled starting at location 0.

Errors: If the expression cannot be evaluated in pass 1 or its value exceeds 32,767 (decimal), the assembler will flag it for a location error (L) and will ignore the statement.

Examples:

```
.LOC 400
```

The next line in the source program will be assembled at location 400.

```
TAB:  .LOC  .+24
```

A location statement can be used to reserve a block of storage. In the example, . LOC allocates a block of twenty locations for a table wherein the first location in the table is labeled TAB.

Pseudo-op: .BLK

Syntax: .BLK △ expression

Purpose: The .BLK pseudo-op is used explicitly to allocate a block of storage. The assembler will increment the location counter by an amount equal to the value of expression. It is important to note that the block of storage reserved is not initialized to zero.

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

Examples:

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

The source coding above will reserve a block of six words starting at location BLK6.

```
TABL1:  .BLK   12
or
TABL1:  .BLK   2*5
```

will each reserve a block of ten locations where the first location is TABL1.

LOCATION COUNTER PERMANENT SYMBOL

Symbol: .

Purpose: The symbol . has the value of the current contents of the location counter and may be used as an operand of an expression.

Example:

```
                                :
00126    003000    C3000:     3000
         000130               . LOC  .+1
                                :
00140    000137               JMP    .-1
```

SYMBOL TABLE PSEUDO-OPS

Symbol table pseudo-ops: .DALC, .DIAC, .DIO, .DIOA, .DMRA, .DMR, and .DUSR have the format:

pseudo-op△symbol = expression

where:

pseudo-op	is one of the pseudo-ops listed above.
symbol	is a programmer-chosen symbol.
expression	is any legal expression which will be assigned as the value of symbol.

Besides having a value, a symbol defined with one of the symbol table pseudo-ops, other than .DUSR, also implies a certain type of instruction mnemonic. Thus, once defined, the symbol must be used with fields appropriate to the defined type of instruction mnemonic. For example, the pseudo-op .DALC defines a symbol that is an implied arithmetic and logical instruction mnemonic and which requires either two or three fields following the symbol, giving the source accumulator, the destination accumulator, and an optional skip field in that order.

For example:

103120	.DALC	MULT4 = 103120	MULT4 is defined as: $\boxed{1\,0\,0\,0\,0\,1\,1\,0\,0\,1\,0\,1\,0\,0\,0\,0}$
127120	MULT4	1,1	MULT4 must be used with at least two fields that evaluate within the limits of the ALC instruction accumulator fields (2 bits each). $\boxed{1}\underbrace{\boxed{0\,1}}_{\text{source-ac}}\underbrace{\boxed{0\,1}}_{\text{destination-ac}}\boxed{1\,1\,0\,0\,1\,0\,1\,0}\underbrace{\boxed{0\,0\,0}}_{\text{skip}}$

SYMBOL TABLE PSEUDO-OPS (Continued)

If the field to be ORed into the instruction cannot be accommodated in the bit positions, an overflow (O) error will occur. The field, though will be unaltered.

```
        103120  .DALC    MULT4 = 103120
O00000  107120  MULT4    4,1
```

If the field is to be ORed into non-zero bit positions, the field must evaluate to zero. Otherwise, an overflow (O) error will result.

```
        123120  .DALC    MULT4 = 123120      ;BIT FIELD 1-2 NOT ZEROED
 O00001 127120   MULT4   1,1                 ;OVERFLOW ON FIELD 1
  00002 127120  MULT4    0,1                 ;O.K. SINCE FIELD 1 = 0
```

If the field following a semi-permanent symbol does not fit the implied format, a formatting (F) error will result.

```
        103120  .DALC    MULT4 = 103120      ;TWO, OPTIONALLY THREE
                                             ;FIELDS REQUIRED FOR
                                             ;MULT4
 F00000 123120  MULT4    1                   ;TOO FEW EXPRESSIONS
  00001 127121  MULT4    1,1,1               ;O.K.
 F00002 127121  MULT4    1,1,1,1             ;TOO MANY EXPRESSIONS
```

The symbol table pseudo-op .DUSR defines a symbol as having a value. No fields are implied. The symbol table pseudo-op .XPNG is used to clear pre-defined semi-permanent symbols so that the programmer can define his own semi-permanent symbols.

