CYBER 180 II ASSEMBLER

for

CPU

EXTERNAL REFERENCE SPECIFICATION

(S5233)

```
+--------------------------------------+
|  This  product  is  intended  for    |
|  used  only  as described in this    |
|  document.  Control  Data  cannot    |
|  be  responsible  for  the proper    |
|  functioning    of    undescribed    |
|  features and parameters.            |
+--------------------------------------+
```

REVISION DEFINITION SHEET

REV	DATE	DESCRIPTION
A	05/30/80	Original, for CPU Assembler only.
B	09/15/80	Revised for comments against REV. A.
C	05/05/81	Revised to add IOU mnemonic instructions, several appendicies, and other corrections.
D	08/14/81	Revised to correct comments against REV. C.
E	12/01/81	Revised to correct grammatical errors, delete obsolete pseudo-op GEN from examples, correct errors in descriptions of the IOU instructions, update titles and update this revision page. Since revision D of this document was never submitted to DCS the revision bars have been generated relative to revision C.
F	04/11/85	Revised to include vector instructions for the Cyber 180-990. Appendix A, which describes the command parameters, changed to include the LIST_OPTIONS parameter.
G	04/18/90	Revised to include the PSFSA instruction for the Cyber 2000 and the vector instructions for the Cyber 2000V. Descriptions of IOU instructions and discussion of differences between CPU and IOU coding removed because IOU assembly not supported.

1.0 SCOPE

This document is the external specification for the CYBER 180 II Assembler. This assembler runs on the CYBER 180 machine in CYBER 180 mode and assembles CYBER 180 CPU code. The object program output of the Assembler is compatible with the NOS/VE loader. The II Assembler is the language successor to the CI Assembler described in the ARH1693 ERS document.

1.1 APPLICABLE DOCUMENTS

The following documents reference related material which would be of value to the reader.

- CYBER 180 Mainframe Model Independent GDS (MIGDS), Rev. AD (ARH 1700).
- CYBER 180 CI CPU Assembler ERS (ARH 1693).
- NOS/VE Command Interface
- NOS/VE Program Interface

2.0 LANGUAGE STRUCTURE

A CYBER 180 Assembly language source program consists of a sequence of statements which contain symbolic machine instructions, pseudo instructions, and comment lines. With the exception of the comment lines, each statement consists of a label field, an operation field, argument field(s), and a comments field. Each field is terminated by one or more blank characters. The size of the argument field is restricted by the maximum statement size only. Statement format is essentially free field, except for the label field which must start in column 1.

A statement consists of one or more physical lines of data. A line may be up to 255 characters long and the Assembler will print the entire line at the rate of 88 characters per print line. Assembler will only examine the first 88 characters of a line. Information after column 88 is presumed to be comments.

The language also supports a procedure mechanism with parameter capability. Each time the name of the procedure is referenced, the body of the procedure will be inserted in the code. This will be further explained in the section entitled 'Procedures'.

2.1 STATEMENT

A statement is an ordered group of fields starting (from left to right) with one Label field followed by Operation and Argument fields and one Comments field. The number of fields allowed in a statement is not limited. The comments field is optional, but the other fields must be accounted for by field delimiters. A statement may be continued onto more than one line, but no more than one statement is allowed per line.

2.1.1 FIELD

A Field is a consecutive group of characters starting with a non-blank character and terminated by a blank character , end-of-line, or character position 88 of the line, whichever occurs first.

The only exceptions to this definition are:

a) Blanks may appear freely in a CHARACTER STRING without causing field termination.

b) Blanks may appear freely in the COMMENTS field.

c) If a continuation character ";" is encountered within a field which is not a COMMENTS or CHARACTER STRING, the field is continued on the next line.

d) Extra or spurious fields in a statement are not detected and no error is diagnosed.

2.1.2 SUBFIELD

A Subfield is a consecutive group of characters starting with a non-blank character and terminated by a comma "," or by End-Of-Field, whichever occurs first. A field may have one or more subfields.

The only exceptions to this definition are:

a) Commas may appear freely in a CHARACTER STRING without causing subfield termination.

b) Commas may appear freely in the COMMENTS field.

c) If a continuation character ";" is encountered within a subfield, the subfield is continued on the next line.

d) Extra or spurious subfields in a field are not detected and no error is diagnosed.

2.1.3 NULL FIELD

The absence of a field or subfield is automatically detected by the Assembler based on the number of fields. An OPERATION field must not be Null and must have as many ARGUMENT fields following it as required by its defining pseudo instruction or PROCEDURE, although the number of ARGUMENT fields can be variable and depend on some other field.

The rules for NULL field:

a) A blank in character position 1 of a line indicates the absence of the LABEL field on that line. The next non-blank character on the line, excluding comments, is accepted as part of an OPERATION field.

b) An OPERATION field cannot be blank.

c) Two consecutive commas indicate the presence of a null subfield.

d) One comma "," followed by a blank indicates (as specified) end-of-subfield and end-of-field and can be used to delimit trailing Null subfields. The configuration blank,blank indicates a Null field with two Null subfields.

2.2 COMMENTS

Comments may start in any column, but are always the last field on a line, and end at end of line. All comments must begin with a period. Scanning by the Assembler stops when a period preceded by a blank or a period in column 1 is encountered, thus comments may contain any Ascii character, including characters that would otherwise have special meaning (e.g. the semicolon which denotes continuation when used outside of comments).

When a statement is continued to the next line, comments may appear after the continuation character on the line being continued.

2.2.1 STATEMENT CONTINUATION

Normally, column 88 terminates a source statement that has not otherwise terminated. However, a statement that cannot be contained in the first 88 characters can be continued on successive lines by placing a semi-colon ";" at the continuation point. A statement may only be broken between fields, subfields, or terms of an expression. A term may not be broken onto 2 lines (e.g. a long character string must fit on one line). The statement will be continued at the first non-blank character on the next line at or after character position 2. Character position 1 of all continuation lines must contain a blank. The continuation character, if used, must appear at or prior to character position 88.

The only exceptions to this definition are:

a) Semicolons may appear freely in a CHARACTER STRING without causing continuation. This implies that character strings cannot be continued across statements.

b) Semicolons may appear freely in a COMMENTS field without causing continuation. Comments cannot be continued across statements.

2.3 CHARACTER SET

The Assembler recognizes the following, graphic character subset of the NOS/VE ASCII character set as input:

Alphabetic   A through Z (upper or lower case)
              $ @ # _ :

Numeric   0 through 9

Special Characters:

Blank or Space

+   Add
-    Subtract or Unary Minus
*   Multiply
/   Divide or Logical NOT
=   Equal
<   Less Than
>   Greater Than
&   Logical AND
| Logical Inclusive OR (vertical bar)
|| Logical Exclusive OR (double vertical bar)
<=   Less Than or Equal To
>=   Greater Than or Equal To
/=   Not Equal To
.   Period or Decimal Point
,   Comma
(Left Parenthesis
)   Right Parenthesis
[Left Bracket
]   Right Bracket
'   Apostrophe
;   Continuation
**   Shift

In addition to the characters listed above, the Assembler accepts the following characters as part of program comments or as part of a Character String:

" \ ↑ ! ? % { } ¬

The Assembler distinguishes between upper and lower case characters only when used within character strings enclosed by quotes.

Other ASCII characters appearing before the comment field are diagnosed as an error.

2.4 SYMBOL DEFINITION

A symbol is a set of alphabetic or numeric characters that identifies a byte address or a value and its associated attributes. The symbol must start with any alphabetic character, and the symbol can be a maximum of thirty-one (31) characters long, and cannot include any of the special characters. The colon (:) may not be used as a character in a user defined symbol, it is reserved for language defined names. Symbols are defined when they are used in the label field of any statement (CPU or pseudo instruction), except for some pseudo instructions which ignore the label field and other pseudo instructions which use the label field for other purposes.

EXAMPLES:

Legal Symbols	Illegal Symbols	
P	543	First character must be alphabetic.
R3	ABCDEFGHIJKLMNOPQRSTUVWXYZ012345	Exceeds 31 characters
PROGRAM	ABE+15	Contains plus sign

2.4.1 LINKAGE SYMBOLS

Modules (assembly units) can be linked to other modules (assembly/compilation units) through symbols defined as entry points.

Entry points in the current module are declared with a DEF or DEFG pseudo instruction. This allows the entry point to be referenced from another module. External entry points can be referenced by declaring them with the REF pseudo instruction and are treated as relocatable values.

To link to entry points with different names, a symbol can be ALIASed to another symbol.

2.4.2 SYMBOL ATTRIBUTES

In addition to the value or byte address associated with a symbol, each symbol has symbol attributes. Symbol attributes are various pieces of information about the symbol which describe properties of that symbol. Attributes are normally associated with a symbol at the time the symbol is defined. This is an automatic process within the Assembler and takes place whenever symbol definition takes place.

The CYBER 180 Assembler contains six built-in attributes which are associated with a symbol. These attributes and their associated mnemonics are:

Symbol Category	SC:
Address Mode	AM:
Symbol Value	VA:
Length	LB:, LC:, LW:
Starting Bit Position	SB:
Symbol Number	SN:

Each attribute is discussed and defined in the section on Attribute Functions. A symbol's attributes are always referenced using one of the attribute function mnemonics listed above. This reference may not be forward. It is used for retrieval only, and has the form:

```
attribute_function(symbol)
```

The Assembler also permits any symbol to have any number of additional programmer defined attributes. These additional attributes can be given names and values by the programmer and can have any meaning desired. The values may not exceed 64 bits. The names and values can be altered during the course of the program assembly using the ANAME and ATRIB pseudo instructions. The ANAME pseudo instruction is used to assign a name to a particular attribute. Following that, a symbol can then be assigned a value associated with the named attribute. This attribute name may then be used in the following manner to retrieve the value of the attribute:

```
user_defined_attribute_name[symbol]
```

An attribute name for any of the programmer defined attributes will be valid until changed.

2.5 REGISTERS

Register designators symbolically represent the 32 operating registers. The designators are inherent to the Assembler and cannot be changed during assembly. However, other symbols may be equated to them. There is an Assembler defined attribute (#regtyp) which defines the type of register a symbol represents.

Register Type	Designator
Address	'An' or a symbol with its #REGTYP attribute set to "#AREG".
Operand	'Xn' or a symbol with its #REGTYP attribute set to "#XREG".

For the forms An or Xn, n is a single hex digit from 0 to F. Any other value for n, for example H, causes An or Xn to be interpreted as a symbol rather than a register designator.

EXAMPLES:

A1 Designates address register 1

A10 Interpreted as a symbol, not a register

2.6 DATA NOTATION

Data notation provides a means of entering values for calculation, increment counts, operand values, line counts, control counter values, text for printing out messages, characters for forming symbols, etc.

The two types of data notation are character and numeric. The Assembler allows the user to introduce data in the program in two basic ways.

As a self defining term

As a number in numeric data notation

2.6.1 SELF DEFINING TERMS

A Self-Defining Term is a constant whose value is defined by its structure. The value of a Self-Defining Term is constant throughout the program and is not altered by the relative position of the program in storage. The Assembler uses two methods by which a Self-Defining Term can be expressed:

a) As an unsigned string of binary, octal, decimal, or hexadecimal characters, the first character of which must be a decimal digit, which has the following format:

```
numeric_character_string(base)
```

Base is optional, but when present it must be enclosed by parenthesis. Base may only be hexadecimal (16), decimal (10), octal (8), or binary (2). Any other value for base results in an error. The following examples illustrate the numeric notation:

```
ALPHA+0FF(16)       "0FF(16)" is a Self-Defining Term
3*(NET_PAY)         "3" is a Self-Defining Term
```

The range of this form of Self-Defining Term must be consistent with its use in the program.

b) As a Generalized Self-Defining Term which has the following structure

```
symbol'character-string'
```

where the character string is always enclosed by apostrophes and where "symbol" is one of the characters:

Symbol	Type of Generalized Self-Defining Term
C	CHARACTER STRING: Constant translated into 8 bit ASCII code. The characters can be any of the characters in the Assembler character set.* Note that a lower case letter will generate a different 8 bit ASCII code than an upper case character. The maximum string length is limited to one line and therefor cannot exceed 87 characters.

Self-defining terms can assume a range of values (e.g. precision or storage occupied) depending on their type and usage. In all cases however, the internal representation of a self-defining term is an integral number of bytes. When translation from input format to internal representation occurs, self-defining terms are expanded to the next nearest multiple of bytes, provided they do not exceed the maximum defined below.

---

*Two consecutive quote marks in a C character string are used to indicate a single quote within the string.

During the expansion process, justification and filling (where required) also take place as defined:

Type of Self-Defining Term	Minimum Size (Bytes)	Maximum Size (Bytes)	Justification	Filling
Decimal	8	8	Right	Zero
Hexadecimal	1	8	Right	Zero
Octal	1	8	Right	Zero
Binary	1	8	Right	Zero
C	1	as needed	Left	Space

A self-defining term used as a single term expression can assume any of the values described above. When self-defining terms are used as part of a multi-term expression however, the following additional restrictions apply:

a) When an address symbol is used only the byte offset for the address is used. Bit offset, if any, and section ordinal are discarded.

b) The size of all numeric terms (decimal, hexadecimal, octal, binary, or string) will be 8 bytes when arithmetic operations are performed. Strings are right justified and truncated or zero filled as necessary to be 8 bytes and are treated as integer. When an expression contains operators, the result is integer. Arithmetic operations are performed using 2's complement arithmetic. When the expression contains only one term, the result is that term (which is not converted in form).

2.6.2 NUMERIC DATA NOTATION

Numeric data can be specified in binary, octal, hexadecimal, or decimal notation with the INT and DINT pseudo instructions. Only decimal notation is available with the FLOAT and DFLOAT pseudo instructions. The value is converted to an integer or a floating point number in single or double precision. Floating point conversion is performed by a CYBER 180 math library conversion program. The actual representation of the output data is beyond the scope of this document.

2.0 LANGUAGE STRUCTURE
2.6.2 NUMERIC DATA NOTATION

Formats:

```
Data Item          +------+-------+----------+
                   | sign | value | modifier |
                   +------+-------+----------+
```

sign Optional.

+ or omitted	The value is positive.
-	The negative value is formed.

value A series of binary, octal, hex or decimal digits consisting of an integer (required), optional decimal point and optional fraction, or optional base. An integer value (fixed point) does not contain a point, but may contain an optional base indicator enclosed in parenthesis. The fixed point format is thus a numeric, self-defining term with a sign preceding. A floating point value is noted by the occurrence of the point. If point occurs then base may not occur and value is decimal.

An octal value can be a maximum of 22 octal digits and cannot exceed 64 bits of significant data. A decimal value cannot exceed 5.2 x 10**1232 in absolute value. used in a floating point pseudo instruction. Extra significant digits cause a diagnostic. A hex value can be a maximum of 16 digits. If value is omitted, it is assumed to be zero. The actual minimum or maximum values permitted are further limited by the pseudo instruction in which the data notation appears.

modifier Associated with a floating point value is an optional exponent modifier. Exponent defines a power of 10 scale factor.

Format is E, En, E+n, or E-n.

When the sign is plus or omitted, the exponent (n) is positive.

When n is omitted, it is assumed to be 0. The value of n cannot exceed 32767 and is always a decimal integer.

A fixed point value can have 32-bits or 64-bits of precision and a floating point value can be generated in either single precision (one word) or double precision (two words), depending on the pseudo instruction.

The effect of the exponent is to multiply the value by 10 decimal raised to the n power or -n power. Limitations of maximum and minimum values and exponents may be found in the appropriate CYBER 180 math library documents.

	Legal	Illegal	Explanation
Examples:	-21904	316E	missing base
	3.14159	7F(16)E-3	value must be decimal
	1.7E-6	.2893	interpreted as comments.

2.7 EXPRESSIONS

Entries in sub-fields of most source statements are interpreted as expressions consisting of a combinartion of one or more terms. A comma or blank terminates the expression. When symbolic names appear as terms in expressions the Assembler must be able to replace the symbolic name with its associated value. The association of a symbolic name with a value is called symbol definition and is described in Section 2.4. An expression in which all the symbolic names can be evaluated (which means the expression can be reduced to a single value) is said to be an "evaluable expression". An "absolute evaluable expression" is an expression whose symbolic name terms are all defined in statements previous to the current statement.

2.7.1 TERMS

A term represents an evaluation made during the assembly process. A value is assigned to a term either by the Assembler or the term may be self-defining (as in the case of a constant).

A term can be a:

Symbol that is evaluable
(One that Assembler can associate with a value)

Self-defining term

Function reference

Attributes

Register designator

2.7.2 ORDER OF EVALUATION

Expression evaluation normally is determined by the binding strength of the operators involved. This can be altered by the use of parenthesis. Terms inside of parenthesis are evaluated first. Parenthesis can be nested to any depth, and will be evaluated in the order of innermost to outermost. An expression such as INDEX+4 or AD*(9+PAN), is reduced to a single value as follows:

a) The expression takes on the attributes of the first term in the expression from left to right.

b) Each term is given its defined value. When arithmetic operations are performed on a term it's internal representation is converted to integer. When strings are used as arithmetic terms they are truncated, if necessary, or right justified with zero fill, if necessary, to occupy 8 bytes and are treated as an integer.

c) Arithmetic operations are performed from left to right. Operations at the same parenthetical level within the highest binding strength are performed first. For example:

```
VE+VX*AE/AX
```

is evaluated as VE+((VX*AE)/AX).

d) Division always yields a truncated integer result and division by zero yields a zero result with a generated diagnostic.

The operators processed by the Assembler during expression evaluation are:

Operators	Binding Strength	Function
+	7	Plus (unary)
-	7	Minus (unary)
/	7	Logical NOT or Complement (unary)
**	6	Binary Shift (logical)
*	5	Integer Multiply
/	5	Integer Divide
+	4	Integer Add
-	4	Integer Subtract
<	3	Less Than
>	3	Greater Than
<=	3	Less Than or Equal
>=	3	Greater Than or Equal
=	3	Equal
/=	3	Not Equal
&	2	Logical AND
|	1	Logical OR
||	1	Logical Exclusive OR

NOTE: All operators are binary (i.e., require two operands) except the three specifically indicated as unary. These require only one operand.

2.7.3 THE LOGICAL NOT OPERATOR

The logical NOT or complement operator causes a one's complement of its operand, based on a length of 64 bits.

Value	Binary Equivalent	One's Complement

5	000...0101	111...1010
12	000...1100	111...0011

2.7.4 LOGICAL AND, OR, EXCLUSIVE OR

The logical AND, OR, and exclusive OR compare two operands "A" and "B" as follows:

<table>
<tr><th>A</th><th>B</th><th>&</th><th>|</th><th>||</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

2.7.5 THE BINARY SHIFT OPERATOR

The Binary Logical Shift Operator determines the direction of shift based on the sign of the second operand: a negative operand denotes a right shift and a positive operand denotes a left shift. For example: 7**(-2) results in a logical right shift of two bit positions for the operand 7. Shifts are end-off with zero bit replacement.

2.7.6 THE COMPARISON OPERATORS

The result of any comparison produced by the comparison operators is: False = 0; True = 1.

EXAMPLES:

Expression	Value	
9>11	0	(9 is not greater than 11)
/3=4	0	(the word-size value /3 is equal to 11...1100 and is not equal to 4; i.e., 00...0100)
3/=4	1	(3 is not equal to 4)

/(3=4)	11...11	(3 is not equal to 4, so the result of the comparison is 0 which NOTed becomes a word size value of all 1's.)

2.8 ABSOLUTE AND RELOCATABLE TERMS AND EXPRESSIONS

Any term in an expression may be relocatable or absolute (non-relocatable). A relocatable term is one which represents the location of some piece of assembled code (i.e. represents an address in the memory of the computer). Its symbol category would be 6. An example would be the label of a BSS statement.

An absolute expression consists of either an absolute term or a combination of terms that, when evaluated, has no relocation. An absolute term is an absolute symbol or a constant. All operators may be used with absolute terms. Absolute terms are always internally represented in the 2's complement number system (the number -0 does not exist in this system).

A relocatable expression consists of a single relocatable term or a number of terms that, when evaluated, has relocation. A relocatable term results when an absolute term is added to or subtracted from a relocatable term and the result is not negative and does not exceed the storage capacity of a section. All arithmetic operations may be performed on relocatable terms. If a relocatable term cannot result, then the relocatable term is first converted to an absolute term whose value is the byte offset of the relocatable term and the result of the arithmetic operation is an absolute term.

If an absolute value is required of an expression, then it is converted to absolute value. A relocatable value is required only for certain operands of the ADDRESS pseudo instruction. If an expression contains only a single term, the result is that term and the result may be absolute, relocatable, or string.

3.0 PROGRAM STRUCTURE

This chapter describes the general structure of a program. In some cases, it repeats information described elsewhere and correlates it so that the programmer will obtain a better understanding of how the program is assembled, loaded, and executed. Some references are made to the NOS/VE Loader but for a complete description of the loader, refer to the applicable NOS/VE document.

A CYBER 180 program consists of one or more modules that can be assembled separately, either in the same computer run or in independent runs. The Assembler will assemble many modules from the same input file per call. These many program modules can all be written in the Assembler source language, or can be written in any other source language available in the product set of the operating system as long as the compiler or Assembler produces relocatable binary output in a form acceptable to the NOS/VE loader. An Assembly language module is composed of statements beginning with an IDENT pseudo instruction and ending with an END pseudo instruction.

The Assembler repertoire includes pseudo instructions that facilitate relocatable module linkage. Through these linkages, modules loaded together can transfer control to each other and can access common storage locations.

The first topic considered in this chapter is the program module and how the Assembler and the programmer organize the object code into program sections. Following this is a brief description of the counters that control the sections.

3.1 PROGRAM SECTIONS

A CYBER 180 Assembly program is a collection of statements which are translated via an assembly process, into a CYBER 180 object module. Object modules resulting from separate assemblies, or compilations by a CYBER 180 Compiler (CYBIL, FORTRAN, etc.) can be combined, via a linking process, into a single object module, and may undergo further transformation into a form capable of direct execution by the CYBER 180 hardware.

A set of statements between an IDENT pseudo instruction and an END pseudo instruction is a program module. A CPU program module can be divided into sections having different attributes. For instance, the CODE section has the attributes of READ and

EXECUTE, while the WORKING section is READ and WRITE. The use of sections provides a means of code protection. As assembly of a program module proceeds, the Assembler or the user designates that object code be generated or that storage be reserved in specific sections. By properly assigning code sequences, data, or reserved storage areas in blocks through use of ORG or USE, a programmer can intermix instructions and data for the different sections. The Assembler assigns locations in a section consecutively as it encounters instructions destined for the section. A symbol defined within a section is not local to the section. That is, it is global and can be referred to from any other section in the program.

For the CPU there are several types of sections available. Only a CPU module may contain SECTION or USE statements. If a CPU module does not contain a USE instruction or if object code is generated (or storage reserved) before the first USE instruction, the Assembler places the object code in the CODE section, which is one of the five default sections. The user controls use of the default-sections and any user-established sections, through USE, ORG, and SECTION pseudo instructions.

3.1.1 DEFAULT SECTIONS

The following is a list of default sections and their attributes established for the user by the Assembler:

CPU SECTIONS:

CODE	READ+EXECUTE
WORKING	READ+WRITE
BINDING*	BIND+READ
STACK*	READ+WRITE

IOU SECTIONS:

CODE	READ+WRITE+EXECUTE

* Symbols may be associated with addresses in these sections, but data may not be initialized at assembly time except for the BINDING section in which pointers may be established through the use of the ADDRESS pseudo instruction.

3.1.2 THE BINDING SECTION

The BINDING section is a special purpose section whose function is to permit access to data and code that is either internal or external to the current module. This is accomplished via pointers in the BINDING section which are built by the NOS/VE loader. In addition, the NOS/VE Library_Generator may "bind" modules together. Part of this "binding" process consists of consolidating the separate BINDING section of each module into one common BINDING section by eliminating redundant entries (pointers) in the BINDING section. This means that "binding" inherently requires that entries in the BINDING section be "order independent". The user must beware to preserve this "order independence".

It is recommended that reference to the pointers in the BINDING section be limited to the "load" type instructions (See Section 7.3.1) or the CALLSEG instruction. For these instructions the Assembler inherently generates "relocation" object text which permits the Library_Generator to adjust the displacement field of these instructions to a new value as a result of module "binding".

The use of other CPU instructions (e.g. ADDRQ) or generation of data which contains a displacement relative to the BINDING section is permitted and the Assembler will generate the necessary "relocation" object text with the assumption that the field (displacement) being generated is an unsigned positive field. If this assumption is not correct, the relocation attributes may be specified by the intrinsic Relocation_function (R:) (See Section 5.1.9). If the relocation attributes cannot be specified by the relocation function (R:), then the module cannot be bound and if the module is to be assembled without diagnostics the module must be declared "NONBINDABLE" via the MACHINE statement (See Section 4.2.1).

3.2 SECTION CONTROL COUNTER

Each section has a section counter from which the byte offset from the beginning of the section, and the bit offset in the current byte can be obtained. The Assembler automatically updates and maintains this counter when a section is first established, or its use is resumed. The current contents of the location counter may be returned as a relocatable value via the location counter function $ (dollar sign).

The byte offset is the relative location of the next byte to be assembled or reserved in the section. It is possible to increment the byte offset simply by using either ORG or BSS pseudo instructions. ORG also permits the programmer to reset the counter to some lower location in the section. The current byte offset can be referenced by using the function $(0).