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DALC

Syntax: .DALC △ symbol = expression

Purpose: The .DALC pseudo-op defines symbol as a semi-permanent symbol having the value of expression. Use of a .DALC-defined symbol implies formatting of an ALC instruction, requiring at least two fields, and optionally, three. The format in which the semi-permanent symbol is used is:

symbol △ source-ac/destination-ac[△skip]

These fields are assembled as shown below.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
| |sac |dac|                |  skip  |
```

Errors: If less than two or more than three fields are included within the instruction, the instruction will be flagged for a format (F) error. Field overflow is flagged with an O error.

Example:

```
       103400    .DALC        AND = 103400
00020  107400    AND          0,1
00021  107402    AND          0,1,SZC
```

Notes: The special atom # may be specified anywhere within the line to specify the setting of the no load bit (bit 12) within the ALC instruction.

Shift bits 8 and 9 and Carry bits 10 and 11 can be set by appending the following letters to a three-character user symbol during the .DALC definition:

$$.\mathrm{DALC}\ \Delta\ \underline{\text{symbol}} \begin{Bmatrix} \mathrm{Z} \\ \mathrm{O} \\ \mathrm{C} \end{Bmatrix} \begin{Bmatrix} \mathrm{L} \\ \mathrm{R} \\ \mathrm{S} \end{Bmatrix} = \underline{\text{expression}}$$

Notes: (Continued)

These letters will cause bits 8 and 9 and bits 10 and 11 to be set as follows:

Letter	Bits 8 and 9	Bits 10 and 11
L	01	
R	10	
S	11	
Z		01
O		10
C		11

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DIAC

Syntax: .DIAC △ symbol = expression

Purpose: The .DIAC pseudo-op defines symbol as a semi-permanent symbol having the value of expression. Use of a .DIAC-defined symbol implies the formatting of an instruction which requires an accumulator. The format in which the semi-permanent symbol is used is:

symbol △ accumulator

One field is required, the field being assembled as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			ac												

Errors: If one field is not included in the instruction, the instruction will be flagged for a format (F) error. If the field requires more than 2 bits, it will be flagged with a field overflow (O) error.

Example:

```
         000430    .DIAC   RPT = 000430
                             :
00150    010430    RPT       2
```

Pseudo-op: .DIO

Syntax: .DIO △ symbol = expression

Purpose: The .DIO pseudo-op defines symbol as a semi-permanent symbol having the value of expression. Use of the .DIO-defined symbol implies the formatting of an input/output instruction not requiring an accumulator. The format in which the semi-permanent symbol is used is:

symbol △ device-code

One field is required for the instruction, being assembled as follows:

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
|                            |           device-code |
```

Errors: If no fields are included, or if more than one are included within the instruction, the instruction will be flagged for a format (F) error. If device-code requires more than 6 bits the instruction is flagged with a field overflow (O) error.

Example:

```
          063400   .DIO      SKPDN = 063400
                               :
00017     063413   SKPDN     PTP
```

Notes: The Busy and Done bits 8 and 9 can be set by appending one of the following letters to a three-character symbol (e.g., NIO) during the .DIO definition:

$$.DIO \triangle \underline{symbol} \begin{Bmatrix} S \\ C \\ P \end{Bmatrix} = \underline{expression}$$

These letters appended to the three-character user symbol cause bits 8 and 9 to be set as follows:

S	01	Busy set, done not set	See page 3 - 9 for values
C	10	Busy not set, done not set.	
P	11	Busy set, done set.	