The bit offset points to the next bit to be used in the current byte, and can range in value from 0 to 7. It can be referenced by using the function $(1).

3.2.1 FORCING PARCEL ALIGNMENT

A parcel is the minimum instruction size. A parcel is 2 bytes or 16 bits. The CYBER 180 hardware requires that all instructions start on a parcel boundary. This also means that the byte address of the instruction must be even. In a CYBER 180 Virtual Machine assembly, if any of the following conditions are true, the Assembler forces parcel alignment.

- Insufficient room remains in a partially filled parcel for the next instruction to be generated.

- The current statement is an END, IDENT, or ALIGN 0,2 pseudo instruction.

4.0 PSEUDO INSTRUCTIONS

Pseudo instructions are instructions needed by the programmer to write programs, but for which there are no hardware equivalents.

Pseudo instructions discussed in this chapter are classified according to application as follows:

Module identification (IDENT and END)

Binary control (MACHINE)

Symbol assignment (EQU, SET, ANAME, ATRIB)

Module linkage (DEF, DEFG, REF, ALIAS and ADDRESS)

Data generation (BSSZ, INT, DINT, FLOAT, DFLOAT, PDEC, CMD, VFD and TRUNC)

Assembly control (DO, ELSE, DEND, WHILE, and SKIPTO)

Error control (ERROR, FLAG)

Listing control (LIST, PAGE, SPACE, TITLE, XRSY)

Section control (SECTION, USE, ORG, POS, BSS, ALIGN)

Procedure/function pseudo instructions (PROC, PEND, PNAME, FNAME LOCAL, OPEN, CLOSE, CONT)

In general, pseudo instructions can be placed anywhere in a module. The following list of pseudo instructions is valid only for a CPU module.

ADDRESS	ALIAS	DEF	DEFG	DFLOAT	DINT
FLOAT	INFOMSG	PDEC	REF	SECTION	USE

4.1 MODULE IDENTIFICATION

Module identification pseudo instructions designate the beginning and end (IDENT-END) of a module).

4.1.1 IDENT - MODULE IDENTIFICATION

An IDENT pseudo instruction of the following form is the first statement of a module recognized by the Assembler. The first input statement must be an IDENT or comment statement and if end of information does not follow an END statement then the statement following END must be another IDENT or comment statement. Assembler flags any spurious use of IDENT before END as an error. For a CPU module the argument field must be blank.

label	operation	argument
name	IDENT	

name — Name of the module, it is required and can be 1-31 characters of which the first must be alphabetic as defined in Section 2.3. This name cannot be redefined, and may be used to reference the code section.

Example:

```
TEST    IDENT                .  TEST is the name of the module
```

4.1.2 END - END MODULE

An END pseudo instruction must be the last statement of each module. It causes the Assembler to terminate all counters, conditional assembly, procedure generation and code duplication.

The Assembler combines all local blocks (sections) into a relocatable subprogram block, generates the relocatable binary tables and produces the listing.

label	operation	argument
label	END	tralabel

label — Optional, last address of the module.

tralabel — Optional, a 1-31 character symbol specifying the entry point to which control transfers for a CPU module. This symbol must be declared as an entry point in the (linked) CPU module, either by a DEF, DEFG, or REF pseudo instruction in this module. At least one module must specify a transfer address or the loader signals an error. If more than one module indicates a transfer address, then the loader uses the first one encountered.

Example:

```
        END   START   .START is the transfer label
```

4.2 BINARY CONTROL

This section describes a pseudo instruction that allows the user to control the binary output produced by the Assembler.

4.2.1 MACHINE - DECLARE OBJECT PROCESSOR TYPE

The MACHINE pseudo instruction specifies the type of computer processor on which the object program can be executed. A MACHINE statement must appear before any generated code. The MACHINE pseudo instruction also identifies which instruction mnemonics are permitted (CPU or IOU) and which type of object text to generate (CPU or IOU). No more than one MACHINE pseudo instruction may appear within any assembly unit (IDENT-END).

label	operation	argument
	MACHINE	type,bind

type C180CPU The object processor is a CYBER 180 CPU (default). The Assembler will accept CPU instruction mnemonics and will generate CPU object text.

C180IOU The object processor is a CYBER 180 IOU. The Assembler will accept IOU instruction mnemonics and will generate IOU object text. Negative numbers in the generated data will be in 1's complement form (since the IOU is a 1's complement processor). This value is accepted, but not supported at this time.

No other type is available at this time.

bind This subfield is applicable only if type is C180CPU.

BINDABLE (DEFAULT) The Assembler will generate additional object text to permit the Library_generator to "bind" the module. If the other statements in the module do not conform to the rules for "bindable" code then a FATAL diagnostic will be issued for each of these statements (See Section 5.1.9).

NONBINDABLE The object text generated will have the "non-bindable" attribute set. No diagnostics will occur if the rules for "bindable" code are not followed. The Library_generator will abort if an attempt is made to "bind" this module.

Example:

```
MACHINE    C180CPU   .Binary is for a CYBER 180 CPU
```

4.3 SYMBOL ASSIGNMENT

The pseudo instructions SET and EQU permit direct assignment of values to symbols. The values can be absolute or relocatable. Subsequent use of the symbol in an expression produces the same result as if the value had been used as a constant. Symbols defined using EQU cannot be redefined.

Any symbol may be given one or more programmer defined attributes by using the ANAME pseudo instruction to define an attribute name, and then using the ATRIB pseudo instruction which assigns a specific value to a specific symbol. Once defined, the attribute function may be used to recover the attribute value assigned to the argument.

4.3.1 SET/EQU - ASSIGNMENT OF VALUES

A SET or EQU pseudo instruction defines the symbol in the label field as having the value and attributes indicated by the expressions in the argument field. The difference between SET and EQU is that symbols defined with an EQU cannot be redefined, whereas symbols defined with a SET may be redefined with a subsequent SET any number of times.

label	operation	argument
label	SET	list
label	EQU	list

label — (Required) A list of one or more symbols, or symbol element number identifiers to which the argument field list is assigned. It will have a symbol category of 9.

list — Evaluatable expressions. The expressions cannot include symbols as yet undefined. The maximum value of a list element cannot exceed 64 bits (0FFFFFFFFFFFFFFFF(16)). When the first element in the list is a symbol, the attributes of that symbol will replace the attributes of the symbol in the label field.

Any symbol in the label field cannot be referred to prior to its first definition.

The SET and EQU pseudo instructions assign a list of values to the symbol(s) in the label field. The list must contain only evaluable expressions at the time the pseudo instruction is processed by the Assembler. The label field may consist of list names (symbols) or list element identifiers.

List elements are referenced using the form:

listname[element number]

where listname is the name of the list, and element number is an evaluable expression denoting a particular element in the list, where, for an n element list, element number = 0, 1, 2,...,n-1. A negative element number is diagnosed as an error.

A SET or EQU pseudo instruction within a PROCEDURE is processed by the Assembler only when the PROCEDURE is referenced and not when the PROCEDURE is defined. The expressions which comprise the list elements must be evaluable therefore, only when the PROCEDURE is referenced.

A particular list element may have a value of ZERO or NULL depending on how that element is defined. A null element is assigned to a list whenever a position for a list element is indicated with appropriate commas, but the position is devoid of contents. A null list element has the numeric value zero when used computationally. Null elements may be transferred from one list to another.

The argument field is completely processed and for each subfield in the argument list the value is assigned to the corresponding value element of each of the symbolic names in the label field. If a list is specified, it is replaced completely by the argument. If a list element is specified, replacement is on an element by element basis. The designated element is replaced by the first argument list value, and succeeding elements being replaced by the corresponding argument value.

Example #1

```
A    SET    3,5,7,12,15
```

When this pseudo instruction is processed by the Assembler, the label "A" is associated with the list 3,5,7,12,15. The elements and their values are:

```
A[0] =  3
A[1] =  5
A[2] =  7
A[3] = 12
A[4] = 15
A[5] =  0 .(Null)
```

Following the previous pseudo instruction, we could then give the pseudo instructions:

```
A[1]    SET    42
A[4]    SET    17
A[5]    SET     8
```

And the list associated with "A" would then be:

```
A[0] = 3
A[1] = 42
A[2] = 7
A[3] = 12
A[4] = 17
A[5] = 8
A[6] = 0 .(Null)
```

Example #2

```
X,Y    SET    SUM+3,12,SUM+7,6
```

In this case, the symbol SUM must have been previously defined. If its value were 50, then the Assembler would establish two lists X and Y which would both be associated with the list:

```
53,12,57,6
```

In addition, any previous list associated with either X or Y would be erased. The following instructions may then be given:

```
Z       SET    X
X[0]    SET    SUM+1
ZZ      SET    X
X       SET    5,3,1
```

After these pseudo instructions have been executed, the lists appear as:

```
X = 5,3,1
Y = 53,12,57,6
Z = 53,12,57,6
ZZ = 51,12,57,6
```

<u>Example #3</u>

```
BIND_REG    EQU    A3    .points to the binding segment
TEMP_REG    SET    A5    .temporary working register
```

BIND_REG now is equal to 3 and has the attributes of #AREG. The symbol BIND_REG cannot be redefined. TEMP_REG is equal to A5 and has the attributes of #AREG. TEMP_REG can be changed with a subsequent SET.

<u>Example #4</u>

```
A      SET  0,1,2,3,4
A[2]        SET  5,6
```

results in the list:

```
A = 0,1,5,6,4
```

The pseudo instruction:

```
A[1]    SET ,,10,,11
```

modifies the list to:

```
A = 0,,10,,11
```

4.3.2 ANAME DIRECTIVE

The ANAME pseudo instruction is used to define a programmer defined attribute name and to assign a particular attribute number to that name. A particular attribute number may have several names associated with it by using ANAME more than once.

label	operation	argument
label	ANAME	value

label — A previously undefined symbol.

value — Evaluatable expression whose value can be any positive integer.

4.3.3 ATRIB DIRECTIVE

The purpose of the ATRIB pseudo instruction is to assign a value to the programmer defined attribute of a particular symbol. The symbol to which the attribute value is assigned is the symbol in the LABEL field. If the symbol in the LABEL field of this pseudo instruction is not previously defined, it will be placed in the permanent symbol table and given a symbol category of 1, and the specified attribute assigned to it. If the symbol in the LABEL field has been previously defined, the value is assigned to the attribute of the symbol and replaces any previous value assigned to that symbol for that attribute. Normally, a symbol must be defined before attribute values are assigned to that symbol. An exception occurs when PROCEDURES are executed while a source statement is being processed.

label	operation	argument
label	ATRIB	attribute,value

label — A label field symbol is required.

attribute — A previously defined (using the ANAME pseudo instruction) programmer defined attribute name.

value — Evaluatable expression.

4.3.4 USE OF THE ANAME AND ATRIB PSEUDO INSTRUCTIONS

```
.  CONSIDER THE FOLLOWING SEQUENCE OF DIRECTIVES:

INDEX   ANAME 1
BASE    ANAME 2

.  At this point we have defined two programmer defined
.  attributes INDEX and BASE.  Any symbol can now have values
.  assigned to these attributes.

SMB1    ATRIB INDEX,5
SMB1    ATRIB BASE,0A(16)

.  At this point, the INDEX attribute of SMB1 is 5
.  and the BASE attribute of SMB1 is a hexadecimal A.

SMB1    ATRIB INDEX,0
SMB1    ATRIB BASE,2

.  At this point the INDEX and BASE attributes of SMB1 have been
.  reassigned to the values:

.          INDEX[SMB1]  = 0
.          BASE[SMB1]   = 2

.   Attributes may be used as terms of an expression.

JA      SET   BASE[SMB1]
JB      EQU   INDEX[SMB1]
```

4.4 MODULE LINKAGE

The pseudo instructions DEF, DEFG, and REF are valid only in CPU modules, and are used to denote entry points, either in the current module or a separately assembled/compiled module. A symbol flagged as an entry point denotes an address representing data or code, which can be referenced by other modules. It is through the use of entry points that the NOS/VE loader is able to link modules together. See the appropriate NOS/VE loader document for complete details.

4.4.1 DEF,DEFG-DECLARE ENTRY SYMBOLS

The DEF and DEFG pseudo instructions define symbols as entry points in the current CPU module. DEFG pseudo instruction defines symbols as gated entry points. (Gated entry points are explained further in the NOS/VE loader documentation.)

label	operation	argument
	DEF	sym1,sym2,...,symn
	DEFG	sym1,sym2,...,symn

symi (Required) Linkage symbol from 1-31 characters of which the first must be alphabetic as defined in section 2.4. (Also see ALIAS statement.) Each symbol must be further defined in the module as a relocatable address (catagory 6). The symbol may not be a LOCAL or OPENED symbol. The appearance of the same symbol more than once in a DEF or DEFG is not an error, but the symbol may not appear in both a DEF and DEFG statement.

Example:

```
DEF   PRG1   .PRG1 is a symbol in this compilation unit.
```

4.4.2 REF-DECLARE EXTERNAL SYMBOLS

The REF pseudo instruction lists symbols that are defined as entry points in independently compiled or assembled CPU modules for which references can appear in the module being assembled.

label	operation	argument
	REF	sym1,sym2,...,symn

symi (Required) Linkage symbol, 1-31 characters of which the first must be alphabetic as defined in Section 2.4. These symbols must not be further defined within the module being assembled. Note that it is still possible to have new definitions for the symbol by using LOCAL or OPEN statements. (Also see ALIAS statement.)

Symbols may be declared in a REF statement prior to or subsequent to their use in the program. They must be global symbols, and cannot have been declared OPEN or LOCAL. Symbols which are declared in a REF pseudo instruction are assumed to be relocatable and their use in expressions must follow the rules for relocatability. Any further definition of a REF symbol will be diagnosed as an error.

Example:

```
        REF   TAGX    .TAGX IS AN ENTRY POINT IN A DIFFERENT
                      .ASSEMBLY/COMPILATION UNIT.
```

4.4.3 ALIAS - EQUATE LINKAGE SYMBOLS

The ALIAS pseudo instruction gives the programmer the ability to declare entry points with names other that that used within the current CPU module.

label	operation	argument
name1	ALIAS	name2

name1 — 1-31 character linkage symbol used by the Assembler. This symbol must be further defined in the module as a DEF, DEFG, or REF symbol.

name2 — 1-31 character CYBER 180 linkage symbol. This symbol is not restricted by the limits of symbol definition in Section 2.4. The symbol must consist of alphabetic or numeric characters, the first of which must be alphabetic. The colon may not be used as one of the characters.

Example:

```
TAG   ALIAS   TAGFORALONGNAME    .TAG FOR A LONG NAME IS
                                 .DEFINED IN A DIFFERENT
                                 .COMPILATION UNIT.
```

4.4.4 ADDRESS - FORM CYBER 180 ADDRESS

The ADDRESS pseudo instruction enables the generation of references to full Process Virtual Address (PVA's) in a CPU module, to be filled in by the NOS/VE Loader. Generally, this pseudo instruction is used in the BINDING section to form pointers.

label	operation	argument
label	ADDRESS	typ1,sym1,...,typn,symn

label — Optional, symbol assigned the value of the beginning of the address list. Symbol category equals 6.

typi — Type designating the address insertion type. It can have only the following values else an error is diagnosed:

- P - (Pointer) Creates a pointer (PVA) to the specified address. The generated object code is one word long and is word aligned relative to the section origin. The PVA is stored in the generated object code right justified with zero fill.

- C - (Code Base Pointer) Used for linking procedures. The format for the PVA is one word of generated object code for internal symbols, and two words of generated object code for external symbols. The generated object code is always word aligned relative to the section origin with the PVA being right justified with zero fill.

- CI- (Code Base Pointer Internal Format) Generates object code for a code base pointer in internal format (1 word) for the symbol, without regard as to whether the symbol is internal or external. The generated object code is word aligned relative to the section origin with the PVA being right justified with zero fill.

- CE- (Code Base Pointer External Format) Generates object code for a code base pointer in external format (2 words) for the symbol, without regard as to whether the symbol is internal or external. The generated object code is word aligned relative to the section origin with the PVA being right justified and zero filled.

- R - (Relative) Generates object code for a PVA which points to a symbol with an offset. The length of the generated object code is 8 bytes in the binding section, or 6 bytes in any other section. The generated object code is word aligned relative to the section origin when in the binding section with the PVA being right justified with zero fill. When not in the binding section, the generated object code is byte aligned.

symi — Following each TYPI subfield there must be a single, corresponding SYMI subfield which contains a symbol or expression which identifies the internal or external location for which a PVA is to be created. Expressions are permitted only when TYPI is R.

Example:

```
      USE    BINDING
      REF    TESTDATA
TAG   ADDRESS   C,TESTDATA   .GENERATES A 2 WORD PVA FOR TESTDATA
      USE #LASTSEC           .WHICH IS IN A DIFFERENT MODULE.
```

4.5 DATA GENERATION

The instructions described in this section are the only pseudo instructions that generate data. All other program data is generated through symbolic machine instructions.

4.5.1 BSSZ-RESERVE ZEROED STORAGE

The BSSZ pseudo instruction generates zeroed bytes of data in the section of a CPU module currently in use.

label	operation	argument
label	BSSZ	aexp

label — Optional, label defined as the byte offset in the section after the appropriate alignment occurs. The symbol identifies the beginning of the reserved storage area.

aexp — Absolute evaluable expression specifying the number of zeroed bytes of storage to be reserved. The expression cannot contain external symbols or result in a relocatable or negative value.

A BSSZ 0 or an erroneous expression causes a force to a byte boundary and the symbol definition, but no storage is reserved. If storage is to be reserved in a CPU module starting at a word, halfword, or parcel boundary, then the BSSZ must be preceded by one of the appropriate alignment pseudo instructions.

Example:

```
     ALIGN  0,8   .FORCE BYTE OFFSET TO A WORD BOUNDARY.
TAG  BSSZ   10    .RESERVES 10 BYTES OF ZEROES.
```

4.5.2 INT - GENERATE INTEGERS

The INT pseudo instruction generates one or more 32-bit integers on a byte boundary in the current section of a CPU module for each item listed in the argument field.

label	operation	argument
label	INT	item1,item2,...,itemn

label — Optional, symbol is assigned the byte offset in the section after the force to the appropriate boundary occurs. Symbol category equals 6.

itemi — Numeric data item. Value of the numeric data item cannot exceed the storage capacity of the item being generated.

Example:

```
TAG   INT   1,2,3
```

4.5.3 DINT - GENERATE 64-BIT INTEGERS

The DINT pseudo instruction generates one 64-bit integer on a byte boundary in the current section of a CPU module for each item in the argument field.

label	operation	argument
label	DINT	item1,item2,...,itemn

label — Optional, symbol assigned the byte offset in the section after the force to a byte boundary occurs. Symbol category equals 6.

itemi   Numeric data item.

Example:

```
TAG   DINT   1,2,3
```

4.5.4 FLOAT - GENERATE SINGLE PRECISION FLOATING-POINT NUMBERS

The FLOAT pseudo instruction generates one 64-bit floating point number on a byte boundary in the current section of a CPU module for each item listed in the argument field. Note that floating point numbers entered with a decimal point must have a digit preceding the period (else the remainder of the statement will be interpreted as comments).

label	operation	argument
label	FLOAT	item1,item2,...,itemn

label   Optional symbol assigned the byte offset in the section after the force to a byte boundary occurs. Symbol category equals 6.

itemi   Numeric data item. Value of numeric data item cannot exceed the storage capacity of a single precision (64-bit) floating point item. Conversion of the numeric data item into the internal floating point representation is performed by a CYBER 180 math libray program. Consult the appropriate CYBER 180 math libray documentation for further information.

Example:

```
TAG   FLOAT   1.347E-6,0,-6.3416E12,1.
```

4.5.5 DFLOAT - GENERATE DOUBLE PRECISION FLOATING-POINT NUMBERS

The DFLOAT pseudo instruction generates one double precision, 128-bit floating point number on a byte boundary in the current section of a CPU module for each item listed in the argument field. Note that floating point numbers entered with a decimal point must have a digit preceding the period (else the remainder of the statement is interpreted as comments).

label	operation	argument
label	DFLOAT	item1,item2,...itemn

label — Optional symbol assigned the byte offset in the section after the force to a byte boundary occurs. Symbol category equals 6.

itemi — Numeric data item. The value of the numeric data item must be within the limits of the storage capacity of the item being generated. Conversion of the item into internal floating point representation is performed by a CYBER 180 math library program. Consult the appropriate CYBER 180 math library documentation for further information.

Example:

```
TAG   DFLOAT   -22.661,6.87701E-14,1E3,0.00000001762
```

4.5.6 PDEC - GENERATE PACKED DECIMAL DATA

The PDEC pseudo instruction generates packed decimal data on a byte boundary for the length of the field desired.

label	operation	argument
label	PDEC	C'string'

label — Optional symbol assigned the byte offset in the section after the force to a byte boundary occurs. Symbol category equals 6.

string — Signed or unsigned numeric decimal character string is required. Any other argument type is diagnosed as an error. Each character in the string generates a 4-bit code. Only the characters 0-9 and + or - are permitted. Any other characters in the string are diagnosed as an error. The sign character (+ or -) must be the last (rightmost) character. If the data is to be used by a BDP instruction the user must insure that the contents of the generated object code fit the requirements of the BDP type designator (See Section 7.4).

Example:

```
TAG   PDEC   C'1234'
```

4.5.7 CMD - GENERATE BIT STRING

The CMD pseudo instruction is a single statement form of PROCEDURE. The output of the CMD pseudo instruction is a string of binary bits together with appropriate control information for the CYBER 180 LOADER. The length of the binary bit string is controlled by the "length list" and the contents of the binary bit string are controlled by the "value list". Both the "length list" and the "value list" can contain multiple subfields, provided that the total bit string produced is greater than zero and less than or equal to 1024 bits.

label	operation	argument
label	CMD,l_lst	v_lst

label — A label field symbol is required. It is used to define the OPERATION field name by which this particular CMD definition will be referenced in subsequent statements of the program. The CMD statement must appear prior to any reference to the operation it defines and may not appear within a PROCEDURE definition. The (optional) label appearing on a line referencing a CMD defined operation will be associated with the generated bit string. Symbol category equals 6.

l_lst — The length list is a list of evaluable expressions whose value represents the length in bits, of each argument field element to be generated by the Assembler. This list is ordered from left to right. If the value of the "l_lst" causes an overflow of the section counter, then an error will be diagnosed.

v_lst — In one-to-one correspondence with the "length list" is a "value list", which is a list of expressions which determines the value assigned to the corresponding element of the "length list". If number of elements in "l_lst" does not match the number of elements in "v_lst" then an error is diagnosed. If the value of a "v_lst" element exceeds the storage capacity allocated by the corresponding "l_lst" element, then an error may or may not be diagnosed depending on the use of the TRUNC statement (See Section 4.5.9).

Example: (Also see the section on PROCEDURES)

```
LA   CMD,8,4,4,16    84(16),F:(2,1),F:(2,0),F:(2,2)
```

4.5.8 VFD - VARIABLE FIELD DEFINITION

The VFD pseudo instruction generates a string of binary bits. The (optional) label is associated with the data string.

The difference between the CMD and VFD pseudo instructions is that the CMD pseudo instruction is a template which does not generate output until called, whereas the VFD pseudo instruction generates output when it is encountered.

label	operation	argument
label	VFD,l_lst	v_lst

label — Optional symbol assigned the byte offset in the section.

l_lst — A list of evaluable expressions which represent the length in bits of each subfield to be constructed. This list is ordered from left to right. If length list causes an overflow of the section counter then an error will be diagnosed.

v_lst — In one-to-one correspondence with the length list is a list of expressions which determine the value assigned to the elements of the length list. If the number of elements of "l_lst" does not match the number of elements of "v_lst" then an error is diagnosed. If the value of the "v_lst" element exceeds the storage capacity specified by the corresponding "l_lst" element, then an error may or may not be diagnosed depending on the use of the TRUNC statement (See Section 4.5.9).

Example:

```
LIST1   VFD,8,16,8,3*8    1,4F(16),6,C'ABC'
```

4.5.9 TRUNC - TRUNCATE

The TRUNC pseudo instruction is used to indicate what action is to be taken, if it is necessary to truncate a value in order to enable it to fit into a field specified by a CMD or VFD pseudo instruction.

label	operation	argument
	TRUNC	value

value — Value is one of the numbers 0 and 1 which have the following meaning:

- 0: Truncate and do not associate an error flag with the data generated.
- 1: Truncate and flag the word generated as in error.

An attempt will always be made to fit the significant bits of a value into a field. When type 1 truncation is specified, the elimination of an unbroken string of non-significant zeros or elimination of an unbroken string of 1's in the case of a negative number, is not considered to be an error. When character data is truncated, trailing blanks are not considered an error.

More than one TRUNC pseudo instruction may appear in a program. The most recently encountered TRUNC pseudo instruction will be used. If no TRUNC pseudo instruction appears in a program, "type 0" truncation will be used.

Example:

```
        TRUNC    1    .FLAG TRUNCATION ERRORS.
```

4.5.10 INFOMSG

The INFOMSG pseudo instruction is used to control the generation of the Informative Diagnostic issued when data generation occurs in the BINDING or STACK sections of a CPU module.*

label	operation	argument
	INFOMSG	value

value - LISTON - Turns generation of error message on (default).

- blank - Suppresses generation of error message.

Example:

```
        INFOMSG   LISTON   .FLAG DATA GENERATION ERRORS.
```

* Data cannot be initialized in the Binding and Stack sections at assembly time, with the exception of the ADDRESS pseudo instruction which can be used in the Binding section.

4.6 ASSEMBLY CONTROL

4.6.1 DO/ELSE/DEND PSEUDO INSTRUCTIONS

This group of pseudo instructions is used for conditional iterative control of Assembler processing. The format of these pseudo instructions is:

label	operation	argument
label	DO	expression
	ELSE	
label	DEND	

label Optional label that is assigned the value of the expression when used on the DO statement. It is not valid on the ELSE pseudo instruction. When specified on a DEND, a cycle effect can be created by using a SKIPTO LABEL instruction. The label of a DEND statement is never entered in the Assembler's symbol table and the presence of a label field is used only as the object of a SKIPTO.

expression Expression must be absolute and evaluable. This expression represents the number of times the DO loop will be executed. If no expression is present, the argument of the DO will be treated as 0. A boolean condition can be specified for conditional assembly of code.

A DEND pseudo operation must be associated with each DO pseudo operation written. However, the ELSE need not be present, but if desired, must occur between the DO and DEND.

The DO pseudo operation operates as follows:

a) An internal counter is set up and initially given the value of 0.

b) If a label is present on the DO line, its value is set to 0.

c) The expression on the DO line is evaluated. Denote the results of this calculation by n. (If no expression was present or the expression was not evaluable, n = 0).

d) If n ≤ 0, skip succeeding lines until an ELSE or DEND pseudo operation is encountered.

   1) If an ELSE pseudo operation is encountered, assemble succeeding statements until a DEND line is encountered. Continue assembly at the statement after the DEND line.

   2) If a DEND pseudo operation is encountered, resume assembly at the statement following the DEND line.

e) If n > 0, the following action occurs:

   1) Increment the internal counter by 1.

   2) If a label was present on the DO line, set the value of the label equal to the new value of the internal counter.

   3) Assemble all lines until an ELSE or DEND pseudo operation is encountered.

   4) Compare the internal counter to n.

      a) If the count is less than n, repeat the procedure from step (e). This causes the count to be incremented, and resumes assembly of the statements following the DO.

      b) If the count is equal to n, terminate control of the DO pseudo operation and resume assembly at the line immediately following the DEND, skipping all statements between the ELSE and DEND if necessary.

Example:

```
 .  EXAMPLE 1)  The following code will assemble one 64-bit word
 .              with a value of X factorial.  If X is negative or
 .              zero, then a word with value zero is assembled
 .              instead:

 FACT      SET    1
 I         DO     X
 FACT      SET    FACT*I         PROCESSED X TIMES IF X>0
           ELSE
 FACT      SET    0              PROCESSED ONCE IF X≤0
           DEND
           VFD,64  FACT

 .  EXAMPLE 2)  The following code will assemble N+1 64-bit words
 .              whose values are 0,...,N where N can be either
 .              positive or negative.  The inner DO block is
 .              processed only if N<0.

           VFD,64  0
 I         DO      N
           VFD,64  I           PROCESSED N TIMES IF N>0
           ELSE
 J         DO      -N
           VFD,64  -J          PROCESSED -N TIMES IF N<0
           DEND
           DEND

 .  If N=3 the above code is equivalent to:
           VFD,16  0
           VFD,16  1
           VFD,16  2
           VFD,16  3

 .  If N=-2 the example code is equivalent to:
           VFD,16  0
           VFD,16  -1
           VFD,16  -2
```

4.6.2 WHILE/ELSE/DEND PSEUDO INSTRUCTIONS

The format of these pseudo instructions are:

label	operation	argument
label	WHILE	expression
	ELSE	
label	DEND	

Label and expression have the same meaning as in the DO pseudo operation. However, there is no limit placed on the value of the expression.

The execution of the WHILE loop is similar to that of the DO, except that the expression is evaluated for each iteration in the loop.

The WHILE pseudo operation is performed as follows:

a) An internal counter is set up and initially is given the value 0.

b) If a label is present on the WHILE line, its value is set to 0.

c) The expression of the WHILE line is evaluated. Denote the results of this evaluation by m. (If no expression is present, or the expression is not evaluable, m = 0.)

d) If m ≤ 0 and this is the first time through the WHILE loop, suppress assembly until an ELSE or DEND pseudo operation is encountered.

   1) If an ELSE pseudo operation is encountered, assemble succeeding statements until a DEND line is encountered. Continue assembly at the statement following the DEND line.

   2) If a DEND pseudo operation is encountered, resume assembly at the line following the DEND line.

   If m ≤ 0 and this is not the first time through the WHILE loop, skip all lines until a DEND pseudo operation is encountered and resume assembly at the line following the DEND.

e) If m > 0,

   1) Increment the internal counter by 1.

   2) Set the value of the label on the WHILE line (if present)

to the new value of the counter.

3) Continue assembly until an ELSE or DEND pseudo operation is encountered, and then repeat the procedure from step c.

Note that the only logical way to get out of a WHILE loop is to change within the loop, one or more of the items which make up the expression on the WHILE line so that the expression will have a value ≤0.

Example:

```
.  This code will assemble a number of 16-bit words whose value
.  are from the Fibonacci series.  Starting with the value 1,
.  each word is equal in value to the sum of the previous two
.  words.  In this example the series is terminated when all of
.  its members less than N have been generated.

         OPEN     A,B,TEMP
A        SET      0
B        SET      1
         WHILE    B<N
         VFD,16   B
TEMP     SET      B
B        SET      A+B
A        SET      TEMP
         DEND
         CLOSE    A,B,TEMP

.  If N=10 the above code is equivalent to:
         VFD,16   1
         VFD,16   1
         VFD,16   2
         VFD,16   3
         VFD,16   5
         VFD,16   8
```

4.6.3 SKIPTO - SKIP CODE

The SKIPTO pseudo operation enables the user to conditionally alter the sequence in which assembly lines are processed. It has the form:

label	operation	argument
	SKIPTO,exp	name1,...,namen

exp Optional, must be evaluable.

namei A valid label appearing on a CONT, DEND, or PEND statement which follows the SKIPTO statement.

If the expression is not present, only a single label is permissible.

SKIPTO operates as follows:

a) If no expression is present on the SKIPTO line, skip succeeding lines until a line with the appropriate label is found.

b) If an expression is present, it is evaluated.

1) If value of the expression is k and k lies between 0 and n-1 where n is the number of labels on the SKIPTO directive, the succeeding lines are skipped until a CONT, DEND, or PEND statement is found which has as its label, namek.

2) If the value of the expression is < 0 or >= n (or the expression is not evaluable), assembly resumes at the line immediately following the SKIPTO pseudo instruction.

Note that when in the skipping mode, all pseudo instructions except LOCAL, OPEN and CLOSE are ignored. Any symbol defined by LOCAL or OPEN pseudo instructions are not recognized. Labels within PROC/PEND, WHILE/DEND, or DO/DEND blocks are not recognized, and it is illegal to write a SKIPTO pseudo instructions which branches out of a procedure definition, WHILE/DEND sequence, or DO/DEND sequence.

Example:

```
.  In the following example, the statement processed following
.  the first SKIPTO directive depends on the value of "A".

         SKIPTO,A    SMALL,MEDIUM,LARGE,HUGE
UNEXPT   SKIPTO  MORE                  THIS STATEMENT IS PROCESSED
.                                      IF A IS NOT EQUAL TO 0, 1, 2
.                                      OR 3.
SMALL    RES     50                    THIS STATEMENT IS PROCESSED
.                                      IF A IS EQUAL TO 0.
         SKIPTO  MORE
MEDIUM   RES     100                   THIS STATEMENT IS PROCESSED
.                                      IF A IS EQUAL TO 1.
         SKIPTO  MORE
```

```
LARGE     RES      250                THIS STATEMENT IS PROCESSED
.                                     IF A IS EQUAL TO 2.
          SKIPTO   MORE
HUGE      RES      1000               THIS STATEMENT IS PROCESSED
.                                     IF A IS EQUAL TO 3.
MORE      CONT
           .
           .
```

. If "RES" is a user-defined procedure which reserves the
. number of words of core specified by its argument, then the
. amount of core reserved by the above code varies depending on
. "A".

. This example illustrates the effect of OPEN/CLOSE and DO/DEND
. blocks on the SKIPTO directive.

```
           .
           .
          SKIPTO   X
           .
           .
          OPEN     X
X         RES      5                  THIS LINE IS  SKIPPED  BECAUSE
                                      IT APPEARS
          CLOSE    X                    BETWEEN AN OPEN AND CLOSE
           .
           .
I         DO       10
          LOCAL    X
X         VFD,16   I                  THIS  LINE  IS SKIPPED BECAUSE
                                      IT APPEARS WITHIN A DO/DEND
          DEND                          BLOCK
           .
           .
X         ADD      BASE,DISP          THIS   LINE    IS    PROCESSED
                                      FOLLOWING THE SKIPTO DIRECTIVE
```

4.7 ERROR CONTROL

4.7.1 ERROR PSEUDO OPERATION

label	operation	argument
	ERROR,exp,label	C'message'

The ERROR pseudo operation provides a method for conditionally generating an error message in the object listing and transferring control to another portion of the program.

label — Label is any valid symbol appearing in the label field of a subsequent CONT, DEND, or PEND statement. The statement must be a CONT, DEND, or PEND statement before label comparison is made.

exp — Exp is a conditional expression whose value determines whether the error message is to be produced and if a transfer of control is necessary. If this subfield is omitted, then the message is unconditionally generated.

message — Message is any valid combination of characters (see Character set).

When an ERROR pseudo instruction is encountered, the expression is evaluated.

> If it is true (1) or not specified, the error message is produced on the object listing. If symbol is present, control is transferred to the indicated line. If no symbol is present, assembly continues with the next statement.
>
> If the expression is false (0), no message is produced and assembly is continued at the succeeding line.

Example:

```
        .
        .
        ERROR,A<0      C'ILLEGAL ARGUMENT'
NELX    SET            2,3,A,M,XOR,COMX

.  WHEN THE ABOVE DIRECTIVE IS ENCOUNTERED, IF A IS LESS THAN
.  ZERO THEN THE MESSAGE "ILLEGAL ARGUMENT" WILL BE PRINTED.  IF
.  A IS NOT LESS THAN ZERO, NO MESSAGE WILL BE PRINTED.  IN
.  EITHER CASE, THE LINE NELX WILL BE PROCESSED NEXT.

        ERROR,B<0|B>15,ILR   C'ILLEGAL REGISTER'
PSRL    LPD,2  B
        SKIPTO NEWL
ILR     ERR
NEWL    CONT

.  WHEN THIS ERROR DIRECTIVE IS ENCOUNTERED, IF 0≤B≤15, NO
.  MESSAGE IS PRINTED OUT AND THE LINE PSRL IS PROCESSED,
.  FOLLOWED BY LINE NEWL.  IF B<0 OR B<15, THEN THE MESSAGE
```

```
.  "ILLEGAL REGISTER" WILL BE PRINTED AND THE LINE ILR IS
.  PROCESSED, FOLLOWED BY THE LINE NEWL.  IN THIS EXAMPLE, "LPD"
.  AND "ERR" ARE USER-DEFINED PROCEDURES.
```

4.7.2 FLAG - CONDITIONALLY SET ERROR FLAG

A FLAG pseudo instruction produces an assembly error, but does not affect other code.

label	operation	argument
	FLAG	errtype

errtype   FATAL - a fatal error detected.
            WARNING - a non-fatal error detected.

Example:

```
      FLAG   FATAL
```

4.8 LISTING CONTROL

The instructions described in this section permit extensive control of the assembly listing format.

4.8.1 LIST - SELECT LIST OPTIONS

label	operation	argument
	LIST,val	exp_1,exp_2,exp_3

The LIST pseudo operation controls the assembly listing generated. The argument field is used to select the various listing options.

val   Val is an optional evaluable expression which is interpreted as follows:

0 = List this statement according to the listing controls in effect when this statement is encountered.

1 = List this statement according to the value of expression 3. This is the default.

exp_1 An evaluable expression which may assume the following values:

0 = Suppress complete listing.

1 = List input statements.

2 = List input statements plus all statements that generate code (VFD, CMD statements that normally would not be listed).

3 = List all generated statements including internal procedure expansions.

4 = List all generated statements.

exp_2 An evaluable expression used to control the listing of unprocessed statements that are by-passed during the assembly procedure and also the repeated statements in a DO/WHILE which normally would not be listed. This may occur during the processing of SKIPTO, DO and WHILE pseudo instructions. The values of the expression are as follows:

0 = List only processed statements, but not repeated DO/WHILE statements.

1 = List processed statements including repeated DO/WHILE statements that are processed.

2 = List all statements.

exp_3 Used to control the listing of the listing control pseudo instructions, TITLE, PAGE, SPACE, XRSY, and LIST. The values of this expression are as follows:

0 = Do not list the Listing control statements.

1 = List the Listing control statements.

The standard LIST parameters established by default are:

```
LIST 1,2,1
```

Causing a full listing to be generated. Subsequently any of these parameters may be altered. A null subfield specifies that the parameter is to be unchanged. If no parameters are specified, the LIST options will revert back to their previous settings.

4.8.2 PAGE - EJECT PAGE

label	operation	argument
	PAGE	

The appearance of this pseudo operation will cause the next line of output to appear at the top of a new page on the computer listing. If the next line would normally appear at the top of a new page, the PAGE pseudo operation is ignored. Two consecutive PAGE directives will generate a blank page.

4.8.3 SPACE - SKIP LINES

label	operation	argument
	SPACE	expression

expression Expression is any evaluable expression. The value of this expression specifies the number of lines to be spaced before the next line appears on the computer listing.

If the expression is not present, a value of 1 is assumed. If the value of the expression is greater than the number of lines remaining on the page, the SPACE pseudo operation will have the same effect as the PAGE pseudo operation.

Example:

```
        SPACE   3
```

4.8.4 TITLE - ASSEMBLY LISTING TITLE

label	operation	argument
	TITLE	C'character string'

character string
Character string is a sequence of any characters (see Character Set) up to a maximum of 56 characters.

The TITLE pseudo instructions enables the programmer to specify an identification for assembly listing.

When a TITLE pseudo instructions is encountered, the assembly listing is advanced to a new page (if it is not already at a new page). The indicated character string is printed at the top of this page and at the top of all succeeding pages until another TITLE pseudo instruction is encountered or the end of assembly is reached.

A null argument field on a TITLE pseudo instruction line will cause the listing to be advanced to a new page, but no heading printed.

Example:

```
        TITLE   C'TESTCODE'
```

4.8.5 XRSY - CONCORDANCE SELECTION

label	operation	argument
	XRSY	name1,...,namen

The XRSY pseudo operation is used to select certain symbols to be included in the concordance.

namen — Namen designates symbols to be included in the concordance.

Example:

```
        XRSY    X0
```

4.9 SECTIONS

Sections are established for the user by the Assembler, and optionally by the user. The concept of sections is valid only for CPU programs. Sections in a CPU module are established with differing levels of access to allow the user who uses them protection for code and data. The concept of sections is similar to the hardware concept of segments. Hardware segments are established to have different levels of access, and generally so are the Assembler sections. However, sections can be established with the same level of access, and they will then be combined into the same hardware segment.

Sections can be used to establish a blocking of data and code. The section counter is automatically maintained by the Assembler, but can be modified by using the ORG, POS or BSS pseudo instructions.

Data and code within a section is not relocatable. The sections are treated as relocatable with references made via the use of pointers. The CYBER 180 instruction set has been designed to efficiently access data and code in other sections via a mechanism of pointers to a byte address plus an offset in the specific section. The pointers are generally established via the ADDRESS pseudo instruction in the BINDING section.

4.9.1 SECTION - ESTABLISH BLOCK

SECTION establishes a new block. This statement is valid only for a CPU module. A user may establish up to 10 sections in addition to the five default sections established for him. All SECTION pseudo instructions must appear before any code or data generation instructions are specified.

label	operation	argument
name	SECTION	type,attr,cid,algn,maxsize

name (Required) Internal section name for USE block definition.

type (Required) The section type identifier which must be one of the following names:

CODE	Code section, only one code section is permitted per module.
BINDING	Binding section, only one binding section is permitted per module.
WORKING	Working storage section.
COMMON	Common block section.
EXTWORK	Extensible working storage section. Data may not be established in sections of this type at Assembly time.
EXTCOM	Extensible common block section. Data may not be established in sections of this type at Assembly time.

attr (Required) An absolute expression which specifies legal combinations of access attributes of the segment to contain the section. Only the "+"

operator is permitted in the expression.

READE - Read
WRITE - Write
EXECUTE - Executable
BIND - Binding
CACHE_BYPASS - cache bypass (hardware feature)

cid — (Optional) Common section name (1-31 character alias name).

algn — (Optional) Two absolute expressions separated by a comma which define section alignment. The first parameter is an offset, the second is the base (modulus).
Examples are:

0,8 - Word aligned section start.

8,64 - Section starts at word one of an 8 word block boundary.

0,8 - Word aligned section start (default for all sections except binding sections).

maxsize — (Optional) Absolute evaluable expression which specifies the maximum section size.

The following default sections are established by the Assembler for a CPU module:

Section Name	Attributes
*CODE	Read+Execute
WORKING	Read+Write
BINDING	Read+Bind
STACK	Read+Write

* The name on the IDENT card can also be used to reference the CODE section.

Example:

```
DUMMY SECTION   WORKING,READ+WRITE,,0,8
```

4.9.2 USE - USE BLOCK

The USE statement is valid only for CPU modules. USE starts/resumes use of an already established section into which code is subsequently assembled.

label	operation	argument
	USE	name

name — The name of the section into which the text that follows is assembled. (It corresponds to the name of a SECTION pseudo instruction). A blank name causes the assembly of code into the default CODE section. The name #LASTSEC will resume using the section in use prior to the last USE statement.

The current position in a section is automatically maintained by the Assembler. When the USE pseudo instruction is executed, the section counter will automatically be restored to its previous value.

Example:

```
DUMMY   SECTION   WORKING,READ+WRITE,,0,8
           .
           .
           .
        USE       DUMMY
           .
           .
           .
        USE       #LASTSEC
```

4.9.3 ORG - SET SECTION COUNTER

The ORG pseudo instruction specifies the byte offset to which the section counter is to be set.

label	operation	argument
label	ORG	exp

label — Optional, if present, is set to the value of exp. Symbol category equals 6.

exp — An absolute expression specifying the address to which the unit offset is to be set. Any symbols in the expression must have been previously defined.

Example:

```
TAG   BSS   10    .DATA AREA.
       .
       .
       .
      ORG   TAG   .STORE IN DATA AREA.
```

4.9.4 POS - SET BIT POSITION IN THE SECTION COUNTER

The POS pseudo instruction sets the value of the bit offset in the section counter to the value specified by the expression in the argument field.

label	operation	argument
	POS	aexp

aexp — An absolute, evaluable expression having a positive value less than or equal to the bit position with a byte. A negative value, or a value greater than 7 causes an error. The value indicates the bit position within the current address unit at which the Assembler is to generate the next data. Use caution, because if the new bit position value is less than the old bit position value, part of the byte is reassembled. (New code is ORed with previously assembled data). If the new bit position value is greater than the old bit position value, the Assembler generates zero bits to the specified bit position.

CAUTION: If the POS pseudo instruction is used on a word containing relocatable or external addresses, undefined results may occur with no diagnostics.

The POS pseudo instruction does not alter the byte offset. The POS instruction never causes the byte to be changed.

Example:

```
      POS   3
```

4.9.5 BSS - STORAGE RESERVATION

The BSS pseudo instruction reserves memory in the section in use by adjusting the addressable byte offset. It does not generate data to be stored in the reserved area.

label	operation	argument
label	BSS	aexp

label Optional label defined as the addressable unit offset after the force to an addressable unit boundary occurs. It is the beginning symbol for the storage area. Symbol category equals 6.

aexp Absolute expression specifying the number of addressable storage units to be reserved. All symbols must be previously defined. Aexp cannot contain external symbols or be relocatable. The value of the expression can be zero or positive, but not negative, and the value is added to the addressable units offset. A BSS 0 causes a force to byte boundary and symbol definition, but no storage is reserved.

Example:

```
TAG   BSS   5
```

4.9.6 ALIGN - FORCE SECTION COUNTER ALIGNMENT

The ALIGN pseudo instruction forces the unit offset to the specified byte boundary and sets the bit offset to zero.

label	operation	argument
label	ALIGN	increment,unitsize

label Optional label defined as the unit offset after the force to the specified offset plus increment occurs. Symbol category equals 6.

increment The increment is a value that is added to the unit offset after the alignment is made to a unitsize boundary.

unitsize The unitsize specifies a value by which the unit offset must be evenly divisible. The number specified must be greater than zero. To do this, a number between 0 and unitsize -1 is added to the unit offset to make it evenly divisible.

Example:

```
ALIGN 0,2    .PARCEL BOUNDARY (CPU).
ALIGN 0,8    .WORD BOUNDARY (CPU).
```

4.10 PROCEDURES

A procedure definition is a sequence of source statements that are saved and then assembled whenever needed through a procedure call. A procedure call consists of the occurrence of the procedure name in the operation field of a statement. It usually includes parameters to be substituted for formal parameters in the procedure code sequence so that code generated can vary with each procedure call.

Use of a procedure requires two steps, definition of the procedure sequence, and calling of the procedure.

A definition consists of three parts: heading, body, and terminator.

Heading — A PROC definition is headed by a PROC pseudo instruction initiating the definition of a procedure, and a PNAME pseudo instruction stating the name of the procedure.

Body — The body begins with the first statement in a definition after the heading. The body consists of a series of symbolic instructions. All instructions other than PEND, including other procedure calls are legal within a definition. Within a PROCEDURE, calls can appear to other Procedures, but a PROCEDURE cannot call itself nor can any PROCEDURE in a nest of calls call any other PROCEDURE previously in the nest. PROCEDURE definitions cannot be nested. That is, a PROC pseudo operation must be followed by a PEND pseudo operation prior to the appearance of another PROC pseudo operation. The overall order of PROCEDURE definition is immaterial so long as the definition precedes the first call to assemble the PROCEDURE (i.e. a procedure call within a procedure definition may reference a procedure that is not defined prior to this point).

Terminator — A PEND pseudo instruction terminates a procedure definition.

A procedure can be defined anywhere in a program before it is called. When the Assembler encounters a definition, it places the name of the procedure along with the number of substitutable parameters and local symbols in the Assembler operation code table.

4.10.1 PARAMETER REFERENCING WITHIN PROCEDURES

Parameters on a procedure call can be referenced using the Field function "F:" and specifying the position of the parameter. The position of the parameter is indicated by using an (i,j) notation to describe where on the procedure call the parameters should be gotten. Using the (i,j) notation, i describes the field number (label field = 0, operation field = 1, argument field = 2), and j describes the position in the field starting at 0. An entire field may be referenced by just quoting the first parameter.

When a label is specified on the PROC statement, that label is equated to the Field function and can optionally be used instead of F: (the colon is part of the Field function name). For more information, refer to the section discussing the PROC statement.

Using F: notation, the i**th field and the j**th subfield of a statement is referenced as:

```
F:(i,j)
```

A reference to the entire i**th field would be:

```
F:(i)
```

References to a particular field or subfield may occur anywhere that such a reference has meaning. Each reference acts as a direct substitution of the referenced subfield into the referencing entity. The actual substitution mechanism can have several meanings which are discussed in subsequent chapters.

4.10.1.1 Parameter Identification Examples

```
.  THIS EXAMPLE SHOWS HOW RELATIVE FIELD IDENTIFICATION WORKS.
.  CONSIDER TRANSLATION OF THE FOLLOWING LINE:

IMPERAT   ADD,3,4   ADDEND,AUGEND   MOVE,5,6   DEST,SOURCE
```

```
.  DURING PROCESSING OF THE OPERATION:
.        F:(0) = IMPERAT
.        F:(1,0) = ADD       F:(1,1) = 3         F:(1,2) = 4
.        F:(2,0) = ADDEND    F:(2,1) = AUGEND
.        F:(3,0) = MOVE      F:(3,1) = 5         F:(3,2) = 6
.        F:(4,0) = DEST      F:(4,1) = SOURCE
```

4.10.2 PROC - PROCEDURE HEADING

The PROC pseudo instruction is the first pseudo instruction which must be given in the process of defining a PROCEDURE. This pseudo instruction may contain an optional label field. Following the PROC pseudo instruction must appear the statements which comprise the entire PROCEDURE being defined. The appearance of the PROC pseudo instruction initiates definition of a PROCEDURE. All statements which follow the PROC pseudo instruction up to and including the first encountered PEND pseudo instruction will be included as part of the PROCEDURE being defined.

The PROCEDURE being defined will be considered terminated when the first subfield of any subsequent OPERATION field contains the pseudo instruction PEND. All statements of the PROCEDURE which lie between the PROC pseudo instruction and the next PEND pseudo instruction are considered to be the body of the PROCEDURE. Within this PROCEDURE body, the first subfield of any subsequent OPERATION field prior to a PEND pseudo instruction cannot contain another PROC pseudo instruction.

label	operation	argument
label	PROC	

label — Optional, the label field of a PROC pseudo instruction contains a symbol, this symbol can then be used as a field function name within the procedure body and also by any other (nested) procedures. Note that the label is defined only while the procedure is active (referenced), and cannot be used to call the procedure.