Pseudo-op: .DIOA

Syntax: .DIOA △ symbol = expression

Purpose: The .DIOA pseudo-op defines symbol as a semi-permanent symbol having the value of expression. Use of a .DIOA-defined symbol implies the formatting of an I/O instruction with two required fields. The format in which the semi-permanent symbol is used is:

symbol △ accumulator △ device-code

The fields are assembled as follows:

```
 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
|        | ac  |               |       device-code      |
```

Example:

```
          060500   .DIOA     DIA = 060500
                                :
00110     060545   DIA       0,45
```

Notes: The Busy and Done bits, bits 8 and 9, can be set by appending one of the following letters to a three-character user symbol during the .DIOA definition.

$$\text{.DIOA} \ \triangle \ \underline{\text{symbol}} \left\{ \begin{matrix} S \\ C \\ P. \end{matrix} \right\} = \underline{\text{expression}}$$

These letters appended to the three-character symbol cause bits 8 and 9 to be set as follows:

S	01	Busy set, done not set.	See page 3 - 9 for values
C	10	Busy not set, done not set.	
P	11	Busy set, done set.	

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DMR

Syntax: .DMR △ symbol = expression

Purpose: The .DMR pseudo-op defines symbol to be a semi-permanent symbol having the value of expression. Use of the .DMR-defined symbol implies formatting of a memory reference instruction with either one or two fields. The format in which the semi-permanent symbol is used is:

symbol △ displacement [△ mode]

The fields are assembled as shown below:

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
|                 | mode |        displacement        |
```

Errors: If less than one or more than two fields are included within the instruction, the instruction will be flagged for a format error (F). Field overflow is flagged with an O error.

```
          000000   .DMR    JMP = 000000
                              :
00205     001400   JMP       0, 3
```

Notes: The special atom @ may be used within the instruction line. If present within the instruction line, the assembler will place a 1 in bit position 5 signifying indirect addressing.

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DMRA

Syntax: .DMRA △ symbol = expression

Purpose: The .DMRA pseudo-op defines symbol to be a semi-permanent symbol having the value of expression. Use of the .DMRA-defined symbol implies the formatting of a memory reference instruction with either two or three fields. The format in which the semi-permanent symbol is used is:

symbol △ accumulator △ displacement [△ mode]

The fields are assembled as shown below:

0 1 2	3 4	5	6 7	8 9 10 11 12 13 14 15
	ac		mode	displacement

Errors: If less than two or more than three fields are included within the instruction, the instruction will be flagged for a format (F) error. Field overflow is flagged with an O error.

Example:

```
        020000    .DMRA    LDA = 20000
                     :
00011   023400    LDA       0,@0,3
```

Notes: The special atom @ may be specified within the instruction line. If it is present, the assembler will assemble a 1 in bit position 5, indicating indirect addressing.

Pseudo-op:	.DUSR
Syntax:	.DUSR △ symbol = expression
Purpose:	The .DUSR pseudo-op defines symbol to be a semi-permanent symbol having the value of expression. Unlike other semi-permanent symbols, a symbol defined by .DUSR is merely given a value and has no implied formatting. It may be used within any expression.

Example:

```
000025    .DUSR     B = 25

000250    .DUSR     C = B*10

000223    C - B
```

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op:	.XPNG
Syntax:	.XPNG
Purpose:	The .XPNG pseudo-op will delete from the assembler symbol table all symbol definitions except those for permanent symbols. After expunging the semi-permanent symbols, the programmer is free to define any instruction mnemonics that he desires.

Example:

```
        .XPNG

020000 .DMRA    LDA = 20000

040000 .DMRA    STA = 40000

        .END
```

TAPE TERMINATING PSEUDO-OPS

Pseudo-op: .END

Syntax: .END [△ expression] ⤸

Purpose: The .END pseudo-op must be the final source line in a source program written in one of the following forms:

.END ⤸

or

.END △ expression ⤸

The line in which the .END pseudo-op appears must be terminated with a carriage return. If the .END pseudo-op is followed by an expression, its value is taken as the starting address of the program just assembled. After reading in the object tape, the loader will automatically start the execution of the program at the location specified. If there is no expression, the loader will halt after loading the object program.

Pseudo-op:	.EOT
Syntax:	.EOT
Purpose:	The .EOT pseudo-op is used when it is necessary to continue a program onto another source tape. When the assembler encounters the .EOT pseudo-op within the source input, the assembler will stop the source input device and halt with 000006 in the address lights. The assembly can then be continued by loading a new tape and pressing the console CONTINUE switch.

NUMBER RADIX PSEUDO-OP

Pseudo-op: .RDX

Syntax: .RDX △ expression

Purpose: The .RDX pseudo-op will change the current input radix. At the beginning of each pass the assembler will start by interpreting integers as octal. The source program can change this radix by giving a pseudo-op line 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.

Errors: If the expression cannot be evaluated in pass 1, or its value is less than two or greater than 10, the assembler flags the line with a radix (D) error and continues to use the previous radix.

Example:

```
Location    Value     Statement

            000002    .RDX  2
00000       000037    101!11011     ;5 ORed with 33₈ (27₁₀)
            000003    .RDX 3
00001       000013    21 + 11       ;7 + 4
00002       000006    12*12/11      ;5*5/4
            000012    .RDX 10
00003       000115    77
00004       000077    63
00005       000037    9*8/3+7
```

Note that the output is always given in octal.

TEXT PSEUDO-OPS

The text pseudo-ops are used to store ASCII octal codes for a character string. The text pseudo-ops have the form:

<u>pseudo-op</u> Δ/<u>text-string</u>/

where: <u>pseudo-op</u> is one of the following: .TXT, .TXTE, .TXTF, or .TXTO.

> / stands for any character that is used to delimit the string. / may be any character other than carriage return, space, tab, comma, null, line feed, form feed, rubout, or any character that the programmer uses within the text string. / delimits the string but is not part of the string.

Upon encountering a text pseudo-op, the assembler takes the next significant character (other than carriage return) as the text delimiter and assigns succeeding pairs of characters to consecutive memory locations until encountering the next delimiter. Every two characters in the text string generate a single storage word. If the string contains an odd number of characters, the final one is paired with a null character. If the string contains an even number of characters, a null word is assigned the location immediately following the string. This provides a convenient method for an output routine to detect the end of a text string.

Storage of ASCII octal codes requires seven bits of an eight-bit byte. The leftmost bit is used to indicate parity, depending upon the pseudo-op used:

.TXT	sets the leftmost bit of the byte to 0 unconditionally.
.TXTE	sets the leftmost bit of the byte for even parity on the byte.
.TXTF	sets the leftmost bit of the byte to 1 unconditionally.
.TXTO	sets the leftmost bit of the byte for odd parity on the byte.

Carriage return and form feed can be used to continue the text string from line to line or page to page. These characters are not stored as part of the text string.

If the programmer wishes to introduce a carriage return, space, tab, comma, null, line feed, form feed, rubout, or delimiting character as part of the text string, he may enclose the ASCII octal equivalent of the character (or the special integer form of the character) within angle brackets. Angle brackets may also be used to enclose an angle bracket. For example:

```
<177><"<> <012> <"/>
```

By default, the assembler will pack text bytes from right to left; the programmer can change the packing mode using the pseudo-op .TXTM.

TEXT PSEUDO-OPS (Continued)

Pseudo-ops:	.TXT .TXTE .TXTF .TXTO
Syntax:	.TXT △/text string/ .TXTE △/text string/ .TXTF △/text string/ .TXTO △/text string/
Purpose:	The text pseudo-ops will store the ASCII codes for text string, packed two per word with parity set in accordance with the text pseudo-op used.

Examples:

```
        .TXT /INCLUDE <177> AND <"<>/

                text string

  This source line would be represented as:

                047111
                046103
                042125
                020105
                020177
                047101
                020104
                000074

        .TXTE /INCLUDE <177> AND <"<>/

  This source line will be represented as:

                047311
                146303
                042125
                120305
                120377
                047101
                120104
                000074
```

TEXT PSEUDO-OPS (Continued)

Examples:

```
.TXTF /INCLUDE <177> AND <"<>/

This source line will be represented as:

    147311
    146303
    142325
    120305
    120377
    147301
    120304
    000274

.TXTO /INCLUDE <177> AND <"<>/

This source line will be represented as:

    147111
    046103
    142325
    020105
    140577
    142316
    136040
    000000
```

TEXT PSEUDO-OPS (Continued)

Pseudo-op: .TXTM

Syntax: .TXTM △ expression

Purpose: The .TXTM pseudo-op is used to specify whether packing of text strings will be right to left or left to right. If expression evaluates to zero, packing will be right justified, and if expression evaluates to non-zero, packing will be left justified. By default, packing is from right to left.

Examples:

```
    .TXTM  0
    .TXT /A/          ; generates storage word 000101
    .TXTM 1
    .TXT /A/          ; generates storage word 040400
```

APPENDIX A

OPERATING PROCEDURES

The assembler (091-000002) is loaded with the binary loader (091-000004). Once loaded, the assembler requests information on I/O devices and assembly mode. If the programmer wishes at any time to restart the assembler, it can be restarted at location 000002 and new I/O device assignments made. Restarting at location 000003 will initiate only a new MODE request, with I/O assignments remaining the same. The assembler queries and appropriate programmer responses are given below:

IN:

The user responds with a single digit indicating the input device as follows:

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

LIST:

The user responds with a single digit indicating the listing device as follows:

1	Teletypewriter Model 33
2	Teletypewriter Model 35
3	Line printer
4	Paper tape punch (for ASR 33)
5	Paper tape punch (for ASR 35)

BIN:

The user responds with a single digit indicating the output device as follows:

1	Teletypewriter punch
2	Paper tape punch

MODE:

The user responds with a single digit indicating the assembly mode as follows:

1	Pass 1
2	Pass 2 - Output an object tape.
3	Pass 2 - Output a listing.
4	Pass 2 - Output an object tape and listing.
5	Output a symbol table.

A response of 4 to MODE is illegal if the programmer selected the same device for both the object tape (BIN) and listing (LIST).

Restarting the Assembler

To restart the assembler, follow the steps below:

Press RESET

Set 000002 in the data switches

Press START

The assembler will start at the IN query, allowing the programmer to reassign devices.

Reassigning the Mode

To reassign assembly mode but keep device assignment the same, follow the steps below:

Press RESET

Set 000003 in the data switches

Press START

The assembler will reissue the MODE query.

Saving the Symbol Table

If the programmer defines new instruction mnemonics using the symbol table pseudo-ops, he may wish to save the symbol table. To do so, the programmer must punch a new object tape of the assembler after pass 1 as follows:

1. Complete pass 1 of assembly.
2. When the assembler finishes pass 1, it types MODE. Respond by typing a 1. This will cause the assembler to eliminate non-initial entries from the symbol table. It will then stop since there is no source tape in the reader.