The label on the PROC pseudo instruction line is normally used within the PROCEDURE followed by field and subfield notation to reference the actual arguments by which the PROCEDURE was called. If no label appears with the PROC pseudo instruction, then the parameters by which the procedure is called can be referenced only by using the F: notation described in the previous section.

Examples can be found in the section entitled "Procedure Examples".

4.10.3 PNAME - PROCEDURE NAME DEFINITION

The PNAME pseudo instruction is used to provide a name by which a PROCEDURE can be referenced. The PNAME pseudo instruction must immediately follow the PROC, FNAME, or another PNAME pseudo instruction when a PROCEDURE is being defined. Any PROCEDURE may have multiple PNAME pseudo instructions and, therefore, be referenced by several names.

label	operation	argument
label	PNAME	value

label   Name by which the procedure is referenced.

value   An evaluable expression.

Within the PROCEDURE, the value of the expression following the name by which the PROCEDURE was actually referenced is available as F:(1,0). This permits the programmer to distinguish between referencing names, when desired.

A PROCEDURE is referenced (as a procedure) by placing one of its defined PNAME's in the first subfield of a OPERATION field. The expression which represents the value associated with the PNAME is evaluated each time the PROCEDURE is referenced using that name.

Examples can be found in the section entitled "Procedure Examples".

4.10.4 FNAME - FUNCTION NAME DEFINITION

The FNAME pseudo operation is used to provide a name by which a PROCEDURE may be referenced as a FUNCTION. The FNAME pseudo operation must immediately follow the PROC, PNAME or another FNAME pseudo operation when a PROCEDURE is being defined. Any PROCEDURE may contain multiple FNAME pseudo instructions and, therefore, be referenced by several names.

label	operation	argument
label	FNAME	value

label — Name by which the procedure is referenced as a function.

value — An evaluable expression.

Within the PROCEDURE, the value of the name by which the PROCEDURE was actually referenced is available as F:(1,0). This permits the programmer to distinguish between referencing names, when desired.

A PROCEDURE is referenced (as a function) by forming a structure:

```
name(argument)
```

Where name is its defined FNAME and argument is the argument to the PROCEDURE. This bounded argument, less parentheses, is available, starting at F:(2,0), just as if the PROCEDURE was referenced as a procedure (via PNAME). The argument is limited to one field, although it may contain as many subfields as necessary. No blanks may appear between the argument and the enclosing parentheses. The expression which represents the value associated with the FNAME is evaluated each time the PROCEDURE is referenced using that name.

A PROCEDURE, referenced using one of its FNAME's will have the entire reference replaced by the value of the expression on the PEND pseudo instruction when the PEND pseudo instruction is executed. This value will always be 8 bytes long.

Note that a function may not generate code or change location counters if it is invoked from a statement which, itself, is generating code.

Examples can be found in the section entitled "Procedure Examples".

4.10.5 PEND - END PROCEDURE DEFINITION

A PEND terminates any unterminated definition. A PEND outside the range of any procedure sequence has no effect other than to be included in statement counts.

label	operation	argument
label	PEND	exp

label (Optional) May be used as the object of a skip by a SKIPTO or ERROR statement. The label symbol is not entered into Assembler's symbol table and the presence of a label does not constitute symbol definition.

exp The argument field can be null or can be an evaluable expression. When the PROCEDURE is called as a procedure reference, any PEND expression is ignored. When a PROCEDURE is called as a function reference, the PEND expression is evaluated and the value is returned as the value of the function. A null expression returns the value zero.

Examples can be found in the section entitled "Procedure Examples".

4.10.6 LOCAL - ESTABLISH LOCAL SYMBOLS

The LOCAL pseudo instruction is used to establish symbols which are to be considered local to the PROCEDURE in which they are defined. The appearance of a LOCAL pseudo instruction supersedes all previous LOCAL pseudo instructions in that program or PROCEDURE and all symbols previously declared local are erased. A PEND or END line terminates the LOCAL.

label	operation	argument
	LOCAL	name1,...namen

name1,...namen Establish symbols local to a procedure.

A symbol may not be defined as LOCAL if its symbol category is one of the following:

2	CMD defined instruction
4	PROCEDURE call
10	PROCEDURE Reference List
12	ANAME defined symbol (programmer defined attribute)
13	Section counter

Examples can be found in the section entitled "Procedure

Examples".

4.10.7 OPEN - DECLARE TEMPORARY SYMBOLS

The OPEN pseudo instruction is used to declare temporary symbols without affecting any prior use of the label. A label declared by an OPEN pseudo instruction remains active until closed by a CLOSE pseudo instructio using the same label. OPEN pseudo instructions may be nested using the same label. The label created under the last OPEN pseudo instruction executed will be active until closed. It is important to note that closing opened symbols takes place in reverse order from the opening process. That is, the last open symbol is closed first, then the next-to-last, etc. Subsequent OPEN pseudo instructions only affect each other if they use the same symbol, otherwise they act independently without cancelling prior OPEN pseudo instructions as is the case with LOCAL pseudo instruction. Definitions of OPEN'ed symbols are restricted in the same way as LOCAL symbols.

label	operation	argument
	OPEN	name1,...,namen

name1,...,namen Establish temporary symbols with names name1,...,namen

Examples can be found in the section entitled "Procedure Examples".

4.10.8 CLOSE - ERASE TEMPORARY SYMBOLS

The CLOSE pseudo instruction erases the symbols whose names are used as arguments to the pseudo instructions. If a symbol has been opened by more than one OPEN pseudo instruction, then CLOSE only erases the last OPEN and the symbol usage then reverts to its usage under the previous OPEN. If there was only one OPEN associated with the symbol, the symbol becomes non-existent and is completely erased. It is illegal to CLOSE a symbol that has not been opened.

label	operation	argument
	CLOSE	name1,...,namen

name1,...namen — Erase temporary label field symbols with names name1,...namen.

Examples can be found in the section entitled "Procedure Examples".

4.10.9 CONT - NO OPERATION

The CONT pseudo instruction is used to place a symbol on a statement only for the purpose of assembly time transfer of control. The CONT pseudo instruction functions in all other respects as a no-op.

label	operation	argument
label	CONT	

label — (Required) Symbol used for transferring control during the assembly process. The symbol is not entered in Assembler's symbol table and use of a symbol in the label field does not constitute symbol definition.

Examples can be found in the section entitled "Procedure Examples".

4.10.10 PROCEDURE CALLS

A procedure headed by the PROC pseudo instruction can be called by an instruction in the following format:

label	operation	argument
label	procname	field1,field2,...fieldn

label — Optional, its value can be retrieved from within the procedure's body by the F:(0) field function.

procname — Name of a predefined procedure (label on PNAME).

fields — One or more fields which might consist of several subfields.

A defined PROCEDURE may be referenced using any one of its names as defined by a PNAME or FNAME pseudo instruction. This name is written as the first subfield of the OPERATION field. The remainder of the OPERATION field and as many argument fields as necessary can follow the OPERATION subfield and contain the arguments to the PROCEDURE. The Assembler is capable of handling as many arguments as the user wishes to provide.

Parameters passed to PROCEDURES are call by name in that a parameter is evaluated each time it is referenced within the body of a PROCEDURE. Any previous statements within the body of the PROCEDURE which have changed the value of a given parameter will affect later references to the parameter. Any OPEN or LOCAL pseudo instructions within the body of a referenced PROCEDURE which declare labels with the same symbol as a label passed as a parameter will not affect the parameter being passed.

It is the actual call to a PROCEDURE which requires that it be defined and not just the appearance of a call in an Assembler statement. Unexecuted calls do not require that the named PROCEDURE be defined.

4.10.11 PROCEDURE EXAMPLES

4.10.11.1 Procedure Definition

```
.  THIS IS AN EXAMPLE OF THE USE OF PROCEDURES.  THE
.  PROCEDURE STATEMENTS (THOSE APPEARING BETWEEN A PROC AND
.  A PEND DIRECTIVE) ARE NOT PROCESSED UNTIL THE PROCEDURE
.  NAME APPEARS IN THE OPERATION FIELD OF A STATEMENT BEING
.  PROCESSED.  IN THIS EXAMPLE, AFTER THE STATEMENT LABELLED
.  "CALLING" IS ENCOUNTERED, PROCESSING OF THE STATEMENTS IN
.  PROCEDURE "SAM" BEGINS.  WHEN THE PEND DIRECTIVE IN "SAM"
.  IS ENCOUNTERED, PROCESSING RESUMES AT "NEXTLINE".
SAMR     PROC
SAM      PNAME  5
A        SET    F:(2,0)              .F:(2,0) REFERENCES X*3
B        SET    F:(2,1)              .F:(2,1) REFERENCES ZXT
F:(2,2)  ANAME  6                    .ASSIGNS NAME INDEX TO
                                     .ATTRIBUTE NUMBER 6
         MAX    SAMR(2,2),SAMR(2,1)
.
         PEND
         PROC
MAX      PNAME
F:(2,1) ATRIB F:(2,0),5              .INTERPRETED AS: ZXT ATRIB
.                                    .INDEX,5
         PEND                        .  F: REFERENCES LINE CALLING
           .                         .  MAX,
           .                         .  SAMR    REFERENCES     LINE
```

```
                                              CALLING
            .                               . SAM
            .
CALLING  SAM    X*3,ZXT,INDEX
NEXTLINE VFD,16 ZXT

.  THE ABOVE CODE IS EQUIVALENT TO:

A        SET    X*3
B        SET    ZXT
INDEX    ANAME  6
ZXT      ATRIB  INDEX,5
NEXTLINE VFD,16 ZXT

.  THE FOLLOWING EXAMPLE INVOLVES TWO DIFFERENT DEFINITIONS OF
.  THE LABEL X.  THE NET EFFECT OF THIS CODE IS TO SET THE VALUES
.  OF X AND Y TO 7:

         PROC
ZED      PNAME
         LOCAL X
X        SET    2
F:(2,0)  SET    F:(2,0)+X
Y        SET    F:(2,0)
         PEND        .WHEN EXECUTED LOCAL X NO LONGER EXISTS
            .
            .
X        SET    5    .GLOBAL X
         ZED    X    .GLOBAL X AS PARAMETER F:(2,0)

.  THIS PROCEDURE DEFINES A SET OF INSTRUCTIONS FOR THE C180 CPU
.
.  EACH OPERATION CODE IS SPECIFIED AS A PROCEDURE ENTRY NAME
.  WHEN HAS THE MACHINE CODE AS THE VALUE.
.
.  THESE INSTRUCTIONS ARE IN THE FORM      OP   R1,R2    WHERE
.  R1 AND R2 SPECIFY REGISTERS.
.
         PROC
ADDR     PNAME 20(16)
SUBR     PNAME 21(16)
MULR     PNAME 22(16)
DIVR     PNAME 23(16)
SR       PNAME 1B(16)
.
F:(0,0)  VFD,8,4,4  F:(1,0),F:(2,1),F:(2,0)
         PEND
```

4.10.11.2 LOCAL Directive's Use

```
A        SET    5         THIS IS A GLOBAL "A"
          .
          .
         PROC
EVAL     PNAME
         LOCAL  A,B,C     ANY REFERENCES TO A, B, OR C WITHIN THE
.                         EVAL PROCEDURE SIGNIFY SYMBOLS LOCAL
A        SET    7         LOCAL "A"
D        SET    A         GLOBAL "D", LOCAL "A"
B        SET    A         LOCAL "B", LOCAL "A"
.                         AT THIS POINT, VA:(A) = 7, VA:(B) = 7,
          .
          .
         PEND
          .
          .
C        SET    A         GLOBAL "C", GLOBAL "A"
E        SET    D
.                         AT THIS POINT, VA:(A) = 5, VA:(C) = 5,
.                         VA:(D) = 7, AND VA:(E) = 7
```

5.0 ATTRIBUTE FUNCTIONS

The Assembler provides a set of built in functions to assign and/or retrieve values of a symbol attribute. They are usually used to aid in parameter analysis in procedure and function definitions.

An attribute function is a replacement operation in which the value of the specified attribute replaces the function in the expression. The permitted arguments to an attribute function are defined later in this section.

The set of Symbol Attribute Functions (SC:, VA:, LB:, LC:, LW:, SB: and SN:), and the basic Field Reference Function ("F:" used for parameter referencing), all include the character ':' (colon), which is an alphabetic character within the meaning of the Assembler. This character is included as a means of avoiding potential conflicts with user-defined symbols, and does not represent an operator of any kind. Note that this character must be entered in the NOS ASCII representation.

The general form of an attribute function is:

    attribute_function_name(argument)

where attribute_function_name is the name of a specific attribute function, and the argument, enclosed in parentheses, immediately follows.

All of the symbol attributes discussed in the section on Symbol Definition have a corresponding attribute function which can be used to retrieve that particular symbol attribute from the internal Assembler symbol table.

5.1 LANGUAGE DEFINED ATTRIBUTES

All the attribute functions described in this section are built into the Assembler.

5.1.1 SYMBOL CATEGORY ATTRIBUTE - SC:

    Format: SC:(argument)

The SYMBOL CATEGORY Attribute function is used to determine the symbol category assigned to the argument. The argument can be a symbol name or a PROCEDURE reference field specification. This function returns the value of the category and may be used for testing. When the argument refers to an expression rather than a symbol, the category of the expression will be the category of the first term in the expression. The category of a NULL subfield in a PROCEDURE reference is zero. The Symbol Category attribute has the following values and meaning:

Category	Meaning
0	Non-existent symbol. The symbol in question has not been encountered by the Assembler. The existence of a blank LABEL field can be detected by this category.
1	The symbol has appeared in a LABEL field, may have certain attributes, but no operation has taken place to further define the symbol. After each statement is processed, any remaining category 1 symbols are erased from the symbol table, unless they have programmer defined attributes.
2	The symbol has been defined by a CMD pseudo instruction and is now recognized as an instruction generating symbol.
3	The symbol is an Assembler defined function.
4	The symbol is a PROCEDURE call, defined by an FNAME or PNAME pseudo instruction.
5	The symbol is an Assembler pseudo instruction.
6	The symbol is a relocatable address defined by use in a code generating statement such as VFD, INT, DINT, FLOAT, DFLOAT, PDEC, BSS, BSSZ, ADDRESS, ORG, ALIGN, or by the execution of an instruction generating symbol defined by a CMD pseudo instruction.
7	The symbol was defined by a REF pseudo instruction.
8	The symbol is the symbol "$" (section counter).
9	The symbol is a list name defined by a SET or EQU pseudo instruction or as the label of a DO or WHILE pseudo instruction.
10	The symbol is a list name of a symbolic list

holding PROCEDURE references. The symbol was defined by a PROC pseudo instruction (see PROCEDURES).

11 The symbol is a self-defining term.

12 The symbol is defined by an ANAME pseudo instruction.

13 The symbol is a list defined by a SECTION pseudo instruction.

Symbols defined in the label field of pseudo instructions where the label field is not ignored will have the symbol category documented for that instruction. Symbols defined in the label field of the symbolic machine instructions will have a Symbol Category of 6.

5.1.2 ADDRESS MODE ATTRIBUTE

Format: AM:(argument)

The ADDRESS MODE attribute function is used to determine the relocatability of the argument. The argument can be a symbol name or a PROCEDURE field reference specification. This function returns the value 1 if and only if the argument is defined and relocatable. Otherwise, it returns a value of zero. When the argument refers to an expression rather than a symbol, the ADDRESS MODE will be the ADDRESS MODE of the first term in the expression. When the symbol is the symbol "$", the address mode value will be 0.

5.1.3 VALUE ATTRIBUTE

Format: VA:(argument)

The VALUE attribute is used to determine the value assigned to the argument, where argument is either a symbol or a PROCEDURE field reference specification. The meaning of the VALUE attribute varies with the symbol category of the argument:

SYMBOL CATEGORY	VALUE and/or MEANING
0	0
1	0
2	0
3	0

4	The value of the PNAME/FNAME symbol when the procedure is called.
5	0
6	(Integer) address assigned to the symbol.
7	0
8	The current integer location counter value.
9	The value of the first element of the list.
10	0
11	The (word) value of the self-defining term.
12	The value is the programmer defined attribute number assigned to the symbol.
13	The value of the first element of the list (the integer location counter).

The value of an expression is the net value found by evaluating the expression. A NULL field or subfield has the value of zero.

The VALUE attribute function is processed in a similar manner to normal expression evaluation, except that errors caused by invalid use of symbols are suppressed.

5.1.4 LENGTH ATTRIBUTES

Format: LB:(argument)
LC:(argument)
LW:(argument)

The LENGTH attribute is used to determine the length in bits (LB:), bytes or cells (LC:), or words (LW:) of the argument, where the argument is a symbol representing a data or instruction area assigned by the Assembler in a module. A CYBER 180 CPU word is 64 bits long.

The LENGTH function rounds up to the next integral number of units in cases where the bit length of the argument is not an exact multiple of the defined character or word. LENGTH returns the value 0 if a symbol has not been defined at the time the evaluation of LB:, LC:, or LW: takes place.

As explained in the section on SYMBOL DEFINITION, a symbol acquires a length attribute when it becomes defined by appearing in the LABEL field of a data generating pseudo instruction. This length attribute is the quantity of storage assigned to the information labeled with the symbol. A Self-Defining Term has a LENGTH attribute assigned to that term based on its structure.

If the symbol has been defined with a code generating pseudo instruction (category 6) then the bit length is given by the total number of bits generated by the statement. Applicable pseudo instructions include VFD, INT, DINT, FLOAT, DFLOAT, PDEC, BSS, BSSZ, ADDRESS, and CMD calls. A character is assumed to be 8 bits, and the word size is taken to be 64 bits for a CPU module.

If the argument is a self-defining term, the length is determined based on its structure. A character string (types C and E) have a character/byte length equal to the number of characters in the string, a bit length of 8*LC. For all other types of self-defining terms, the bit length is equal to the appropriate CYBER 180 word size.

5.1.5 STARTING BIT POSITION ATTRIBUTE

Format: SB:(argument)

The STARTING BIT POSITION attribute is used to determine the value of the BIT offset in the stored byte at the time storage was assigned to the argument. This function has a zero value for all arguments whose symbol category is not equal to 6. The STARTING BIT POSITION attribute for an expression is the STARTING BIT POSITION attribute of the first term in the expression. The STARTING BIT POSITION attribute of a NULL field or subfield is zero. The maximum value for this attribute is 15.

5.1.6 ELEMENT NUMBER ATTRIBUTE

Format: EN:(argument)

The ELEMENT NUMBER attribute determines the number of subfields (elements) associated with or assigned to the argument. The argument can be any list name and the value of the EN: function will be the number of elements assigned to the list at the time evaluation takes place. Note that a symbol name becomes defined as a list only by appearing in the LABEL field of the SET pseudo instruction.

When a PROCEDURE field reference is used as an argument to the EN: function, then one of two forms of substitution take place:

a) If the specification contains a field index and no subfield index (F:(0),F:(1),...etc.), then the count is made against the actual subfield elements in the PROCEDURE reference line itself.

b) If the specification contains both a field AND subfield index (F:(0,0),F:(2,0),...etc.), then the count is made against the contents of the designated subfield.

5.1.7 LAST ELEMENT NUMBER ATTRIBUTE

Format: EL:(argument)

The LAST ELEMENT NUMBER attribute determines the element number of the last element assigned to the list used as an argument. For lists with one or more elements:

EL:(argument) = EN:(argument)-1

For all non-list arguments:

EL:(argument) = 0

When a PROCEDURE field reference is used as an argument to the EL: function, then one of two forms of substitution take place:

a) If the specification contains a field index and no subfield index (F:(0),F:(1),,,,.etc.), then the count is made against the actual subfield elements in the PROCEDURE reference line itself.

b) If the specification contains both a field AND subfield index (F:(0,0),F:(2,0),...,etc.), then the count is made against the contents of the designated subfield.

5.1.8 SYMBOL NUMBER ATTRIBUTE

Format: SN:(argument)

The SYMBOL NUMBER attribute determines a unique value representing the symbol. This value is only meaningful when used for comparison to test equality with the SYMBOL NUMBER of other symbols. If the argument does not correspond to a symbol, then a value of zero is returned.

5.1.9 RELOCATION ATTRIBUTE

The Relocation attribute is not a property of a symbolic name. The Relocation attribute is a function that is used to associate relocation information with the generation of data and as such it is meaningful only when used in an expression in the argument field of a VFD, CMD, INT, or DINT statement. See the example below. The function is valid only in a CPU module. If the CPU module is declared "NONBINDABLE", then the relocation information is ignored. This function must have three (3) arguments. The relocation function is called as follows:

```
R:(EXP,RCT,ADT)
```

EXP An expression defining the byte offset to be used as a displacement. If the expression is not relocatable in the BINDING section then no "relocation" object text is generated. The function result is the expression result divided, if necessary, as determined by the ADT subfield.

RCT Defines the relocation container type (width and alignment). This applies to the field being generated. (Note that only discrete values are permitted.): Unless otherwise indicated the field must start on an addressable boundary.

```
0 = Parcel Size          (2-bytes)
1 = Three Bytes          (3-bytes)
2 = Half Word            (4-bytes)
3 = Word                 (8-bytes)
4 = Instr.  D-Field      (12-bits/MOD 4)
5 = Instr.  Q-Field      (2-bytes)
6 = Long D-Field         (3-bytes) ENTC & ENTA Instr.
```

Any other value is diagnosed as an error.

ADT Defines the address displacement type of the field. The function result is EXP divided by a constant determined by the ADT subfield as follows:

```
0 = Byte Positive        R: = EXP
1 = Parcel Positive      R: = EXP / 2
2 = Halfword Positive    R: = EXP / 4
3 = Word Positive        R: = EXP / 8
4 = Byte Signed          R: = EXP
5 = Parcel Signed        R: = EXP / 2
6 = Halfword Signed      R: = EXP / 4
7 = Word Signed          R: = EXP / 8
```

Any other value is diagnosed as an error.

EXAMPLE:

```
VFD,16    R:(binding_sect_disp,5,5)
```

5.2 PROGRAMMER DEFINED ATTRIBUTE FUNCTIONS

Any symbol may be given one or more programmer defined attributes by first using the ANAME pseudo instruction to give each programmer defined attribute a name and then using the ATRIB pseudo instruction which assigns a value to a specific attribute of a symbol. The Assembler permits the definition up to 16 programmer defined attribute names. Each programmer defined attribute is given a name and an attribute number using the ANAME pseudo instruction:

```
INDEX      ANAME     1
BASE       ANAME     2
FREQ       ANAME     3
             .
             .
             .
           etc.
```

Once defined, a programmer defined attribute function of the form:

programmer defined_attribute_name(argument)

may be used in the same way as an Assembler defined attribute function to recover the value of a particular programmer defined attribute assigned to the argument.

When the argument to an programmer defined attribute function is an expression, the function value is the value of the named programmer defined attribute of the first symbol in the expression.

The names and values can be altered during the course of the program assembly using the ANAME and ATRIB pseudo instructions discussed in the section on pseudo instructions.

5.3 SYMBOL ATTRIBUTE EXAMPLES

```
          .
          .
          .
length    aname 1   .LENGTH IS A PROGRAMMER DEFINED ATTRIBUTE
          .
          .
          .
          proc
data      pname
.
.  This procedure generates a character string of data
.  in the WORKING section starting on a half-word
.  boundary.  It will also assign the length in
.  bytes as an attribute called length.
```

```
.
.        label   data   charstring
.
         use     working     .puts us in working section
         align   0,4         .puts us on a half-word boundary
f:(0,0)  vfd,lb:(f:(2,0))    f:(2,0)   .generate data
f:(0,0)  atrib   length,lc:(f:(2,0))   .puts byte length
         use     #lastsec
         pend
           .
           .
           .
label1   data    C'EXAMPLE'      .data procedure call
numbyte  set     length(label1)  .picks up byte length of string
           .
           .
           .
```

6.0 OFFSET FUNCTIONS (#WOFF, #HOFF, #POFF, #BOFF)

The offset functions return the Word, Half-Word, Parcel, or Byte offset of an address relative to the beginning of a CPU section in which it is defined. An informative error will be generated if label does not fall on the appropriate boundary.

The functions are:

#WOFF(label)	Returns the offset in words.
#HOFF(label)	Returns the offset in half-words.
#POFF(label)	Returns the offset in parcels.
#BOFF(label)	Returns the offset in bytes.

7.0 CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTIONS

The CYBER 180 Assembler recognizes symbolic notation for all CYBER 180 CPU Instructions. Instructions in this group are valid only for a MACHINE pseudo instruction type of C180CPU. If the MACHINE pseudo instruction type is C180IOU the mnemonics listed in this section will generate errors.

The Assembler identifies each symbolic instruction according to its mnemonic. The object code for the instruction is generated in the block in use when the instruction is encountered. For a more complete description of the hardware instructions, refer to the CYBER 180 Processor-Memory Model-Independent GDS.

7.1 SYMBOLIC NOTATION

This section describes notation used for coding symbolic CYBER 180 instructions. The CPU instructions are listed according to the CYBER 180 MIGDS Reference Numbers.

The instruction descriptions are obtained from the CYBER 180 MIGDS. Lengths will always specify the actual number. The Assembler will make any adjustments necessary, as when the hardware requires the length to be entered as length-1. Any D or Q field that is adjusted by the Assembler will be denoted by the word label in the mnemonic description, and will then be further described as to exactly what the Assembler expects for that field.