3. Using the Binary Punch Program, punch the tape from location 000002 to the location specified by the contents of location 000004, which addresses the last location in the symbol table.
4. Specify a starting address of 000002 to the Binary Punch Program as the the location to be punched in the start block.

APPENDIX B

CHARACTER SET

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	Rubout	177
3	063	H	110		

APPENDIX C

PSEUDO-OPS

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

APPENDIX D

ERRORS

Extensive examination of statement syntax takes place during both passes of assembly to detect input errors. A statement in error will be flagged with from one to three letters indicating the class into which the error or errors fall. Statements in error on pass 1 will be printed with their error flag(s) to the teletypewriter. The user can then decide whether to continue to pass 2 or correct existing errors. Statements in error on pass two will be printed with their error flag(s) to the teletypewriter. If a listing of the program was requested to a device, the listing of the program will contain error flags at the appropriate statements.

Examples of the syntax errors are given in the listing on page 1-3. The meaning of the error codes is described below.

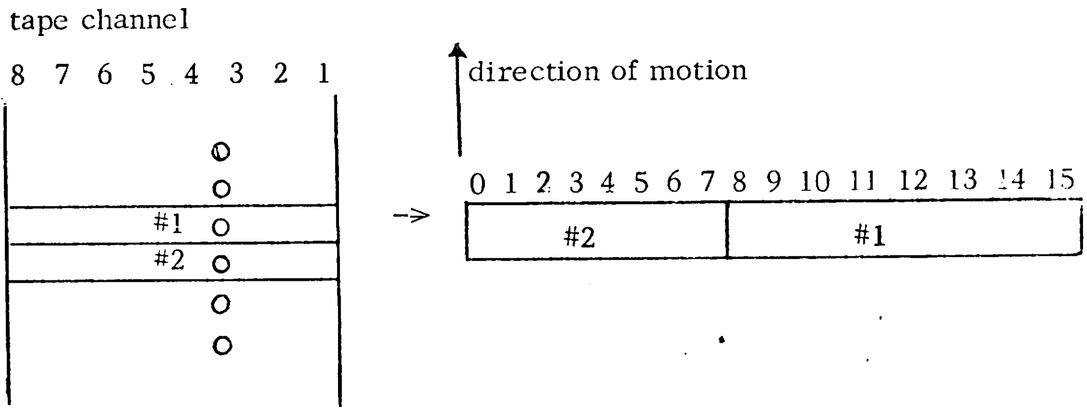
Flag		Type of Error
A	Address error:	An address is outside the addressing range in an MR instruction.
B	Bad character:	An illegal character occurs in some symbol.
C	Colon error:	An illegal character (not legal in a label) occurs before a colon.
D	Radix error:	An attempt is made to define a radix outside the range 2-10 or to use a digit that is not in the range of the current radix.
E	Equivalence error:	An undefined symbol appears on the righthand side of an equivalence line.
F	Format error:	An instruction format is in error, such as too many or too few arguments.
I	Input error:	Parity was checked on input and a character in error was found (shown as \ ).
L	Location error:	The expression in a . LOC or . BLK is outside the range of the LC, such as an attempt to move the LC to a location less than 0.
M	Multiply defined:	A previously defined symbol is redefined, such as an attempt to use the same label symbol twice.

Flag		Type of Error
N	Number error:	A number contains a non-digit or is outside the integer range.
O	Field overflow:	An argument field is outside the range, such as an accumulator greater than 3 or a skip field greater than 7.
P	Phase error:	An unexpected difference is detected between pass one and pass two in the source program scan, for example a symbol having a different value on pass two.
Q	Questionable line:	An unexpected format occurs in some line.
S	Symbol table overflow:	Memory capacity is exceeded.
T	Symbol table pseudo-op:	The format of a symbol table pseudo-op line is illegal.
U	Undefined symbol:	The scan encounters an undefined symbol on the righthand side of an equivalence line on pass one or encounters a symbol on pass two that was not defined on pass one.