The label field of a symbolic machine instruction optionally contains a label. When the label is present, it is assigned the value of the byte offset after it is forced (if required) to parcel boundary. The symbol category of the label will be set to 6.

The operation field of a symbolic machine instruction contains an instruction mnemonic and might also contain several other subfields.

The argument field contains the instruction operands as one or more subfields.

An optional comment field may appear following the last subfield of the argument field. A comment field must begin with a period (.) character.

7.0 CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTIONS
7.1 SYMBOLIC NOTATION

The mnemonics chosen are descriptive of the actual hardware operation being performed and will provide for a high degree of recognition by the 2nd and 3rd reader of assembly language programs. In all cases, the mnemonics are 8 characters or less, and in most cases much less. This should provide for a certain ease in programming. The rules enforced when defining the instructions are:

- o A common abbreviation used when shortening the mnemonics.

- o The first part of the mnemonic describing the action to be performed.

- o The second part of the mnemonic further qualifies the type of action to be taken (X used to represent a full X register, R for right half of an X register, BIT signifying operation on a bit field, etc.).

- o The operand fields are written such that multiple subfields relating to source or destination are positioned together.

- o Implied registers are written as part of required instruction syntax.

- o The operands are written such that the most significant or resultant register is written first.

7.2 CPU INSTRUCTION FORMATS

The figures in this section illustrate the formats for the CYBER 180 16-bit and 32-bit CPU instructions generated by the Assembler. For all instructions the Assembler generates parcel alignment whenever necessary.

```
+------------------+----+----+----+-------------+
| Operation Code   | j  | k  | i  |      D      |
+------------------+----+----+----+-------------+
         8           4    4    4        12
```

Figure 8.1 CYBER 180 jkiD Instruction Format

```
+-------------+---+----+----+----+-----------+
| Operation   | S | j  | k  | i  |     D     |
|   Code      |   |    |    |    |           |
+-------------+---+----+----+----+-----------+
      5         3   4    4    4        12
```

Figure 8.2 - CYBER 180 SjkiD Instructions Format

For these 32-bit instruction formats: the j, k, and i fields provide register designations, the D field provide either a signed shift count, a positive displacement or a bit-string descriptor, and the S field provide a sub-operation code.

```
+-------------------+----+----+
| Operation Code    | j  | k  |
+-------------------+----+----+
         8            4    4
```

Figure 8.3 - CYBER 180 jk Instruction Format

For this 16-bit instruction format, the j field provides a register designation, a sub-operation code, or an immediate operand value and the k field provides a register designation.

```
+-------------------+----+----+---------------+
| Operation Code    | j  | k  |       Q       |
+-------------------+----+----+---------------+
         8            4    4         16
```

Figure 8.4 - CYBER 180 jkQ Instruction Format

For this 32-bit instruction format, the j and k fields provide register designations or sub-operation codes. The 16-bit Q-field provides a signed displacement or an immediate operand value.

7.3 GENERAL CPU INSTRUCTIONS

The CYBER 180 Assembler's CPU Instructions Group is subdivided into the following classes of instructions according to function.

7.3.1 LOAD AND STORE

This sub-group of instructions shall provide the means for transferring data, in the form of a single bit, a byte string, a 64-bit word, or multiple 64-bit words between one or more Registers and one or more locations in central memory as specified by the individual instruction mnemonic.

For the purpose of establishing operand access validity for the associated central memory read and write accesses, the ring number used for validation is the value of the ring number contained in bit positions 16 through 19 of the associated A Register.

The central memory operand access type is read-access for any instruction which loads an A or X register, and write-access for any instruction which stores an A or X register.

Instructions which transfer data from one or more Registers to central memory, (namely, Store instructions), do not alter the contents of any Register which serves as a source of the data to be transferred to central memory.

7.3.1.1 LBYTS,SBYTS-Load/Store Bytes, Xk Length Per S

a) Load Bytes to Xk from (Aj) displaced by D and indexed by (Xi) Right, Length Per S.

LBYTS - (Format = SjkiD Op Code = D0-D7 Ref# = 001)

label	operation	argument
	LBYTS,S	Xk,Aj,Xi,D

S - number of bytes to load(1-8).

b) Store Bytes from Xk at (Aj) displaced by D and indexed by (Xi) Right, Length per S.

SBYTS - (Format = SjkiD Op Code = D8-DF Ref# = 003)

label	operation	argument
	SBYTS,S	Xk,Aj,Xi,D

S - number of bytes to store(1-8).

7.3.1.2 LXI,LX,SXI,SX-Load/Store Word, Xk

a) Load Xk from (Aj) displaced by 8*D and indexed by 8*(Xi) Right.

LXI - (Format = jkiD Op Code = A2 Ref# = 005)

label	operation	argument
	LXI	Xk,Aj,Xi,label

label - byte address, must be on a word boundary.

b) Load Xk from (Aj) displaced by 8*Q.

LX - (Format = jkQ Op Code = 82 Ref# = 006)

label	operation	argument
	LX	Xk,Aj,label

label - byte address, must be on a word boundary.

c) Store Xk at (Aj) displaced by 8*D and indexed by 8*(Xi) Right.

SXI - (Format = jkiD Op Code = A3 Ref# = 007)

label	operation	argument
	SXI	Xk,Aj,Xi,label

label - byte address, must be on a word boundary.

d) Store Xk at (Aj) displaced by 8*Q.

SX - (Format = jkQ Op Code = 83 Ref# = 008)

label	operation	argument
	SX	Xk,Aj,label

label - byte address, must be on a word boundary.

7.3.1.3 LBYT,SBYT-Load/Store Bytes, Xk Length Per X0

a) Load Bytes to Xk from (Aj) displaced by D and indexed by (Xi) Right, Length per X0.

LBYT - (Format = jkiD Op Code = A4 Ref# = 009)

label	operation	argument
	LBYT,X0	Xk,Aj,Xi,D

b) Store Bytes from Xk at (Aj) displaced by D and indexed by (Xi) Right, Length per X0.

SBYT - (Format = jkiD Op Code = A5 Ref# = 011)

label	operation	argument
	SBYT,X0	Xk,Aj,Xi,D

7.3.1.4 LBYTP-Load Bytes, Xk Length per j

a) Load Bytes to Xk from (P) displaced by Q, Length per j.

LBYTP - (Format = jkQ Op Code = 86 Ref# = 013)

label	operation	argument
	LBYTP,j	Xk,label

label - byte address of the data.
j - number of bytes to load(1-8).

7.3.1.5 LBIT,SBIT-Load/Store Bit, Xk

a) Load Bit to Xk (Aj) displaced by Q and bit indexed by (X0) Right.

LBIT - (Format = jkQ Op Code = 88 Ref# = 014)

label	operation	argument
	LBIT	Xk,Aj,Q,X0

b) Store Bit from Xk at (Aj) displaced by Q and bit indexed by (X0) Right.

SBIT - (Format = jkQ Op Code = 89 Ref# = 015)

label	operation	argument
	SBIT	Xk,Aj,Q,X0

7.3.1.6 LAI,LA,SAI,SA-Load/Store,Ak

a) Load Ak from (Aj) displaced by D and indexed by (Xi) Right.

LAI - (Format = jkiD Op Code = A0 Ref# = 016)

label	operation	argument
	LAI	Ak,Aj,Xi,D

b) Load Ak from (Aj) displaced by Q.

LA - (Format = jkQ Op Code = 84 Ref# = 017)

label	operation	argument
	LA	Ak,Aj,Q

c) Store Ak at (Aj) displaced by D and indexed by (Xi) Right.

SAI - (Format = jkiD Op Code = A1 Ref# = 018)

label	operation	argument
	SAI	Ak,Aj,Xi,D

d) Store Ak at (Aj) displaced by Q.

SA - (Format = jkQ Op Code = 85 Ref# = 019)

label	operation	argument
	SA	Ak,Aj,Q

7.3.1.7 LMULT,SMULT-Load/Store Multiple Registers

a) Load Multiple Registers from (Aj) displaced by 8*Q, Selectivity per (Xk) Right.

LMULT - (Format = jkQ Op Code = 80 Ref# = 020)

label	operation	argument
	LMULT	Xk,Aj,label

label - byte address, must be on a word boundary

b) Store Multiple Registers at (Aj) displaced by 8*Q, Selectivity per (Xk) Right.

SMULT - (Format = jkQ Op Code = 81 Ref# = 021)

label	operation	argument
	SMULT	Xk,Aj,label

label - byte address, must be on a word boundary

. ...

7.3.2 INTEGER ARITHMETIC

Integer arithmetic operations shall be performed on words and halfwords contained in Register Xk and Register Xk Right, respectively, as described in the following subparagraphs.

Binary integers contained in the X Registers consist of signed, two's complement, 32-bit or 64-bit quantities. The leftmost bit, (in position 00 for 64-bit integers and in position 32 for 32-bit integers), constitute the sign bit.

The ranges in magnitude, M, covered by binary integers in each of the two fixed point formats, are the following:

32-bit Integer: $-2(31) \leq M \leq 2(31)-1$

64-bit Integer: $-2(63) \leq M \leq 2(63)-1$

7.3.2.1 ADDX,ADDXQ,INCX-Integer Sum, Xk

a) Integer Sum, (Xk) replaced by (Xk) plus (Xj).

ADDX - (Format = jk Op Code = 24 Ref# = 022)

label	operation	argument
	ADDX	Xk,Xj

b) Integer Sum, (Xk) replaced by (Xj) plus Q.

ADDXQ - (Format = jkQ Op Code = 8B Ref# = 143)

label	operation	argument
	ADDXQ	Xk,Xj,Q

c) Integer Sum, (Xk) replaced by (Xk) plus j.

INCX - (Format = jk Op Code = 10 Ref# = 166)

label	operation	argument
	INCX	Xk,j

7.3.2.2 SUBX,DECX-Integer Difference, Xk

a) Integer Difference, (Xk) replaced by (Xk) minus (Xj).

SUBX - (Format = jk Op Code = 25 Ref# = 023)

label	operation	argument
	SUBX	Xk,Xj

b) Integer Difference, (Xk) replaced by (Xk) minus j.

DECX - (Format = jk Op Code = 11 Ref# = 167)

label	operation	argument
	DECX	Xk,j

7.3.2.3 MULX,MULXQ-Integer Product, Xk

a) Integer Product, (Xk) replaced by (Xk) times (Xj).

MULX - (Format = jk Op Code = 26 Ref# = 024)

label	operation	argument
	MULX	Xk,Xj

b) Integer Product, (Xk) replaced by (Xj) times Q.

MULXQ - (Format = jkQ Op Code = B2 Ref# = 168)

label	operation	argument
	MULXQ	Xk,Xj,Q

7.3.2.4 DIVX-Integer Quotient

a) Integer Quotient, (Xk) replaced by (Xk) divided by (Xj).

DIVX - (Format = jk Op Code = 27 Ref# = 025)

label	operation	argument
	DIVX	Xk,Xj

7.3.2.5 ADDR,ADDRQ,INCR-Integer Sum, Xk right

a) Integer Sum, (Xk) Right replaced by (Xk) Right plus (Xj) Right.

ADDR - (Format = jk Op Code = 20 Ref# = 027)

label	operation	argument
	ADDR	Xk,Xj

b) Integer Sum, (Xk) Right replaced by (Xj) Right plus Q.

ADDRQ - (Format = jkQ Op Code = 8A Ref# = 028)

label	operation	argument
	ADDRQ	Xk,Xj,Q

c) Integer Sum, (Xk) Right replaced by (Xk) Right plus j.

INCR - (Format = jk Op Code = 28 Ref# = 029)

label	operation	argument
	INCR	Xk,j

7.3.2.6 SUBR,DECR-Integer Difference, Xk Right

a) Integer Difference, (Xk) Right replaced by (Xk) Right minus (Xj) Right.

SUBR - (Format = jk Op Code = 21 Ref# = 030)

label	operation	argument
	SUBR	Xk,Xj

b) Integer Difference, (Xk) Right replaced by (Xk) Right minus j.

DECR - (Format = jk Op Code = 29 Ref# = 031)

label	operation	argument
	DECR	Xk,j

7.3.2.7 MULR,MULRQ-Integer Product, Xk Right

a) Integer Product, (Xk) Right replaced by (Xk) Right times (Xj) Right.

MULR - (Format = jk Op Code = 22 Ref# = 032)

label	operation	argument
	MULR	Xk,Xj

b) Integer Product, (Xk) Right replaced by (Xj) Right times Q.

MULRQ - (Format = jkQ Op Code = 8C Ref# = 033)

label	operation	argument
	MULRQ	Xk,Xj,Q

7.3.2.8 DIVR-Integer Quotient, Xk Right

a) Integer Quotient, (Xk) Right replaced by (Xk) Right divided by (Xj) Right.

DIVR - (Format = jk Op Code = 23 Ref# = 034)

label	operation	argument
	DIVR	Xk,Xj

7.3.2.9 CMPX,CMPR-Integer Compare

a) Integer Compare (Xj) to (Xk), result to X1 Right.

CMPX - (Format = jk Op Code = 2D Ref# = 035)

label	operation	argument
	CMPX	X1,Xj,Xk

b) Integer Compare (Xj) Right to (Xk) Right, result to X1 Right.

CMPR - (Format = jk Op Code = 2C Ref# = 036)

label	operation	argument
	CMPR	X1,Xj,Xk

. ...

7.3.3 BRANCH

The instructions within this subgroup consist of both conditional and unconditional branch instructions.

Each conditional branch instruction performs a comparison between the contents of two general registers. Then, based on the relationship between the results of that comparison and the branch condition as specified by means of the instruction's operation code, each conditional branch instruction performs either a normal exit or a branch exit.

Normal exit: When the results of a comparison do not satisfy the branch condition as specified by the operation code, a normal exit is performed. A normal exit for all conditional branch instructions consists of adding four to the rightmost 32 bits of the PVA obtained from the P Register, with that 32-bit sum returned to the P Register in its rightmost 32-bit positions.

Branch exit: When the results of a comparison satisfy the branch condition as specified by the operation code, a branch exit is performed. A branch exit consists of expanding the 16-bit Q field from the instruction to 31 bits by means of sign extension, shifting these 31 bits left one bit position with a zero inserted on the right, and adding this 32-bit shifted result to the rightmost 32-bits of the PVA obtained from the P Register, with the 32-bit sum returned to the P Register in its rightmost 32-bit positions.

Unconditional branch instructions perform branch exits according to the appropriate instruction descriptions contained in subparagraphs 2.2.3.5 and 2.2.3.6 of the MIGDS.

The Assembler sets the instruction's Q field according to the value of the 'label' subfield of the instruction mnemonics, which must correspond to a label of an Assembler statement within the currently active section. Relative addresses cannot span section boundaries.

7.3.3.1 BRXEQ,BRXNE,BRXGT,BRXGE-Branch Conditional

a) Branch to (P) displaced by 2*Q if (Xj) equal to (Xk).

BRXEQ - (Format = jkQ Op Code = 94 Ref# = 037)

label	operation	argument
	BRXEQ	Xj,Xk,label

label - byte address of the new location.

b) Branch to (P) displaced by 2*Q if (Xj) not equal to (Xk).

BRXNE - (Format = jkQ Op Code = 95 Ref# = 038)

label	operation	argument
	BRXNE	Xj,Xk,label

label - byte address of the new location.

c) Branch to (P) displaced by 2*Q if (Xj) greater than (Xk).

BRXGT - (Format = jkQ Op Code = 96 Ref# = 039)

label	operation	argument
	BRXGT	Xj,Xk,label

label - byte address of the new location.

d) Branch to (P) displaced by 2*Q if (Xj) greater than or equal to (Xk).

BRXGE - (Format = jkQ Op Code = 97 Ref# = 040)

label	operation	argument
	BRXGE	Xj,Xk,label

label - byte address of the new location.

7.3.3.2 BRREQ,BRRNE,BRRGT,BRRGE-Conditional, X Right

a) Branch to (P) displaced by 2*Q if (Xj) Right equal to (Xk) Right.

BRREQ - (Format = jkQ Op Code = 90 Ref# = 041)

label	operation	argument
	BRREQ	Xj,Xk,label

label - byte address of the new location.

b) Branch to (P) displaced by 2*Q if (Xj) Right not equal to (Xk) Right.

BRRNE - (Format = jkQ Op Code = 91 Ref# = 042)

label	operation	argument
	BRRNE	Xj,Xk,label

label - byte address of the new location.

c) Branch to (P) displaced by 2*Q if (Xj) Right greater than (Xk) Right.

BRRGT - (Format = jkQ Op Code = 92 Ref# = 043)

label	operation	argument
	BRRGT	Xj,Xk,label

label - byte address of the new location.

d) Branch to (P) displaced by 2*Q if (Xj) Right greater than or equal to (Xk) Right.

BRRGE - (Format = jkQ Op Code = 93 Ref# = 044)

label	operation	argument
	BRRGE	Xj,Xk,label

label - byte address of the new location.

7.3.3.3 BRINC-Conditional, with Increment

a) Branch to (P) displaced by 2*Q and increment (Xk) if (Xj) greater than (Xk).

BRINC - (Format = jkQ Op Code = 9C Ref# = 045)

label	operation	argument
	BRINC	Xj,Xk,label

label - byte address of the new location.

7.3.3.4 BRSEG-Conditional, Ak

a) Branch to (P) displaced by 2*Q if SEG(Aj) not equal to SEG(Ak); else Compare BN(Aj) to BN(Ak), result to X1 Right.

BRSEG - (Format = jkQ Op Code = 9D Ref# = 046)

label	operation	argument
	BRSEG	X1,Aj,Ak,label

label - byte address of the new location.

7.3.3.5 BRREL-Unconditional Branch, (P) indexed

a) Branch to (P) indexed by 2*(Xk) Right.

BRREL - (Format = jk Op Code = 2E Ref# = 047)

label	operation	argument
	BRREL	Xk

7.3.3.6 BRDIR-Unconditional Branch, (A) indexed

a) Branch to (Aj) indexed by 2*(Xk) Right.

BRDIR - (Format = jk Op Code = 2F Ref# = 048)

label	operation	argument
	BRDIR	Aj,Xk

. ...

7.3.4 COPY

The instructions within this subgroup provide the means for accomplishing inter-register transfers to the extent defined by the following instruction descriptions.

7.3.4.1 CPYXX-Copy to Xk from Xj

CPYXX - (Format = jk Op Code = 0D Ref# = 049)

label	operation	argument
	CPYXX	Xk,Xj

7.3.4.2 CPYAX-Copy to Xk from Aj

CPYAX - (Format = jk Op Code = 0B Ref# = 050)

label	operation	argument
	CPYAX	Xk,Aj

7.3.4.3 CPYAA-Copy to Ak from Aj

CPYAA - (Format = jk Op Code = 09 Ref# = 051)

label	operation	argument
	CPYAA	Ak,Aj

7.3.4.4 CPYXA-Copy to Ak from Xj

CPYXA - (Format = jk Op Code = 0A Ref# = 052)

label	operation	argument
	CPYXA	Ak,Xj

7.3.4.5 CPYRR-Copy to Xk Right from Xj Right

CPYRR - (Format = jk Op Code = 0C Ref# = 053)

label	operation	argument
	CPYRR	Xk,Xj

. ...

7.3.5 ADDRESS ARITHMETIC

The instructions within this subgroup shall provide the means for accomplishing address arithmetic to the extent defined by the following instruction descriptions.

7.3.5.1 ADDAQ-Copy A with Displacement

a) Address (Ak) replaced by (Aj) plus Q.

ADDAQ - (Format = jkQ Op Code = 8E Ref# = 054)

label	operation	argument
	ADDAQ	Ak,Aj,Q

7.3.5.2 ADDPXQ-Copy P with Indexing and Displacement

a) Address (Ak) replaced by (P) plus 2*(Xj) Right plus 2*Q.

ADDPXQ - (Format = jkQ Op Code = 8F Ref# = 055)

label	operation	argument
	ADDPXQ	Ak,Xj,label

label - byte address of the new location.

7.3.5.3 ADDAX-A Indexed

a) Address (Ak) replaced by (Ak) plus (Xj) Right.

ADDAX - (Format = jk Op Code = 2A Ref# = 056)

label	operation	argument
	ADDAX	Ak,Xj

7.3.5.4 ADDAD-Copy A with Displacement, Modulo

a) Address (Ak) replaced by (Ai) plus D per j.

ADDAD - (Format = jkiD Op Code = A7 Ref# = 161)

label	operation	argument
	ADDAD	Ak,Ai,D,j

. ...

7.3.6 ENTER

The instructions within this subgroup provide the means for entering immediate operands, (consisting of logical quantities of signed, two's complement binary integers), into the X Registers to the extent defined by the following instruction descriptions.

7.3.6.1 ENTP,ENTN-Enter j

a) Enter Xk with plus j.

ENTP - (Format = jk Op Code = 3D Ref# = 057)

label	operation	argument
	ENTP	Xk,j

b) Enter Xk with minus j.

ENTN - (Format = jk Op Code = 3E Ref# = 058)

label	operation	argument
	ENTN	Xk,j

7.3.6.2 ENTE-Enter Q

a) Enter Xk with sign extended Q.

ENTE - (Format = jkQ Op Code = 8D Ref# = 059)

label	operation	argument
	ENTE	Xk,Q

7.3.6.3 ENTL,ENTX-Enter jk

a) Enter X0 with logical jk.

ENTL - (Format = jk Op Code = 3F Ref# = 060)

label	operation	argument
	ENTL	X0,jk

b) Enter X1 with logical jk.

ENTX - (Format = jk Op Code = 39 Ref# = 164)

label	operation	argument
	ENTX	X1,jk

7.3.6.4 ENTZ,ENTQ,ENTS-Enter Signs

a) Enter Xk Left with signs per j.

The value of the right most 2-bits of the j field from the instruction shall be translated as follows:

If 00, 32 bit positions of Xk Left shall be cleared (zeroes).
If 01, 32 bit positions of Xk Left shall be set (ones).
If 10 or 11, 32 bit positions of Xk Left shall be set to the value of the sign bit in position 32 of Xk Right.

ENTZ - (Format = jk Op Code = 1F Ref# = 061)

ENTO - (Format = jk Op Code = 1F Ref# = 061)

ENTS - (Format = jk Op Code = 1F Ref# = 061)

label	operation	argument
	ENTZ ENTO ENTS	Xk

The assembler computes the value of j from the specific instruction mnemonic used.

7.3.6.5 ENTC-Enter X1 jkQ

a) Enter X1 with sign extended jkQ.

ENTC - (Format = jkQ Op Code = 87 Ref# = 165)

label	operation	argument
	ENTC	X1,jkQ

7.3.6.6 ENTA-Enter X0 jkQ

a) Enter X0 with sign extended jkQ.

ENTA - (Format = jkQ Op Code = B3 Ref# = 169)

label	operation	argument
	ENTA	X0,jkQ

. ...

7.3.7 SHIFT

The instructions within this subgroup provided the means for shifting the initial contents of the Xj Register and transferring the result to the Xk Register, to the extent defined by the following descriptions.

All of the instructions within this subgroup derive the computed shift count in the following manner: The rightmost 8 bits of the D field from the instruction is added to the rightmost 8 bits initially contained in bit positions 56 through 63 of Register Xi Right and the 8-bit sum represents the computed shift count. Any overflow from the 8-bit sum is ignored. In this context, the contents of Register X0 Right are interpreted as consisting entirely of zeroes.

The instructions within this subgroup shall interpret the computed shift count as follows: The sign-bit in the leftmost position of the 8-bit computed shift count shall determine the direction of the shift. When the computed shift count is positive (sign bit of zero), these instructions shall left shift. When the computed shift count is negative (sign-bit of one), these instructions shall right shift. For 32-bit quantities, shifts shall be from 0-31 bits left and from 1-32 bits right. For 64-bit quantities, shifts shall be from 0-63 bits left and from 1-64 bits right.

When these interpretations of the computed shift count result in an actual shift count of zero, the associated instructions transfer the initial contents of the Xj Register to the Xk Register and no shifting is performed.

7.3.7.1 SHFC-Shift (Xj) to Xk, Circular

a) Shift (Xj) to Xk Circular, Direction and Count per (Xi) Right plus D.

SHFC - (Format = jkiD Op Code = A8 Ref# = 062)

label	operation	argument
	SHFC	Xk,Xj,Xi,D

7.3.7.2 SHFX,SHFR-Shift (Xj) to Xk, End-Off

a) Shift (Xj) to Xk, Direction and Count per (Xi) Right plus D.

SHFX - (Format = jkiD Op Code = A9 Ref# = 063)

label	operation	argument
	SHFX	Xk,Xj,Xi,D

b) Shift (Xj) Right to Xk Right, Direction and Count per (Xi) Right plus D.

SHFR - (Format = jkiD Op Code = AA Ref# = 064)

label	operation	argument
	SHFR	Xk,Xj,Xi,D

7.3.8 LOGICAL

The instructions within this subgroup shall provide the means for performing Boolean operations on the 64-bit words contained in the X Registers to the extent defined by the following instruction descriptions.

7.3.8.1 IORX,XORX,ANDX-Logical Sum, Diff. and Prod.

a) Logical Sum (Xk) replaced by (Xk) OR (Xj).

IORX - (Format = jk Op Code = 18 Ref# = 065)

label	operation	argument
	IORX	Xk,Xj

b) Logical Difference, (Xk) replaced by (Xk) EOR (Xj).

XORX - (Format = jk Op Code = 19 Ref# = 066)

label	operation	argument
	XORX	Xk,Xj

c) Logical Product, (Xk) replaced by (Xk) AND (Xj).

ANDX - (Format = jk Op Code = 1A Ref# = 067)

label	operation	argument
	ANDX	Xk,Xj

7.3.8.2 NOTX-Logical Complement

a) Logical Complement, (Xk) replaced by (Xj) NOT.

NOTX - (Format = jk Op Code = 1B Ref# = 068)

label	operation	argument
	NOTX	Xk,Xj

7.3.8.3 INHX-Logical Inhibit

a) Logical Inhibit, (Xk) replaced by (Xk) AND (Xj) NOT

INHX - (Format = jk Op Code = 1C Ref# = 069)

label	operation	argument
	INHX	Xk,Xj

7.3.9 REGISTER BIT STRING

The instructions within this subgroup provide the means for addressing a contiguous string (field) of bits, beginning and ending independently with any bit positions within a 64-bit word.

For each of these instructions in this subgroup, the bit string is addressed by means of a 12-bit field referred to as a bit string descriptor. Any field of bits, including the field constituting a bit field descriptor, is numbered from left to right, with the leftmost bit numbered 00. The six-bit subfield in bit positions 00 through 05 of a bit string descriptor designates the beginning, or leftmost, bit position within a 64-bit word. The 6-bit subfield in bit positions 06 through 11 of the bit string descriptor is a length designator that is interpreted as designating one less than the length (in bits) of a bit string within a 64-bit word.

Bit String Descriptor

```
|00                           05|06                        11|
+-------------------------------+----------------------------+
| Leftmost Position Designator  |     Length Designator      |
+-------------------------------+----------------------------+
                                       (Bit Length-1)
```

For all instructions within this subgroup, indexing is carried out as follows: the bit string descriptor obtained from the D field of the instruction is zero-extended on the left to 32 bits and then added, without overflow detection, to the contents of register Xi Right (in this context, the contents of register X0 shall be interpreted as all zeroes); the rightmost 12 bits of the result is then interpreted as a bit string descriptor, in the manner described above.

7.3.9.1 ISOM-Isolate Bit Mask

a) Isolate Bit Mask into Xk per (Xi) Right plus D.

ISOM - (Format = jkiD Op Code = AC Ref# = 070)

label	operation	argument
	ISOM	Xk,Xi,D,j

7.3.9.2 ISOB-Isolate into Xk

a) Isolate into Xk from Xj per (Xi) Right plus D.

ISOB - (Format = jkiD Op Code = AD Ref# = 071)

label	operation	argument
	ISOB	Xk,Xj,Xi,D

7.3.9.3 INSB-Insert into Xk

a) Insert into Xk from Xj per(Xi) Right plus D.

INSB - (Format = jkiD Op Code = AE Ref# = 072)

label	operation	argument
	INSB	Xk,Xj,Xi,D

7.3.10 MARK-MARK TO BOOLEAN

This instruction tests the two bits initially contained in the leftmost two bit positions, 32 and 33, of Register X1 Right according to the 4-bit j field from the instruction. When the value of the two leftmost bits initially contained in Register X1 Right is equal to any of the one or more values specified by the instruction's j field, Register Xk shall be cleared in bit positions 1 through 63 and set in bit position 0. When the value of the two leftmost bits initially contained in Register X1 Right is not equal to any of the one or more values specified by the instruction's j field, Register Xk Right is cleared in all 64 bit positions, 0 through 63.

The values of the j field and the leftmost two bits initially contained in Register X1 Right shall be interpreted with respect to equality (EQ) as follows:

j	Binary Value of Bits 32 and 33 of X1 Right, respectively 00	01	10	11
0000	Unconditional inequality			
0001				EQ
0010			EQ	
0011			EQ	EQ
0100		EQ		
0101		EQ		EQ
0110		EQ	EQ	
0111		EQ	EQ	EQ
1000	EQ			
1001	EQ			EQ
1010	EQ		EQ	
1011	EQ		EQ	EQ
1100	EQ	EQ		
1101	EQ	EQ		EQ
1110	EQ	EQ	EQ	
1111	Unconditional Equality			

```
 ||||        ^              ^              ^               ^
 ++++--------+              |              |               |
  +++-----------------------+              |               |
   ++--------------------------------------+               |
    +------------------------------------------------------+
```

The four individual bits of j can be visualized as individual pointers which are associated, from left to right, with the four possible values (00,01,10,11) of the tested bit-pair (bits 32 and

33 of Register X1 Right). For example, if j = 0101, equality is detected when the value of the tested bit pair is 01 or 11.

a) Set Xk per j and (X1) Right.

MARK - (Format = jk Op Code = 1E Ref# = 145)

label	operation	argument
	MARK	Xk,X1,j

. ...

7.4 BUSINESS DATA PROCESSING INSTRUCTIONS

The general form of execution for this group shall involve the utilization of a first data field in central memory, referred to as the "source", to modify, replace, or compare with a second data field in central memory referred to as the "destination". Both the source and destination fields shall be individually described by means of independently designated Data Descriptors, with respect to the types of representation, sign and zone conventions, lengths and relative locations of the data fields.

The Data Descriptors shall be obtained from central memory at locations immediately following the BDP instruction, as defined by the BDP instruction format and number of descriptors used by the instruction. All descriptors consist of a 32-bit half word, aligned to a parcel (16 bit) boundary in central memory.

7.4.1 GENERAL DESCRIPTION

The instructions of this group utilize the jk and jkiD instruction formats in combination with one or two descriptors in the following combinations:

1) jk and two descriptors.

```
        Operation Code   j   k

     +--------------+---+---+
  P  |      8       | 4 | 4 |
     +--------------+---+---+
```

```
                              Descriptor j

         +----------------------------------------+
  P+2    |                  32                    |
         +----------------------------------------+

                              Descriptor k

         +----------------------------------------+
  P+6    |                  32                    |
         +----------------------------------------+
```

2) jkid and two descriptors.

```
         Operation Code   j   k   i         D

         +--------------+---+---+---+------------+
  P      |      8       | 4 | 4 | 4 |     12     |
         +--------------+---+---+---+------------+

                              Descriptor j

         +----------------------------------------+
  P+4    |                  32                    |
         +----------------------------------------+

                              Descriptor k

         +----------------------------------------+
  P+8    |                  32                    |
         +----------------------------------------+
```

3) jkiD and one descriptor.

```
         Operation Code   j   k   i         D

         +--------------+---+---+---+------------+
  P      |      8       | 4 | 4 | 4 |     12     |
         +--------------+---+---+---+------------+

                            Descriptor j or k

         +----------------------------------------+
  P+4    |                  32                    |
         +----------------------------------------+
```

7.4.1.1 Operation Codes

A total of 18 operation codes shall be utilized by the instructions comprising the BDP Instruction group. For the purpose of this specification, the BDP Instruction group shall be further divided into four subgroups, including "short" instruction names, as follows:

NOTE: For the order of exception sensing for these instructions, as well as all other instructions, refer to the CYBER 180 Processor-Memory Model-Independent GDS.

Subgroup	Short Name
BDP Numeric	Sum Difference Product Quotient Scale Scale Rounded Decimal Compare Numeric
Byte	Compare Compare Collated Scan While Non-Member Translate Move Bytes Edit
Subscript	Calculate Subscript
Immediate Data	Move Immediate Data Compare Immediate Data Add Immediate Data

7.4.2 DATA DESCRIPTORS

The generated Data Descriptor shall be formatted as follows:

```
| F | D | T |        L        |            O             |
+---+---+-----+-----------------+--------------------------+
| 1 | 3 |  4  |        8        |           16             |
+---+---+-----+-----------------+--------------------------+
 00                 32-bit Descriptor
```

When specifying the data descriptor, the D field is not specified. The format for the source descriptor (SD) and the destination descriptor (DD) is the same, and is specified as F,T,L,O.

- F - 1 bit - field specifier for length
- T - 4 bits - data types
- L - 8 bits - optional length field
- O - 16 bits - offset address field

The data descriptor fields may be specified via either of two methods.

1. - The field may consist of four subfields each containing an evaluatable expression.

2. - The field may consist of a single SET or EQU symbol (category 9) which is associated with four values.

Example:

```
        ADDN,A7,X0  AF,X1  0,7,0,16  1,7,0,16   .DESCRIPTOR
                                               .FIELDS ARE F,T,L,O.

DSCRPTR SET         0,7,0,16     .BDP DESCRIPTOR
        ADDN,A7,X0  AF,X1  DSCRPTR  DSCRPTR    .ALTERNATE METHOD
```

7.4.2.1 BDP Descriptor, D Field

The D field is a 3 bit reserved field in bit positions 01, 02 and 03 of the data descriptor. Interpretation of other Data Descriptor fields follow. This field is not specified in the instruction.

7.4.2.2 BDP Operand Type, T Field

The T field shall consist of 4 bits, in bit positions 04 through 07 of the Data Descriptor, and shall describe the type of data representation used in the associated source or destination field. The 16 values of the T field are assigned data type representations as follows:

0 Packed Decimal No Sign

1 Packed Decimal No Sign Leading Slack Digit

2 Packed Decimal Signed

3 Packed Decimal Signed Leading Slack Digit

4 Unpacked Decimal Unsigned

5 Unpacked Decimal Trailing Sign Combined Hollerith

6 Unpacked Decimal Trailing Sign Separate

7 Unpacked Decimal Leading Sign Combined Hollerith

8 Unpacked Decimal Leading Sign Separate

9 Alphanumeric

10 Binary Unsigned

11 Binary Signed

12 Translated Packed Decimal Signed

13 Translated Packed Decimal Signed Leading Slack Digit

14 Translated Binary Unsigned

15 Translated Binary Signed

As determined by the operation code, source and destination field, data types shall be restricted to only those combinations which are defined as valid within the instruction descriptions. The designation of invalid T field combinations within the associated Data Descriptors shall result in the detection of an Instruction Specification error, the instruction's execution shall be inhibited and the corresponding program interruption shall occur. The term "freely compatible" as used in the BDP instruction descriptions, means that any allowable source field data type may be used with any allowable destination field data type.

7.4.2.3 BDP Operand Address, O Field

The PVA corresponding to the leftmost byte of a BDP source or destination field shall be obtained by utilizing the 16 bit O field of the corresponding data descriptor (bit positions 16 through 31) as a byte item count to be added as a sign extended 32 bit offset (2's complement for negative offset) to the byte number (BN) portion of the base PVA contained in Register Aj or Ak respectively.

7.4.2.4 BDP Operand Length, F and L Fields

The length in bytes of a BDP source or destination field shall be obtained according to the value of the 1-bit F field (bit 00) of the corresponding descriptor as follows:

F	Length
0	Obtained from the 8 bit L field (bits 08 through 15) of the corresponding descriptor.
1	Obtained from bits 55-63 of X0 Right for the first descriptor following an instruction, and from bits 55-63 of X1 Right for the second descriptor following an instruction.

Although field lengths as long as 256 bytes are possible, the length of a BDP operand shall be restricted to a smaller value for decimal and binary operations, according to the operand data type. These inclusive limits are the following:

19 bytes for Packed Decimal (types 0 through 3, 12 and 13)

38 bytes for Unpacked Decimal (types 4 through 8)

8 bytes for Binary (types 10, 11, 14, and 15)

When any BDP field length exceeds the specified maximum associated with a given data type, an Instruction Specification error shall be detected, the execution of that instruction shall be inhibited, and the corresponding program interruption shall occur.

If F equals 1, then only the rightmost 9 bits of X0 and X1 will be checked to determine whether or not the field length exceeds the maximum allowed. The other bits of X0 and X1 will not be inspected.

7.4.3 DATA AND SIGN CONVENTIONS

With respect to numeric data and sign conventions, interpretation shall be performed according to Type (T) where applicable, for characters (C), Digits (D) and Signs (S), using hexadecimal notation.

NOTE: Data field examples are illustrated in the CYBER 180 Processor-Memory Model-Independent GDS.

7.4.4 BDP NUMERIC

The instructions in this subgroup shall provide the means for performing arithmetic, shift, conversion and comparison operations for byte fields in central memory consisting of numeric decimal data.

Unless the length and format fields within the Data Descriptors associated with the source and destination fields, conform to the restrictions defined within the following instruction descriptions, the detection of a Length or Type error shall result in an Instruction Specification Error condition, the execution of the associated instruction shall be inhibited and the corresponding program interruption shall occur.

Overflow into or other alteration of the slack digit of destination field types 1 and 3 is not allowed. The result shall be right justified in the destination field. If the decimal result is shorter than the destination field, the destination field shall be zero filled to the left. If the result is longer than the destination field, the result shall be truncated on the left as necessary. Thus, conceptually, these instructions shall process the data fields from right to left.

Note that these conventions shall cover the end cases for numeric operands of length equal to 1 for all numeric data types. For instance, a Move Numeric from a type 5 operand to a type 3 or type 6 operand of length 1 would amount to an extraction of the source field sign.

A source BDP operand of numeric type (0 through 8 and 12 through 15) and a length zero, shall be interpreted as the value zero.

A destination BDP operand of length zero shall transform the associated instruction into a no-op. However, exception sensing for the source field shall occur normally, including the testing for an Arithmetic Loss of Significance or Arithmetic Overflow condition, provided the source field does not also have a length of zero.

Minus zero shall be considered equivalent to plus zero by all the instructions in this subgroup, with respect to decimal numeric data.

The representation for zero, zones and signs shall be normally determined by interpreting the T field from the Data Descriptor associated with the destination field.

Division by zero shall not be allowed to the extent that the destination field in central memory shall not be changed and a Divide Fault condition shall be detected.

Each source digit shall be checked for decimal digit validity. An invalid decimal digit shall cause an Invalid BDP Data condition to be detected and, if enabled, a program interruption shall occur upon the completion of these instructions.

7.4.4.1 <u>Arithmetic</u>

a) Decimal Sum, D(Ak) replaced by D(Ak) plus D(Aj).

074 jk (2 descriptors)

b) Decimal Difference, D(Ak) replaced by D(Ak) minus D(Aj).

075 jk (2 descriptors)

c) Decimal Product, D(Ak) replaced by D(Ak) times D(Aj).

076 jk (2 descriptors)

d) Decimal Quotient, D(Ak) replaced by D(Ak) divided by D(Aj).

077 jk (2 descriptors)

Operation: These instructions shall arithmetically modify the initial contents of the destination field in central memory, (treated as an augend, minuend, multiplicand or dividend as determined by the operation code) by the contents of the source field in central memory (treated as an addend, subtrahend, multiplier or divisor as determined by the operation code) and shall transfer the decimal result consisting of a sum, difference, product or quotient, as determined by the operation code, to the destination field in central memory.

Types: All Packed decimal types and all Unpacked decimal types, except for the Leading Sign formats, shall be freely allowed for decimal arithmetic; i.e., types 0 through 6, 12 and 13 shall be compatible for these instructions.

Unpacked Decimal Leading Sign (both conventions) shall not be supported in the decimal arithmetic. A Numeric Move instruction must be generated to format the operands of those types prior to their use in arithmetic operations.

Lengths: The maximum allowable lengths for the source and destination fields shall be determined according to their respective decimal data types.

NOTE: Decimal operands shall be treated as integer values.

When the results of these instructions exceed the capacity of the designated field such that significant digits are not stored into central memory, an Arithmetic Overflow condition shall be detected. When the corresponding user condition mask bit is set and the trap is enabled, instruction execution shall be inhibited and program interruption shall occur.

7.4.4.2 ADDN,SUBN,MULN,DIVN-Arithmetic

a) Decimal Sum, D(Ak) replaced by D(Ak) plus D(Aj).

ADDN - (Format = jk2 Op Code = 70 Ref# = 075)

label	operation	argument
	ADDN,Aj,X0	Ak,X1 SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

b) Decimal Difference, D(Ak) replaced by D(Ak) minus D(Aj).

SUBN - (Format = jk2 Op Code = 71 Ref# = 075)

label	operation	argument
	SUBN,Aj,X0	Ak,X1 SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

c) Decimal product, D(Ak) replaced by D(Ak) times D(Aj).

MULN - (Format = jk2 Op Code = 72 Ref# = 076)

label	operation	argument
	MULN,Aj,X0	Ak,X1 SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

d) Decimal Quotient, D(Ak) replaced by D(Ak) times D(Aj).

DIVN - (Format = jk2 Op Code = 73 Ref# = 077)

label	operation	argument
	DIVN,Aj,X0	Ak,X1 SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

7.4.4.3 SCLN,SCLR-Shift

The following instructions shall move data initially contained in the source field to the destination field, and shall provide shifting of the data under control of a shift count. The shift count shall be derived in the following manner: The rightmost 8 bits from the instruction's D field shall be added to the rightmost 8 bits initially contained in bit positions 56 through 63 of Register Xi Right and the 8-bit sum shall represent the computed shift count. Any overflow from the 8-bit sum is ignored. In this context, the contents of Register X0 shall be interpreted entirely of zeroes. A zero shift count shall cause the instruction to act as a move only instruction.

The 8-bit shift count shall be interpreted as a signed, binary integer. When this 8-bit shift count is positive, the direction of the shift shall be left with the number of decimal digit positions to be shifted determined by the value of the right-most seven bits (bit positions 57-63) of the shift count. When this 8-bit shift count is negative, the direction of the shift shall be right with the number of decimal digit positions to be shifted determined by the value of the 2's complement of the rightmost 7 bits (bit positions 57-63) of the shift count, with minus 128 (1000 0000) being interpreted as zero. Thus, positive shift counts shall provide the means for multiplying the source data field by powers of ten, and negative shift counts shall provide the means for dividing the source data fields by powers of ten, as the source data is moved to the destination field.

When non-zero digits are shifted left end-off, or truncated on the left, an Arithmetic Loss of Significance condition shall be detected. If the corresponding user condition mask bit is set and the trap is enabled, instruction execution shall be inhibited and program interruption shall occur.

Shifting shall be accomplished end-off with zero fill on the appropriate end(s) as required to accommodate the length and type of the receiving field. (For example, when the destination field is longer than the source field, and the difference in field lengths is greater than the left shift count, such a scale instruction shall provide zero fill, to the extent required, on both the right and left ends of the destination field result).

Types: Source field data shall be restricted to Types 0 through 6, 9, 12 and 13, all of which shall be freely compatible with allowable destination field data Types of 0 through 6, 12 and 13.

Lengths: The maximum allowable lengths for the source and destination fields shall be determined according to their respective decimal data types.

a) Decimal Scale, D(Ak) replaced by D(Aj), scaled per (Xi) Right plus D.

SCLN - (Format = jkiD2 Op Code = E4 Ref# = 078)

label	operation	argument
	SCLN,Aj,X0	Ak,X1,Xi,D SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

b) Decimal Scale Rounded, D(Ak) replaced by rounded D(Aj), scaled per (Xi) Right plus D.

SCLR - (Format = jkiD2 Op Code = E5 Ref# = 079)

label	operation	argument
	SCLR,Aj,X0	Ak,X1,Xi,D SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

These instructions shall move and scale the decimal data field initially contained in the source field to the destination field. They shall transfer the sign of the source field to the destination field without change (unless the results consist entirely of zeroes and there is no loss of significance, in which case the sign of the destination field shall be made positive, or unless the result would otherwise contain a non-preferred sign, in which case the sign of the destination field shall contain the preferred sign).

When specified by means of the operation code, rounding shall be performed for negatively signed scale factors by adding five to the last digit shifted end-off and propagating carries, if any, through the decimal result transferred to the destination field. Thus the absolute value shall be rounded upwards.

7.4.4.4 MOVN-Move

a) Numeric Move, D(Ak) replaced by D(Aj), after formatting.

MOVN - (Format = jk2 Op Code = 75 Ref# = 092)

label	operation	argument
	MOVN,Aj,X0	Ak,X1 SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

This instruction formats the number obtained from the source field and transfers the result to the destination field.

The source field validated according to the T field from its associated descriptor; the source field is reformatted according to the T field from the data descriptor associated with the destination field and the result is transferred to the destination field.

7.4.4.5 CMPN-Comparison

a) Decimal Compare, D(Aj) to D(Ak), result to X1 Right.

CMPN - (Format = jk2 Op Code = 74 Ref# = 083)

label	operation	argument
	CMPN,Aj,X0	Ak,X1 SD DD

When the F field of the source descriptor is equal to 0, X0 is not a required parameter.

This instruction algebraically compares the decimal contents of the source field to the decimal contents of the destination field and transfers a 32-bit halfword to Register X1 Right according to the results of the comparison.

When the results of the source and destination fields are equal, the entire 32-bit positions of Register X1 Right are cleared.

When the contents of the source field are greater than the contents of the destination field, Register X1 Right is cleared in bit positions 32 and 34 through 63, and set in bit position 33.

When the contents of the source field are less than the contents of the destination field, Register X1 Right is cleared in bit positions 34 through 63 and set in bit positions 32 and 33.

. ...

7.4.5 BYTE

The instructions in this subgroup provide the means for comparing, scanning, translating, moving and editing byte fields in central memory to the extent defined by the following descriptions.

7.4.5.1 CMPB,CMPC-Comparison

a) Byte Compare, D(Aj) to D(Ak), result to X1 Right, Index to X0 Right.

CMPB - (Format = jk2 Op Code = 77 Ref# = 084)

label	operation	argument
	CMPB,Aj,X0	Ak,X1 SD DD

b) Byte Compare Collated, D(Aj) to D(Ak), both translated per (Ai) plus D, result to X1 Right, Index to X0 Right.

CMPC - (Format = jkiD2 Op Code = E9 Ref# = 085)

label	operation	argument
	CMPC,Aj,X0	Ak,X1,Ai,D SD DD

These instructions compare the bytes contained in the source field to the bytes contained in the destination field and transfer the results to the comparison to Register X1 Right.

The comparison proceeds from left to right. When the field lengths are unequal, trailing space characters are used for the field having the shorter length. The maximum length for each operand is 256 bytes.

The comparison continues until the longer field has been exhausted or until an "inequality" is detected between corresponding bytes from the source and destination fields according to the following definitions. For the compare instruction, inequality between the bytes obtained directly from the source and destination fields results in the completion of the comparison. For the Collated Compare instruction inequality of the bytes obtained directly from the source and destination fields results in the translation of both bytes by means of a translation table, and inequality of the post-translation bytes results in the completion of the comparison. When the translated bytes are equal, and the longer field has not been exhausted, comparison between the corresponding bytes obtained directly from the source and destination fields is resumed.

Each byte shall be translated by using its value as a positive offset to be added to the beginning (leftmost) address of the Translation Table, (Ai) + D, for the purpose of addressing the translated byte to be read from central memory.

7.4.5.2 SCNB-Byte Scan

a) Byte Scan While Non-Member, D(Ak) for presence bit in (Ai)+D, index to X0 Right, character to X1 Right.

SCNB - (Format = jkiD1 Op Code = F3 Ref# = 086)

label	operation	argument
	SCNB,Aj,X0	Ak,X1,Ai,D DD

The Aj field of this instruction is unused and optional. Operation: The operation shall proceed from left to right on the destination field addressed by D(Ak). One character at a time shall be taken from this character string and used as a bit address into the string addressed by a PVA whose Ring Number (RN) and Segment (SEG) are obtained from Ai, and whose Byte Number (BN) is formed by the 32-bit sum (ignoring overflow) of the rightmost 32 bits of Ai plus the instruction's 12-bit D field extended to the left with 20 zeroes. The scan shall terminate if the bit thus addressed in ON or if the destination field has been exhausted; otherwise the next character in D(Ak) is considered.

Source Field: The operand addressed by Ai+D shall be interpreted as a bit string consisting of 256 bits (32 bytes). The entire table, consisting of 256 bits, may be loaded internally to the processor, on a model dependent basis, before any operation on the data is performed.

Destination Field: The type field in D(Ak) shall be ignored. The operand addressed by D(Ak) shall be interpreted as a byte string, and restricted to no more than 256 characters.

The binary value of the sequence number in the string of the byte which caused the scan to terminate shall be placed right justified into X0 Right.

The binary value of the character itself which caused the scan to terminate shall be placed right justified into X1 Right.

If the scan stops by exhaustion of the characters in the byte string, X0 Right shall contain the length of the original byte string and X1 Right shall be set in bit position 32 and cleared in bit positions 33 through 63.

7.4.5.3 TRANB-Translate

a) Byte Translate, D(Ak) replaced by D(Aj), translated per (Ai) plus D.

TRANB - (Format = jkiD2 Op Code = EB Ref# = 088)

label	operation	argument
	TRANB,Aj,X0	Ak,X1,Ai,D SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

This instruction translates each byte contained in the source field according to the translation table in central memory and transfers the results of the byte-by-byte translation to the destination field.

The translation table is addressed in a manner identical to that previously described for the Collated Compare instruction. The type fields in the Data Descriptors associated with the source field and the destination field are ignored. Both operands are restricted to no more than 256 bytes.

The translation operation shall occur from left to right with each source byte used as a positive offset to be added to the beginning (leftmost byte) address of the translation table for the purpose of permitting each byte's translation. Translated bytes, thus obtained from the translation table, shall be transferred to the destination field. The translation operation shall terminate after the destination field length has been exhausted. When the source field length is greater than the destination field length, rightmost bytes from the source field shall be truncated, to the extent required, with respect to the translation operation. When the source field length is less than the destination field length, translated space characters shall be used to fill the rightmost byte positions of the destination field to the extent required.