X	Text error:	There is an error in a text string or in the format of one of the text pseudo-ops.

APPENDIX E

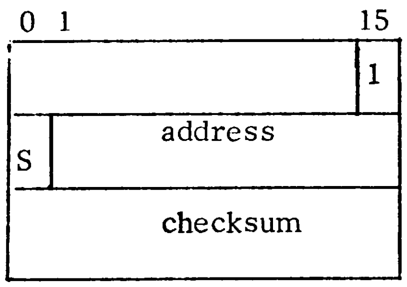
OBJECT TAPE FORMAT

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:



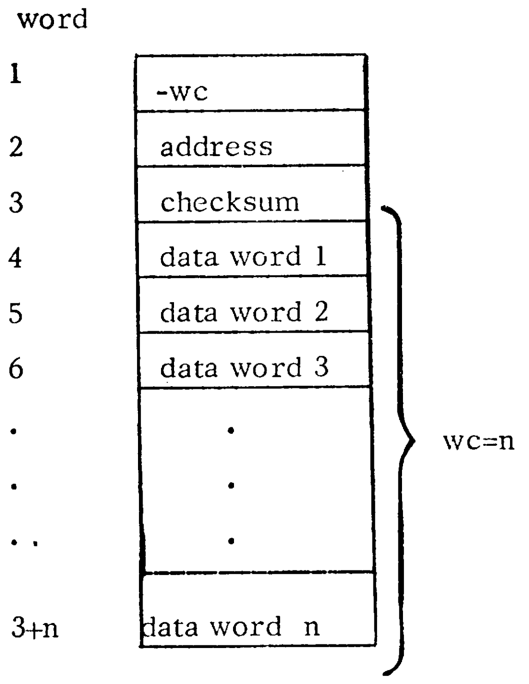
In other words, the first tape character forms bits 8 through 15 of the data word (Channel 8 to bit 8, etc.) and the second tape character forms bits 0-7 of the data word (Channel 8 to bit 0, etc.). The first non-null tape character signifies the start of a block. The block type is determined by the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

<u>Start Block - First Word is 000001.</u>



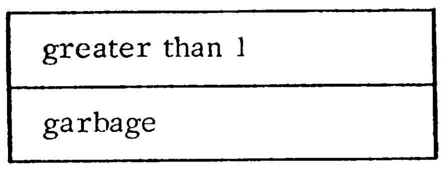
The first word contains 1. The second word uses the sign bit as a flag. If S is equal to 0, the loader will transfer to the address in bits 1 through 15 of the word. If S is equal to 1, the loader will halt. The third word checksum is the same as that for a data block.

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



The twos complement (see page 2-1, How to Use the Nova Computers) 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.

Error Block - First Word is greater than 1.



The first word is greater than positive 1. An error block is ignored in its entirety by the loader. All error blocks are terminated by a rubout.

APPENDIX F

RADIX 50 REPRESENTATION

Radix 50 representation is used to condense symbols of five characters into two words of storage using only 27 bits. Each symbol consists of from 1 to 5 characters and a symbol having five characters may be represented as:

$$a_4 a_3 a_2 a_1 a_0$$

where: each a_i may be one of the following characters:

A - Z	(26 characters)
0 - 9	(10 characters)
.	(1 character)

All symbols are padded, if necessary, with nulls. Each character can be translated into octal representation as follows:

Character a_i	Translation b_i
null	0
0 to 9	1 to 12_8
A to Z	13 to 44_8
.	45_8

If a_i is translated to b_i, the bits required to represent the symbol can be compared as follows:

$$N_1 = ((b_4 * 50) + b_3) * 50 + b_2$$

$N_{1\ \text{maximum}} = (50)^3 - 1 = 174777$, which can be represented in 16 bits (one word)

$$N_2 = (b_1 * 50) + b_0$$

$N_{2\ \text{maximum}} = (50)^2 - 1 = 3077$, which can be represented in 11 bits.

Thus the symbol can be represented in 27 bits of storage.

APPENDIX G

INSTRUCTION SET

ADC	102000
ADCC	102060
ADCCL	102160
ADCCR	102260
ADCCS	102360
ADCL	102100
ADCO	102040
ADCOL	102140
ADCOR	102240
ADCOS	102340
ADCR	102200
ADCS	102300
ADCZ	102020
ADCZL	102120
ADCZR	102220
ADCZS	102320
ADD	103000
ADDC	103060
ADDCL	103160
ADDCR	103260
ADDCS	103360
ADDL	103100
ADDO	103040
ADDOL	103140
ADDOR	103240
ADDOS	103340
ADDR	103200
ADDS	103300
ADDZ	103020
ADDZL	103120
ADDZR	103220
ADDZS	103320
AND	103400
ANDC	103460
ANDCL	103560
ANDCR	103660
ANDCS	103760
ANDL	103500
ANDO	103440
ANDOL	103540
ANDOR	103640
ANDOS	103740
ANDR	103600
ANDS	103700
ANDZ	103420
ANDZL	103520
ANDZR	103620
ANDZS	103720
COM	100000
COMC	100060
COMCL	100160
COMCR	100260
COMCS	100360
COML	100100
COMO	100040
COMOL	100140