7.4.5.4 MOVB-Move

a) Move Bytes, D(Ak) replaced by D(Aj).

MOVB - (Format = jk2 Op Code = 76 Ref# = 089)

label	operation	argument
	MOVB,Aj,X0	Ak,X1 SD DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

This instruction provides the means for moving the bytes contained in the source field to the destination field. The type fields of the source and destination data descriptors are ignored. Field lengths are restricted to a maximum of 256 bytes.

7.4.5.5 EDIT-Edit

a) Edit, D(Ak) replaced by D(Aj) edited per M((Ai) + D).

EDIT - (Format = jkiD2 Op Code = ED Ref# = 091)

label	operation	argument
	EDIT,Aj,X0	Ak,X1,Ai,D SD DD

The Aj field is unused and optional. When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

This instruction shall edit the digits or characters contained in the source field according to an edit mask in central memory and shall transfer the result to the destination field. The edit mask shall be addressed by a PVA whose Ring Number (RN) and Segment (SEG) are obtained from Ai, and whose Byte Number (BN) is formed by the 32-bit sum (ignoring overflow) of the rightmost 32 bit of Ai plus the instruction's 12-bit D field extended to the left with 20 zeroes. The edit mask shall consist of a one byte length indication followed by a string of micro-operations. The length indication shall include the byte containing the length.

7.4.6 IMMEDIATE DATA

Within this instruction group, the Immediate Data Byte is an 8 bit field formed by the 2's complement addition of bits 56-63 (Xi) Right and the rightmost 8 bits of the instruction's D field. Overflow is ignored on this summation. In this context, the contents of Register X0 shall be interpreted as consisting entirely of zeroes.

7.4.6.1 MOVI-Move Immed Data (Xi) Right plus D to D(Ak)

MOVI - (Format = jkiD1 Op Code = F9 Ref# = 154)

label	operation	argument
	MOVI,Xi,D	Ak,X1,j DD

When the F field in the data descriptor is equal to 0, the length register (X1 for destination) is not a required parameter.

This instruction shall move the Immediate Data Byte to the destination field after format conversion per the destination field type and the j field sub-operation code. The least significant 2 bits of the j field shall be used as an encoding of the operation to be performed:

a) If = 00, the unsigned (considered positive) numeric value (Type 10) contained in the Immediate Data Byte shall be moved right justified to the receiving field, which must be of type 10, 11, 14 or 15. If necessary, the destination field is filled with zeroes on the left.

b) If = 01, the decimal numeric value (Type 4) contained in the

Immediate Data Byte shall be moved right justified, to the receiving field after possible reformatting to match the data type of the destination. If the format requires a sign, a positive sign shall be supplied. The destination shall be restricted to one of the decimal data types 0 through 6, 12 or 13. This move shall be executed according to the rules of the numeric move for truncation, padding and validation.

Each source digit shall be checked for decimal digit validity. An invalid decimal digit shall cause an Invalid BDP Data condition to be detected. When the corresponding user mask bit is set, and the trap is enabled, instruction execution shall be inhibited and program interruption shall occur.

c) If = 10, the ASCII character contained in the Immediate Data Byte is repeated left to right in the receiving field. The destination data type shall be ignored.

d) If = 11, the ASCII character contained in the Immediate Data Byte is moved left justified into the receiving field, the rest of that field is space filled. The destination data type shall be ignored.

7.4.6.2 CMPI-Compare Immed Data(Xi) Right plus D to D(Ak)

CMPI - (Format = jkiD1 Op Code = FA Ref# = 155)

label	operation	argument
	CMPI,Xi,D	Ak,X1,j DD

This operation shall, depending on the value of the j field, compare the explicit value contained in the Immediate Data Byte to D(Ak) after a possible reformatting to match the data type and shall transfer a 32-bit half word to Register X1 Right according to the result of the comparison.

When the contents of the source and destination fields are equal, the entire 32-bit positions of Register X1 Right shall be cleared.

The rightmost two bits of the j field shall be used as an encoding of the operation to be performed:

a) If J=00, the unsigned (considered positive) numeric value (Type 10) contained in the Immediate Data Byte shall be compared to the contents of field D(Ak), which must be of type 10, 11, 14 or 15. If field D(Ak) is longer than one byte, then the Immediate Data Byte will be zero filled to the left as necessary.

b) If j=01, the decimal numeric value (Type 4) contained in the Immediate Data Byte shall be compared to the contents of field D(Ak) after possible reformatting to match the data type of field D(Ak). If the format requires a sign, a positive sign shall be supplied. The D(Ak) field shall be restricted to one of the decimal data types 0 through 6, 12 or 13. If field D(Ak) is longer than one byte, then the Immediate Data Byte shall be zero filled to the left as necessary.

   Each source digit shall be checked for decimal digit validity. An invalid decimal digit shall cause an Invalid BDP Data condition to be detected. When the corresponding user mask bit is set, and the trap is enabled, instruction execution shall be inhibited and program interruption shall occur.

c) If j=10, the ASCII character contained in the Immediate Data Byte shall be compared left to right with each successive byte contained in the D(Ak) field. The data type of field D(Ak) shall be ignored.

d) If j=11, the ASCII character contained in the Immediate Data Byte shall be compared to the leftmost byte in field D(Ak). If the comparison is equal and if field D(Ak) is longer than one byte, then a space character shall be compared left to right with each successive remaining byte contained in the D(Ak) field. The data type of field D(Ak) shall be ignored.

When the contents of the source field are greater than the contents of the destination field, Register X1 Right shall be cleared in bit positions 32 and 34 through 63 and shall be set in bit position 33.

When the contents of the source field are less than the contents of the destination field, Register X1 Right shall be cleared in bit positions 34 through 63 and shall be set in bit positions 32 and 33.

The interpretation of the source and destination fields are analogous to those described under the Move Immediate Data Instruction.

7.4.6.3 ADDI-Add Immed Data (Xi) Right plus D to D(Ak)

ADDI - (Format = jkiD1 Op Code = FB Ref# = 156)

label	operation	argument
	ADDI,Xi,D	Ak,X1,j DD

When the F field in the data descriptor is equal to 0, the length register (X0 for source, X1 for destination) is not a required parameter.

This operation shall add the explicit integer value contained in the Immediate Data Byte to D(Ak) after a possible conversion to match the destination data type.

Source: The Immediate Data Byte is used to store the integer value of the addend. The j field is used as an encoding of the type of the data contained in the Immediate Data Byte. The least significant bit of the j field is decoded as follows:

a) If = 0, the Immediate Data Byte, contains an unsigned (considered positive) binary integer value; Immediate Data Byte = Data Type 10.

b) If = 1, the Immediate Data Byte, contains one ASCII character representing a decimal digit; if invalid decimal data is encountered in the Immediate Data Byte, an Invalid BDP Data condition shall be detected. When the corresponding user condition mask bit is set and the trap is enabled, instruction execution shall be inhibited and program interruption shall occur. Immediate Data Byte = Data Type 4.

If the source corresponds to case a) above, the destination shall be confined to types 10, 11, 14 and 15.

If the source corresponds to case b) above, the destination shall be confined to types 0 through 6, 12 and 13.

. ...

7.5 FLOATING POINT INSTRUCTIONS

7.5.1 GENERAL DESCRIPTION

A floating point number consists of a signed exponent and a signed fraction. The signed exponent can also be referred to as the characteristic and the signed fraction can also be referred to as the coefficient.

The quantity expressed by a floating point number is of the form (f)2x where f represents the signed fraction and x represents the signed exponent of the base 2.

The exponent base of 2 is an implied constant for all floating point numbers and thus does not explicitly appear in any floating point format.

7.5.2 FORMATS

Floating point data occupies one of two fixed length formats; 64-bit word (Single Precision) or 128-bit doubleword (Double Precision).

In both the single and double precision formats, the leftmost bit position, 00, is occupied by the sign of the fraction. The fifteen bit positions immediately to the right of bit 00, 01 through 15, occupied by the signed exponent.

The field immediately to the right of the signed exponent is occupied by the fraction which in single precision format consists of 48 bits and in double precision format consists of 96 bits, according to the following figures.

```
|00|01           15|16                             63|
+--+---------------+---------------------------------+
|S |Signed Exponent |          48-bit fraction        |
+--+---------------+---------------------------------+
```

Single Precision Floating Point Number

```
|00|01          15|16                              63|
+--+--------------+----------------------------------+
|S |Signed Exponent |Leftmost 48-bits of the fraction |
+--+--------------+----------------------------------+

|64|65          71|72                              127
+--+--------------+----------------------------------+
|S |Signed Exponent |Rightmost 48-bits of fraction    |
+--+--------------+----------------------------------+
```

Double Precision Floating Point Number

A double precision floating point number consists of two single precision floating point numbers located in consecutively numbered X Registers. The two single precision floating point numbers comprising a double precision floating point number are referred to as the leftmost and rightmost parts as contained in the Xn and Xn+1, respectively. The leftmost part may be any single precision floating point number and when it is normalized, (the leftmost bit of the fraction, in bit position 16, is equal to a one) the double precision floating point number is considered to be normalized. The sign of the fraction and the characteristic of the leftmost part constitutes the sign of the fraction and the characteristic of the double precision number.

The fraction field of the leftmost part constitutes the leftmost 48 bits of the 96-bit double precision fraction. The fraction field of the rightmost part constitutes the rightmost 48 bits of the 96-bit double precision fraction. The sign of the fraction and the characteristic of the rightmost part cannot be utilized from any number constituting an input operand (argument) to a double precision floating point operation. Such operations assume that the sign of the fraction of the rightmost part is the same as the sign of the fraction of the leftmost part and that the characteristic of the rightmost part is 48 less than the characteristic of the leftmost part. However, the formation of a double precision floating point result includes making the sign of the fraction of the rightmost part the same as that of the leftmost part and, except for certain cases involving non-standard forms of floating point results, also includes making the characteristic of the rightmost part 48 less than the characteristic of the leftmost part.

The following table illustrates hexadecimal exponent codes for corresponding non-standard as well as standard floating point numbers:

7.0 CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTIONS
7.5.2 FORMATS

```
              +------------------------------------------------
              |Hexadecimal Exponent including coefficient sign
              |    +------------------------------------------
              |    |Actual Exponent (to the base of 2)
              |    |           +------------------------------
              |    |           |Input Arguments
              |    |           |          +-------------------
              |    |           |          |Results
+-------------+----+-----------+----------+-------------------------+
|      ^      |7XXX|   -----   |Indefinite|7000.0--->0              |
|      |      +----+-----------+----------+-------------------------+
|      |      |6FFF|2**12287   |          |Overflow Mask = 0 :      |
|      |      | ^  |     ^     |Infinite  |5000.00--->00            |
|      |      | |  |     |     |          |Overflow Mask = 1 :      |
| Coefficient |5000|2**4096    |          |As Shown                 |
| Sign Equal  +----+-----------+----------+-------------------------+
|     to 0    |4FFF|2**4095    |          |                         |
| (Positive   | ^  |     ^     |          |                         |
|    numbers) | |  |     |     |          |                         |
|      |      |4000|2**0       |Standard  |As Shown                 |
|      |      |3FFF|2**(-1)    |          |                         |
|      |      | |  |     |     |          |                         |
|      |      | v  |     v     |          |                         |
|      |      |3000|2**(-4096) |          |                         |
|      |      +----+-----------+----------+-------------------------+
|      |      |2FFF|2**(-4097) |          |Underflow Mask = 0 :     |
|      |      | |  |     |     |Zero      |000.00--->00             |
|      |      | v  |     v     |          |Underflow Mask = 1 :     |
|      |      |1000|2**(-12288)|          |As Shown                 |
|      |      +----+-----------+----------+-------------------------+
|      v      |0XXX|           |          |                         |
|-------------+----+   -----   |Zero      |Not Applicable           |
|      ^      |8XXX|           |          |                         |
|      |      +----+-----------+----------+-------------------------+
|      |      |9000|2**(-12288|          |Underflow Mask = 0 :     |
|      |      | ^  |     ^     |          |0000.00--->00            |
|      |      | |  |     |     |Zero      |Underflow Mask = 1 :     |
| Coefficient |AFFF|2**(-4097) |          |As Shown                 |
| Sign Equal  +----+-----------+----------+-------------------------+
|     to 1    |B000|2**(-4096) |          |                         |
| (Negative   | ^  |     ^     |          |                         |
|    Numbers) | |  |     |     |          |                         |
|      |      |BFFF|2**(-1)    |Standard  |As Shown                 |
|      |      |C000|2**0       |          |                         |
|      |      | |  |     |     |          |                         |
|      |      | v  |     v     |          |                         |
|      |      |CFFF|2**4095    |          |                         |
|      |      +----+-----------+----------+-------------------------+
|      |      |D000|2**4096    |          |Overflow Mask = 0 :      |
|      |      |    |           |          |                         |
```

```
|          |    | |   |          |Infinite  |D000.00--->00          |
|          |    | v   |     v    |          |Overflow Mask = 1 :    |
|          |    |EFFF |2**12287  |          |As Shown               |
|          |    +-----+----------+----------+-----------------------+
|    v     |    |FXXX |  -----   |Indefinite|7000.00--->00          |
+----------+----+-----+----------+----------+-----------------------+
```

Floating Point Representation

7.5.3 EXPONENT ARITHMETIC

When the exponent fields from input arguments are added, as for floating point multiplication, or subtracted, as for floating point division, the exponent arithmetic is performed algebraically in 2's complement mode. Moreover, such operations take place, conceptually, as if the bias were removed from each exponent field prior to performing the addition or subtraction and then restored following exponent arithmetic so as to correctly bias the exponent result.

Exponent Underflow and Overflow conditions are detected for all single precision, but only for the leftmost part of double precision floating point results. When the generation of the exponent of the rightmost part, by reducing the exponent of the leftmost part by 48, results in underflow for the rightmost part, this underflow is not to be detected and utilization of an Out of Range exponent permits the rightmost part of the double precision floating point number to correctly express its value.

7.5.4 NORMALIZATION

A normalized floating point number has a one in the leftmost bit position, 16, of the fraction field. If the leftmost bit of the fraction is a zero, the number is considered unnormalized. Normalization takes place when intermediate results are changed to final results. Numbers with zero fractions cannot be normalized and such fractions remain equal to zero.

For intermediate results in which coefficient overflow has not occurred and the initial operands were normalized, the normalization process consists of left shifting the fraction until the leftmost bit position contains a one and correspondingly reducing the characteristics by the number of positions shifted. For intermediate results in which coefficient overflow has occurred, the normalization process consists of right shifting the fraction one bit position and correspondingly

increasing the characteristic by one. For double precision floating point numbers, the entire fraction participates in the normalization such that the rightmost part may or may not appear as a normalized single precision number as determined by the value of the fraction.

For quotient and product instructions (reference numbers 103, 104, 107, 108) if the operands are unnormalized, the results may be unnormalized.

When exponent arithmetic operations on standard floating numbers generate an intermediate exponent which is Out of Range, but normalization requirements generate an adjusted exponent which is no longer Out of Range, then neither Exponent Overflow nor Exponent Underflow is recorded for the final results.

7.5.5 DOUBLE PRECISION REGISTER DESIGNATORS

The terms "Xk+1" and "Xj+1" is used to designate an X Register associated with the rightmost part of a double precision floating point number. When the leftmost part of a double precision floating point number, as designated by the terms "Xk" and"Xj" is associated with Register XF (in hexadecimal notation) the terms "Xk+1" and "Xj+1" are interpreted as designating Register X0.

7.5.6 CONVERSION

The instructions within this subgroup provide the means for converting 64-bit words, contained in the X Registers, between floating point and integer formats.

7.5.6.1 CNIF-Convert From Integer to Floating Point

a) Floating Point Convert from Integer, Floating Point (Xk) formed from Integer (Xj).

CNIF - (Format = jk Op Code = 3A Ref# = 097)

label	operation	argument
	CNIF	Xk,Xj

7.5.6.2 CNFI-Convert Floating Point to Integer

a) Floating Point Convert to Integer, Integer (Xk) formed from Floating Point (Xj).

CNFI - (Format = jk Op Code = 3B Ref# = 098)

label	operation	argument
	CNFI	Xk,Xj

7.5.7 ARITHMETIC

The instructions within this subgroup provide the means for performing arithmetic operations on floating point numbers to the extent described in the following subparagraphs.

7.5.7.1 ADDF,SUBF-Add/Subtract, Xk

a) Floating Point Sum, (Xk) replaced by (Xk) plus (Xj).

ADDF - (Format = jk Op Code = 30 Ref# = 099)

label	operation	argument
	ADDF	Xk,Xj

b) Floating Point Difference, (Xk) replaced by (Xk) minus (Xj).

SUBF - (Format = jk Op Code = 31 Ref# = 100)

label	operation	argument
	SUBF	Xk,Xj

7.5.7.2 MULF-Product to XK

a) Floating Point Product, (Xk) replaced by (Xk) times (Xj).

MULF - (Format = jk Op Code = 32 Ref# = 103)

label	operation	argument
	MULF	Xk,Xj

7.5.7.3 DIVF-Quotient to XK

a) Floating Point Quotient, (Xk) replaced by (Xk) divided by (Xj).

DIVF - (Format = jk Op Code = 33 Ref# = 104)

label	operation	argument
	DIVF	Xk,Xj

7.5.7.4 ADDD,SUBD-Add/Subtract, Xk and Xk+1

a) Floating Point DP Sum (Xk, Xk+1) replaced by (Xk, Xk+1) plus (Xj, Xj+1).

ADDD - (Format = jk Op Code = 34 Ref# = 105)

label	operation	argument
	ADDD	Xk,Xj

b) Floating Point DP Difference (Xk, Xk+1) replaced by (Xk, Xk+1) minus (Xj, Xj+1).

SUBD - (Format = jk Op Code = 35 Ref# = 106)

label	operation	argument
	SUBD	Xk,Xj

7.5.7.5 MULD-Product to Xk and Xk+1

a) Floating Point DP Product (Xk, Xk+1) replaced by (Xk, Xk+1) times (Xj, Xj+1).

MULD - (Format = jk Op Code = 36 Ref# = 107)

label	operation	argument
	MULD	Xk,Xj

7.5.7.6 DIVD-Quotient to Xk and Xk+1

a) Floating Point DP Quotient, (Xk, Xk+1) replaced by (Xk, Xk+1) divided by (Xj, Xj+1).

DIVD - (Format = jk Op Code = 37 Ref# = 108)

label	operation	argument
	DIVD	Xk,Xj

7.5.8 BRANCH

The instructions in this subgroup consist of conditional branch instructions.

Each of these conditional branch instructions perform a comparison between two floating point numbers. Then, based on the relationship between the results of that comparison and the branch condition as specified by means of the instruction's operation code, each conditional branch instruction performs either a normal exit or a branch exit.

Normal Exit: When the results of a comparison do not satisfy the branch condition as specified by the operation code, a normal exit is performed. A normal exit for all conditional branch instructions consist of adding four to the rightmost 32 bits of the PVA obtained from the P Register with that 32-bit sum returned to the P Register in its rightmost 32-bit positions.

Branch Exit: When the results of a comparison satisfy the branch condition as specified by the operation code, a branch exit is performed. A branch exit consists of expanding the 16-bit Q field from the instruction to 31 bits by means of sign extension, shifting these 31 bits left one bit position with a zero inserted on the right and adding this 32-bit shifted result to the rightmost 32-bits of the PVA obtained from the P Register with the 32-bit sum returned to the P Register in its rightmost 32-bit positions.

The Assembler sets the instruction's Q field according to the value of the 'label' subfield of the instruction mnemonics, which must correspond to a label of an Assembler statement within the currently active section. Relative addresses cannot span section boundaries.

7.5.8.1 BRFEQ,BRFNE,BRFGT,BRFGE-Compare and Branch

a) Branch to (P) displaced by 2*Q if Floating Point (Xj) equal to (Xk).

BRFEQ - (Format = jkQ Op Code = 98 Ref# = 109)

label	operation	argument
	BRFEQ	Xj,Xk,label

label - byte address of the new location.

b) Branch to (P) displaced by 2*Q if Floating Point (Xj) not equal to (Xk).

BRFNE - (Format = jkQ Op Code = 99 Ref# = 110)

label	operation	argument
	BRFNE	Xj,Xk,label

label - byte address of the new location.

c) Branch to(P) displaced by 2*Q if Floating Point (Xj) greater than (Xk).

BRFGT - (Format = jkQ Op Code = 9A Ref# = 111)

label	operation	argument
	BRFGT	Xj,Xk,label

label - byte address of the new location.

d) Branch to (P) displaced by 2*Q if Floating Point (Xj) greater than or equal to (Xk).

BRFGE - (Format = jkQ Op Code = 9B Ref# = 112)

label	operation	argument
	BRFGE	Xj,Xk,label

label - byte address of the new location.

7.5.8.2 BROVR,BRUND,BRINF-Exception Branch

a) Branch to (P) displaced by 2*Q if Floating Point Exception per j contained in Xk.

The values of the rightmost 2 bits of the j field from the instruction are associated with exception conditions as follows:

if 00, Exponent Overflow
if 01, Exponent Underflow
if 10 or 11, Indefinite

BROVR - (Format = jkQ Op Code = 9E Ref# = 113)

BRUND - (Format = jkQ Op Code = 9E Ref# = 113)

BRINF - (Format = jkQ Op Code = 9E Ref# = 113)

label	operation	argument
	BROVR BRUND BRINF	Xk,label

label - byte address of the new location.

The Assembler computes the value of j from the specific instruction mnemonic used.

7.5.8.3 CMPF-Compare

a) Compare Floating Point (Xj) to (Xk), result to X1 Right.

CMPF - (Format = jk Op Code = 3C Ref# = 114)

label	operation	argument
	CMPF	X1,Xj,Xk

7.6 SYSTEM INSTRUCTIONS

7.6.1 NON-PRIVILEGED MODE

This class of instructions is permitted to execute in any processor mode.

7.6.1.1 EXECUTE, HALT, SYNC

a) Execute Algorithm - Processor Model Dependent Instruction.

EXECUTE - (Format = SjkiD Op Code = C0-C7 Ref# = 139)

label	operation	argument
	EXECUTE,S	j,k,i,D

b) Program Error.

HALT - (Format = jk Op Code = 00 Ref# = 121)

label	operation	argument
	HALT	jk

c) Synchronization - Scope Loop Sync.

SYNC - (Format = jk Op Code = 01 Ref# = 194)

label	operation	argument
	SYNC	jk

7.6.1.2 CALLSEG,CALLREL-Call

These instructions save the "environment",as designated by the contents of Register X0 Right, in the stack frame save area pointed to by the Dynamic Space Pointer initially contained in Register A0. The stack associated with the current ring of execution, as determined by the RN field initially contained in the P Register, "pushed" by transferring the Dynamic Space Pointer, modified in its rightmost 32-bit positions by the addition of 8 times the number of words stored into the stack frame save area, to the appropriate Top of Stack entry in the executing process's Exchange Package.

The A0, A1, and A2 Registers altered to reflect changes with respect to the Current and Previous Stack Frames and the A3, and A4 Registers shall be altered to reflect pertinent parameter changes as required, in accomplishing this transfer of control from a "calling" procedure to a "called" procedure.

Register assignments are as follows:

(A0)- Dynamic Space Pointer
(A1)- Current Stack Frame Pointer
(A2)- Previous Save Area Pointer
(A3)- Binding Section Pointer
(A4)- Argument Pointer

(X0) RIGHT - the Save Environment is defined as follows:

Bits 52-55: Xs = Starting X-Reg to save

Bits 56-59: At = Final A-Reg to save

Bits 60-64: Xt = Final X-Reg to save

a) Call per (Aj) displaced by 8*Q, Arguments per (Ak).

The PVA obtained from Register Aj is modified in its rightmost 32-bit positions by the addition of the zero-extended Q field from the instruction, (shifted left 3-bit positions with zeroes inserted on the right), and the resulting PVA is used to address a Code Base Pointer from a Binding Section Segment. This Code Base Pointer is translated into a PVA used to address the first instruction to be executed in the "called" procedure. The ring of execution of the called procedure, P(RN) final, shall be used to obtain a Top of Stack pointer from the process' Exchange Package to be used as the new Current Stack Frame Pointer.

CALLSEG - (Format = jkQ Op Code = B5 Ref# = 115)

label	operation	argument
	CALLSEG	label,Aj,Ak

label - byte address of entry point in the new procedure, must be on a word boundary.

b) Call to (P) displaced by 8*Q, Binding Section Pointer per (Aj), Arguments per (Ak).

The P Register shall be modified in its rightmost 32-bit positions by the sign extended Q field from the instruction, (left shifted 3-bit positions with zeroes inserted on the right) and the final contents of the P Register shall be made zeroes in the least significant three bit positions (61-63) and shall be used to address the first instruction to be executed in the "called" procedure.

CALLREL - (Format = jkQ Op Code = B0 Ref# = 116)

label	operation	argument
	CALLREL	label,Aj,Ak

label - byte address of the location to continue execution, must be on a word boundary.

The Assembler computes the value of Q from the "label" field of the instruction mnemonics, which must correspond to a label of an Assembler statement within the currently active section. Relative addresses cannot span section boundaries. The address represented by the label must be on a word boundary. This can be insured by using the ALIGN pseudo instruction.

7.6.1.3 RETURN

This instruction re-establishes the Stack Frame and "environment" of a previous procedure as defined by the Previous Save Area Pointer.