COMOR	100240
COMOS	100340
COMR	100200
COMS	100300
COMZ	100020
COMZL	100120
COMZR	100220
COMZS	100320
DIA	060400
DIAC	060600
DIAP	060700
DIAS	060500
DIB	061400
DIBC	061600
DIBP	061700
DIBS	061500
DIC	062400
DICC	062600
DICP	062700
DICS	062500
DIV	073101
DOA	061000
DOAC	061200
DOAP	061300
DOAS	061100
DOB	062000
DOBC	062200
DOBP	062300

DOBS	062100
DOC	063000
DOCC	063200
DOCP	063300
DOCS	063100
DSZ	014000
HALT	063077
INC	101400
INCC	101460
INCCL	101560
INCCR	101660
INCCS	101760
INCL	101500
INCO	101440
INCOL	101540
INCOR	101640
INCOS	101740
INCR	101600
INCS	101700
INCZ	101420
INCZL	101520
INCZR	101620
INCZS	101720
INTA	061477
INTDS	060277
INTEN	060177
IORST	062677
ISZ	010000
JMP	000000
JSR	004000
LDA	020000
MOV	101000
MOVC	101060
MOVCL	101160
MOVCR	101260
MOVCS	101360
MOVL	101100
MOVO	101040
MOVOL	101140
MOVOR	101240
MOVOS	101340
MOVR	101200
MOVS	101300
MOVZ	101020
MOVZL	101120
MOVZR	101220
MOVZS	101320
MSKO	062077
MUL	073301
NEG	100400
NEGC	100460
NEGCL	100560
NEGCR	100660
NEGCS	100760
NEGL	100500
NEGO	100440
NEGOL	100540
NEGOR	100640
NEGOS	100740
NEGR	100600
NEGS	100700
NEGZ	100420
NEGZL	100520
NEGZR	100620
NEGZS	100720
NIO	060000
NIOC	060200
NIOP	060300
NIOS	060100
READS	060477
SBN	000007
SEZ	000006
SKP	000001
SKPBN	063400
SKPBZ	063500
SKPDN	063600
SKPDZ	063700
SNC	000003
SNR	000005
STA	040000
SUB	102400
SUBC	102460
SUBCL	102560
SUBCR	102660
SUBCS	102760
SUBL	102500
SUBO	102440
SUBOL	102540
SUBOR	102640
SUBOS	102740
SUBR	102600
SUBS	102700
SUBZ	102420
SUBZL	102520
SUBZR	102620
SUBZS	102720
SZC	000002
SZR	000004
@	002000
@	100000
#	000010

INDEX

Where there are a large number of page references for a given topic, the primary page reference will be indicated by an asterisk (*) following the reference.

ABSOLUTE ASSEMBLER ERROR CODES

Up to three error codes may be output per source line. The error codes are output in the first three character positions of the listing line. The first error encountered causes a code to be placed in column 3, the second in column 2, and the third in column 1.

Code	Error
A	Addressing error
B	Bad Character
C	Colon error
D	Radix error
E	Equivalence error
F	Formatting error
I	Parity error on input
L	Location counter error
M	Multiply defined symbol error
N	Number error
O	Overflow field
P	Phase error
Q	Questionable line error
U	Undefined symbol error
X	Text input error

cut along dotted line

DATA GENERAL CORPORATION
PROGRAMMING DOCUMENTATION
REMARKS FORM

DOCUMENT TITLE ______________________________

DOCUMENT NUMBER (lower righthand corner of title page) ____________

TAPE NUMBER (if applicable) ______________________________

Specific Comments. List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

General Comments and Suggestions for Improvement of the Publication.

FROM:	Name: ____________________	Date: ____________	
	Title: ____________________		
	Company: ____________________		
	Address: ____________________		
	____________________		

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed In The United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Programming Documentation

FOLD UP SECOND FOLD UP

STAPLE