The j and k fields from this instruction are not translated by the hardware. Th values have no effect on the execution of this instruction for which all execution parameters are implicit.

The Stack Frame Save Area from which a previous procedure's "environment" is obtained, is addressed by means of the PVA initially contained in Register A2.

The RETURN instruction may also require global privilege. Consult the MIGDS for further information.

RETURN - (Format = jk Op Code = 04 Ref# = 117)

label	operation	argument
	RETURN	jk

7.6.1.4 POP

This instruction re-establishes the Stack Frame of a previous procedure as defined by the Previous Stack Frame's Save Area.

The j and k fields from this instruction are not translated by the hardware. Th values have no effect on the execution of this instruction for which all execution parameters are implicit.

The Stack Frame Save Area from which a previous procedure's Stack Frame pointers is obtained, is addressed by means of the PVA initially contained in Register A2.

POP - (Format = jk Op Code = 06 Ref# = 118)

label	operation	argument
	POP	jk

7.6.1.5 EXCHANGE

When executed in Monitor mode, this instruction shall change the processor from monitor process state to job process state.

When executed in Job mode this instruction changes the processor from job process state to monitor process state. In addition, the System Call bit in position 10 of the Monitor Condition Register, job process state, is set.

The PVA contained in Word 0 (P Register) of the Exchange Package associated with the state from which the exchange is taking place, is updated such that it points to the instruction which would have been executed had the exchange not taken place, i.e., the PVA of the "Exchange" instruction with 2 added to its BN field.

The j and k fields from this instruction are not translated and their values have no effect on the execution of this instruction.

EXCHANGE - (Format = jk Op Code = 02 Ref# = 120)

label	operation	argument
	EXCHANGE	jk

7.6.1.6 KEYPOINT

The Keypoint Instruction allows performance monitoring of programs via the optional Performance Monitoring Facility or via Trap Interrupts. The Keypoint Instruction shall test bit j of the Keypoint Mask Register. The j field, termed the Keypoint Class Number (KCN), shall be used as a bit index into the Keypoint Mask Register. Thus, a KCN or j field of value 4 tests the fifth bit from the left in the Keypoint Mask Register (KMR).

a) Keypoint, class j, code equal to (XK) Right plus Q.

KEYPOINT - (Format = jkQ Op Code = B1 Ref# = 136)

label	operation	argument
	KEYPOINT	j,Xk,Q

7.6.1.7 CMPXA-Compare Swap

a) Compare (Xk) at (Aj); if not equal, load Xk from (Aj); if equal store (X0) at (Aj); however, if (Aj) locked, branch to P plus 2*Q.

CMPXA - (Format = jkQ Op Code = B4 Ref# = 125)

label	operation	argument
	CMPXA	Xk,Aj,X0,label

label - byte address of the new location, must be in the same section.

A serialization function is performed before this instruction begins and again at its end. Execution of this instruction is delayed until all previous accesses to central memory on the part of this processor are completed. Execution of subsequent instructions is delayed until all central memory accesses due to this instruction are completed.

Conceptually, the execution of this "Compare" instruction on the part of a processor results in preventing other processors from accessing any part of the central memory word at the PVA contained in Register Aj between the read and write accesses associated with the execution of this instruction, provided such processors are also executing a "Compare" instruction. With respect to this instruction only, in order to satisfy its "non-preemptive" requirement, the use of 64-bit words consisting entirely of ones in their leftmost 32-bit positions, 00 through 31, is reserved for each processor's implementation of this instruction.

7.6.1.8 LBSET-Load Bit

a) Load Bit to Xk Right from (Aj) bit indexed by (X0) Right and set bit in central memory.

LBSET - (Format = jk Op Code = 14 Ref# = 124)

label	operation	argument
	LBSET	Xk,Aj,X0

This instruction transfers a single bit into Register Xk Right, bit position 63, from a bit position in central memory. This instruction also clears the Xk Register in its leftmost 63 bit positions, 00 through 62. The bit position in central memory is unconditionally set without changing any other bit positions within the byte or word.

No other accesses from any port shall be permitted access to the byte in central memory from the beginning of the read access until the end of the write access which sets the bit within that byte.

A serialization function is performed before this instruction begins and again at its ending. Execution of this instruction is delayed until all previous accesses to central memory by this processor are completed. Execution of subsequent instructions by this processor is delayed until all central memory accesses from this instruction are completed.

7.6.1.9 TPAGE-Test Page

a) Test Page (Aj) and Set Xk Right.

TPAGE - (Format = jk Op Code = 16 Ref# = 126)

label	operation	argument
	TPAGE	Xk,Aj

This instruction shall test for the presence of the page in central memory corresponding to the PVA contained in Register Aj. When this instruction finds the corresponding page in central memory, the "Used" bit in the UM field of the associated Page Descriptor is set, and the Real Memory Address (RMA) translated from the PVA contained in Register Aj is transferred to Register Xk Right. When this instruction cannot find the corresponding page in central memory, Register Xk Right is set in bit position 32 and cleared in bit positions 33 through 63.

7.6.1.10 CPYTX-Copy Free Running Counter(TIME) to X

a) Copy to Xk from Central Memory Maintenance Register at (Xj) Right.

CPYTX - (Format = jk Op Code = 08 Ref# = 132)

label	operation	argument
	CPYTX	Xk,Xj

This instruction shall copy the central memory Maintenance Register specified by the contents of Register Xj into the Xk Register. All 64 bits of the Xk Register shall be cleared before the selected register is copied into it.

7.6.1.11 PSFSA-Purge SFSA Pushdown

PSFSA - (Format = jk Op Code = 07 Ref# = 203)

label	operation	argument
	PSFSA	k

This instruction, when the sub-op k=1, shall cause processors having Stack Frame Save Area (SFSA) Pushdown to store the contents of the Pushdown into its properly defined locations in central memory. For the purposes of the this store the processor shall ignore the state of the valid bit in the Page Table Entry. If a Page Table Search without Find is encountered, the processor shall record a DUE and take the appropriate interrupt. This instruction, when the sub-op k is not 1, shall be executed as a no-op.

For all processors other than Cyber 2000, this instruction shall be executed as a no-op.

7.6.2 LOCAL PRIVILEGED MODE

This class of instructions shall be permitted to execute only from segments having either local privileged mode or global privileged mode. If an instruction in the local privileged mode class attempts execution from a segment having neither local nor global privileges, a Privileged Instruction Fault shall be detected, execution of that instruction shall be inhibited, and the corresponding program interruption shall occur.

Instructions in the local privileged mode class are executable whenever a processor is executing instructions from a segment whose Segment Descriptor defines that segment as either a local privileged executable segment or a global privileged executable segment.

7.6.2.1 LPAGE-Load Page Table Index

a) Load Page Table Index per (Xj) to Xk Right and Set X1 Right.

LPAGE - (Format = jk Op Code = 17 Ref# = 127)

label	operation	argument
	LPAGE	Xk,Xj,X1

This local privileged instruction searches the Page Table in central memory, returns the final index value to Register Xk Right, and sets Register X1 Right according to the results of the search.

The entry searched for within the Page Table is defined by the System Virtual Address (SVA) contained in Register Xj.

The number of entries searched shall always be transferred to Register X1 Right, bits 33-63, right-justified with zeroes extended.

When a Page Descriptor corresponding to the SVA initially contained in Register Xj is found, the index into the Page Table which is associated with that entry shall be transferred right-justified and zero-extended to Register Xk Right, and bit 32 of Register X1 Right shall be set.

When the Page Table search terminates as a result of not finding a Page Descriptor which corresponds to the SVA initially contained in Register Xj (whether the termination results from a Continue bit equal to 0 or performing a maximum of 32 comparisons), the index into the Page Table associated with the last entry compared shall be transferred into Register Xk Right and bit 32 of Register X1 Right shall be cleared.

7.6.3 GLOBAL PRIVILEGED MODE

This class of instructions shall be permitted to execute only from segments having global privileged mode. If an instruction in the global privileged mode class attempts execution from a segment not having global privileges, a Privileged Instruction Fault shall be detected, execution of that instruction shall be inhibited, and the corresponding program interruption shall occur.

Global privileged mode exists whenever the processor is executing instructions from a segment whose Segment Descriptor defines that segment as a global privileged executable segment.

7.6.3.1 INTRUPT-Interrupt Processor

a) Interrupt Processor per (Xk).

INTRUPT - (Format = jk Op Code = 03 Ref# = 122)

label	operation	argument
	INTRUPT	Xk,j

The execution of this global privileged class instruction sends an external interrupt to one or more processors via their central memory ports. The processors are identified by the central memory port number to which they are connected.

The interrupting processor sends the contents of Register Xk to central memory. Central memory then sends an external interrupt to the processor(s) on those ports whose port numbers correspond to the bit positions which are set within Register Xk. When the interrupting processor has two ports connected to the same memory, a "Switch" selects the port used to transmit the contents of Register Xk to central memory along with the "interrupt" function.

When the interrupting processor has two ports connected to independent memories, the state of Bit 33 of Register Xk selects the port used to transmit the contents of Register Xk to central memory along with the "interrupt" function. When Bit 33 is clear, Port 0 is used; when Bit 33 is set, Port 1 is used.

A serialization function is performed before this instruction begins execution. That is, execution of this instruction is delayed until all previous central memory accesses on the part of the interrupting processor are complete.

7.6.4 MIXED MODE

This class of instructions includes those instructions whose mode is dependent on a parameter selection within the instruction. Depending on the value of the parameter, the mode of the instruction is non-privileged, local privileged, global privileged, or monitor. The description of each instruction defines which parameter selects the mode and how the selection is made.

7.6.4.1 BRCR-Branch and Alter Condition Register

a) Branch to (P) displaced by 2*Q and alter Condition Register per jk.

BRCR - (Format = jkQ Op Code = 9F Ref# = 134)

label	operation	argument
	BRCR	j,k,label

label - byte address of the new location.

This instruction tests the value of a selected bit in the Condition Register. The j field selects the bit number within the Monitor Condition Register or within the User Condition Register depending on the k field. The k field shall also determine the branch decision and Condition Register bit alteration as follows:

k = 0 or 8, if bit j of the Monitor Condition Register is set, clear it and take a branch exit.

k = 1 or 9, if bit j of the Monitor Condition Register is not set, set it and take a branch exit.

k = 2 or A, if bit j of the Monitor Condition Register is set, take a branch exit.

k = 3 or B, if bit j of the Monitor Condition Register is not set, take a branch exit.

k = 4 or C, if bit j of the User Condition Register is set, clear it and take a branch exit.

k = 5 or D, if bit j of the User Condition Register is not set, set it and take a branch exit.

k = 6 or E, if bit j of the User Condition Register is set, take a branch exit.

k = 7 or F, if bit j of the User Condition Register is not set, take a branch exit.

Monitor and Privileged Mode - Some values of the k field of this instruction shall cause this instruction to be a Monitor or Non-privileged instruction as follows:

k	Mode
0 or 8	Monitor
1 or 9	Monitor
2 or A	Non-privileged
3 or B	Non-privileged
4 or C	Non-privileged
5 or D	Non-privileged
6 or E	Non-privileged
7 or F	Non-privileged

7.6.4.2 CPYSX,CPYXS-Copy State Registers

These instructions provide the means for copying certain state registers to and from X Registers. The state register is addressed by means of the rightmost 8-bits initially contained in Register Xj Right.

The address assignments are defined in Table 2.6-1 and the restrictions in Table 2.6-2 of the MIGDS.

a) Copy to Xk per (Xj).

CPYSX - (Format = jk Op Code = 0E Ref# = 130)

label	operation	argument
	CPYSX	Xk,Xj

This instruction copies the contents of the state register addressed by the contents of Register Xj into Register Xk. This instruction is a non-privileged instruction.

b) Copy from XK per (Xj).

CPYXS - (Format = jk Op Code = 0F Ref# = 131)

label	operation	argument
	CPYXS	Xk,Xj

This instruction copies the contents of Register Xk into the state register addressed by the contents of Register Xj.

7.6.4.3 PURGE-Purge Buffer

a) Purge Buffer k of Entry per (Xj).

PURGE - (Format = jk Op Code = 05 Ref# = 138)

label	operation	argument
	PURGE	Xj,k

The Purge Buffer instruction invalidates entries in the Map and Cache buffers. The purge may invalidate all entries in a buffer, invalidates all entries in a buffer which derive from a given segment, invalidate all entries in a buffer for a given page, or invalidate all entries in a buffer for a given 512 byte block. Register Xj contains the required address information, either System Virtual Address (SVA) or Process Virtual Address (PVA).

An SVA contains the Active Segment (ASID) in bits 16 through 31 of Register Xj. A PVA contains the Segment number (SEG) in bits 20 through 31 of Register Xj. Bits 32 through 63 contain the

Byte Number (BN) for either an SVA or a PVA. The rightmost 9 bits of the BN are ignored and assumed to be zeros since the smallest purgeable portion of a buffer is a 512 byte page or a 512 byte block of a larger page. Proportionately more rightmost bits of the BN are ignored and assumed to be zero as page size becomes larger than the 512 byte minimum.

```
16        20                   32                  55        63
+-----------------------------+-------------------+----------+
|//////////|      SEG         |        BN         |//////////|
+----------+------------------+-------------------+----------+
|          |      ASID        |        BN         |//////////|
+-----------------------------+-------------------+----------+
```

The value of k determines the buffer to be purged, the range of entries to be purged, and the type of addressing used to determine the range of entries to be purged. The definition of k follows.

k=0, Purge all entries in Cache which are included in the 512 byte block defined by the SVA in Xj.
k=1, Purge all entries in Cache which are included in the ASID defined by the SVA in Xj.
k=2, Purge all entries in Cache.
k=3, Purge all entries in Cache which are included in the 512 byte block defined by the PVA in Xj.
k=4->7, Purge all entries in Cache which are included in the SEG defined by the PVA in Xj.
k=8, Purge all entries in Map which are included in the page defined by the SVA in Xj. This size of the page involved shall be determined by the contents of the Page Size Mask Register.
k=9, Purge all entries in Map which are included in the ASID defined by the SVA in Xj.
k=A, Purge all information from the map pertaining to the PTE defined by the PVA in Xj. The size of the page involved shall be determined by the contents of the Page Size Mask Register.
k=B, Purge all information from the MAP pertaining to the SDE defined by PVA in Xj, and to all PTE's included within that segment.
k=C->F, Purge all entries in Map.

For k=0, 1, 2, 8->F this instruction is a local privileged instruction. It is non-privileged for all other values of k.

7.7 VECTOR INSTRUCTIONS

7.7.1 GENERAL DESCRIPTION

This class of instructions operate on vectors, that is, sequences of full-word integer or real numbers. These instructions are only implemented on the Cyber 180 Model 99x class and on the Cyber 2000V. Attempting to execute a vector instruction on any other processor will result in an Unimplemented Instruction condition.

7.7.2 COMMON ATTRIBUTES OF VECTOR INSTRUCTIONS

All vector instructions utilize the jkiD instruction format. However, some instructions do not use all operand fields. In general, the J operand either is an A register which points to a source vector, or is an X register which contains a value which is turned into a vector by "broadcasting" or repeating the value the necessary number of times. The K operand is an A register which points to the destination vector. The I operand is normally a second source vector, but is used differently by some instructions. All addresses used by vector instructions must point to a word boundary, or an Address Specification error will result.

In the instruction descriptions that follow, V(Aj) represents either the vector addressed by Aj, or the broadcast vector created from the value in Xj.

The D field contains the length of the vector, when non-zero. It must be an positive integer less than or equal to 512. This is the size of the vector in words. When the rightmost ten bits of the D field are zero, X1 Right specifies the length of the vector. When X1 Right is negative, an Instruction Specification error is recorded, except on the Cyber 2000V, where the instruction will be treated as a no-op. When X1 Right is greater than 512, 512 is used for the size of the vector. When the rightmost ten bits of the D field are greater than 512, an Instruction Specification is recorded.

The leftmost bit of the D field is set by the Assembler when the J operand is an X register, to indicate that broadcasting shall take place.

7.7.3 INTEGER VECTOR ARITHMETIC

7.7.3.1 ADDXV-Add Integer Vectors

a) Integer vector sum, V(Ak) replaced by V(Aj) plus V(Ai).

ADDXV - (Format = jkiD Op Code = 44 Ref# = 172)

label	operation	argument
	ADDXV	Ak,Aj,Ai,D
	ADDXV	Ak,Xj,Ai,D

The first form of this instruction adds each word in the vector pointed to by Aj to the corresponding value in the vector pointed to by Ai, storing the result in the vector pointed to by Ak. The second form adds the value in Xj to each word pointed to by Ai, storing the result in the vector pointed to by Ak.

7.7.3.2 SUBXV-Subtract Integer Vectors

a) Integer vector difference, V(Ak) replaced by V(Aj) minus V(Ai).

SUBXV - (Format = jkiD Op Code = 45 Ref# = 173)

label	operation	argument
	SUBXV	Ak,Aj,Ai,D
	SUBXV	Ak,Xj,Ai,D

In the first form, each value in the vector pointed to by Ai is subtracted from its corresponding value in the vector pointed to by Aj. The results are stored in the vector pointed to by Ak. In the second form, the values pointed to by Ai are subtracted from the value in Xj.

7.7.4 INTEGER VECTOR COMPARISON

The following four instructions compare corresponding elements of two vectors. The results are stored in the vector indicated by Ak. If the compare is true, bit 0 of the corresponding word in V(Ak) is set and bits 1 through 63 are cleared. If the compare is false, bits 0 through 63 are cleared. If the second form is used, where Xj is specified, each value in V(Ai) is compared to the value in Xj.

The following example shows the results in V(Ak) after the instruction is executed.

```
        CMPEQV  A9,A7,A8,3

        +-------+       +-------+         +--------+
  A7--->|    230|  A8-->|    200|    A9-->|00...00 |
        |     75|       |     75| (binary)|10...00 |
        |     18|       |     27|         |00...00 |
        +-------+       +-------+         +--------+
```

7.7.4.1 CMPEQV-Integer Vector Comparison - Equal

a) Integer vector compare, V(Ak) replaced by V(Aj) equal to V(Ai).

CMPEQV - (Format = jkiD Op Code = 50 Ref# = 176)

label	operation	argument
	CMPEQV	Ak,Aj,Ai,D
	CMPEQV	Ak,Xj,Ai,D

7.7.4.2 CMPLTV-Integer Vector Comparison - Less Than

a) Integer vector compare, V(Ak) replaced by V(Aj) less than V(Ai).

CMPLTV - (Format = jkiD Op Code = 51 Ref# = 177)

label	operation	argument
	CMPLTV	Ak,Aj,Ai,D
	CMPLTV	Ak,Xj,Ai,D

7.7.4.3 CMPGEV-Integer Vector Comparison - Greater Than Or Equal

a) Integer vector compare, V(Ak) replaced by V(Aj) greater than or equal to V(Ai).

CMPGEV - (Format = jkiD Op Code = 52 Ref# = 178)

label	operation	argument
	CMPGEV	Ak,Aj,Ai,D
	CMPGEV	Ak,Xj,Ai,D

7.7.4.4 CMPNEV-Integer Vector Comparison - Not Equal

a) Integer vector compare, V(Ak) replaced by V(Aj) not equal to V(Ai).

CMPNEV - (Format = jkiD Op Code = 53 Ref# = 179)

label	operation	argument
	CMPNEV	Ak,Aj,Ai,D
	CMPNEV	Ak,Xj,Ai,D

7.7.5 SHIFT VECTOR CIRCULAR

a) Shift vector circular, V(Ak) replaced byV(Aj), direction and count per V(Aj).

SHFV - (Format = jkiD Op Code = 4D Ref# = 180)

label	operation	argument
	SHFV	Ak,Aj,Ai,D
	SHFV	Ak,Xj,Ai,D

This instruction performs a left circular shift on each element of V(Ai), as directed by the corresponding element of V(Aj), storing the results in V(Ak). The shift count for each element of V(Ai) is taken form the rightmost 8 bits of the corresponding element of V(Aj) and is interpreted as follows:

The sign-bit in the leftmost position of the 8-bit shift count shall determine the direction of the shift. When the shift count is positive (sign bit of zero), this instruction shall left shift. When the shift count is negative (sign bit of one), this instruction shall right shift. Shifts shall be from 0-63 bits left and from 1-64 bits right. Based on an 8-bit signed 2's complement shift count, these shifts are as follows:

```
0111 1111  --\
    :          --  Left Shift 0-63
0100 0000  --/
0011 1111          Left Shift 63
    :                  :
0000 0000          Left Shift 0
-----------------------------------
1111 1111          Right Shift 1
    :                  :
1100 0000          Right Shift 64
1011 1111  --\
    :          --  Right Shift 1-64
1000 0000  --/
```

When these interpretations of the shift count result in an actual shift count of zero, the instruction transfers the element of V(Ai) to the corresponding element of V(Ak) with no shift.

When broadcast of V(Aj) is selected and j=0, the contents of the X0 register shall be interpreted as consisting entirely of zeros.

7.7.6 LOGICAL VECTORS

7.7.6.1 IORV-Inclusive Or Vectors

a) Logical vector sum, V(Ak) replaced by V(Aj) OR V(Ai).

IORV - (Format = jkiD Op Code = 48 Ref# = 181)

label	operation	argument
	IORV	Ak,Aj,Ai,D
	IORV	Ak,Xj,Ai,D

7.7.6.2 XORV-Exclusive Or Vectors

a) Logical vector difference, V(Ak) replaced by V(Aj) XOR V(Ai).

XORV - (Format = jkiD Op Code = 49 Ref# = 182)

label	operation	argument
	XORV	Ak,Aj,Ai,D
	XORV	Ak,Xj,Ai,D

7.7.6.3 ANDV-Logical And Vectors

a) Logical vector product, V(Ak) replaced by V(Aj) AND V(Ai).

ANDV - (Format = jkiD Op Code = 4A Ref# = 183)

label	operation	argument
	ANDV	Ak,Aj,Ai,D
	ANDV	Ak,Xj,Ai,D

7.7.7 CONVERT VECTORS

7.7.7.1 CNIFV-Convert Vector From Integer to Float

a) Convert vector, floating point V(Ak) formed from integer V(Aj).

CNIFV - (Format = jkiD Op Code = 4B Ref# = 184)

label	operation	argument
	CNIFV	Ak,Aj,D
	CNIFV	Ak,Xj,D

7.7.7.2 CNIFV-Convert Vector From Float to Integer

a) Convert vector, integer V(Ak) formed from floating point V(Aj).

CNFIV - (Format = jkiD Op Code = 4C Ref# = 185)

label	operation	argument
	CNFIV	Ak,Aj,D
	CNFIV	Ak,Xj,D

7.7.8 FLOATING POINT VECTOR ARITHMETIC

7.7.8.1 ADDF-Floating Point Vector Sum

a) Floating point vector sum, V(Ak) replaced by V(Aj) plus V(Ai).

ADDFV - (Format = jkiD Op Code = 40 Ref# = 186)

label	operation	argument
	ADDFV	Ak,Aj,Ai,D
	ADDFV	Ak,Xj,Ai,D

7.7.8.2 SUBFV-Floating Point Vector Difference

a) Floating point vector difference, V(Ak) replaced by V(Aj) minus V(Ai).

SUBFV - (Format = jkiD Op Code = 41 Ref# = 187)

label	operation	argument
	SUBFV	Ak,Aj,Ai,D
	SUBFV	Ak,Xj,Ai,D

7.7.8.3 MULFV-Floating Point Vector Product

a) Floating point vector product, V(Ak) replaced by V(Aj) times V(Ai).

MULFV - (Format = jkiD Op Code = 42 Ref# = 188)

label	operation	argument
	MULFV	Ak,Aj,Ai,D
	MULFV	Ak,Xj,Ai,D

7.7.8.4 DIVFV-Floating Point Vector Quotient

a) Floating point vector quotient, V(Ak) replaced by V(Aj) divided by V(Ai).

DIVFV - (Format = jkiD Op Code = 43 Ref# = 189)

label	operation	argument
	DIVFV	Ak,Aj,Ai,D
	DIVFV	Ak,Xj,Ai,D

7.7.9 FLOATING POINT VECTOR SUMMATION

7.7.9.1 SUMFV-Floating Point Vector Summation

a) Floating point vector summation, Xk replaced by summation of elements in V(Ai).

SUMFV - (Format = jkiD Op Code = 57 Ref# = 190)

label	operation	argument
	SUMFV	Xk,Ai,D Merge Vector

7.7.9.2 MRGV-Merge Vector

a) Merge vector, V(Ak) partially replaced by V(Aj) per mask V(Ai).

MRGV - (Format = jkiD Op Code = 54 Ref# = 191)

label	operation	argument
	MRGV	Ak,Aj,Ai,D
	MRGV	Ak,Xj,Ai,D

This instruction replaces the first element of V(Ak) with the first element of V(Aj) if bit 0 is set in the first element of V(Ai). If bit 0 is clear, the first element of V(Ak) is left unchanged. This operation in repeated for successive elements until the required number of operations has been performed.

7.7.10 GATHER AND SCATTER VECTOR

7.7.10.1 Gather Vector

a) Gather vector, V(Ak) replaced by gathered V(Aj) with interval Xi.

GTHV - (Format = jkiD Op Code = 55 Ref# = 192)

label	operation	argument
	GTHV	Ak,Aj,Xi,D
	GTHV	Ak,Xj,Ai,D

This instruction obtains the first element from V(Aj) and stores it as the first element of V(Ak). The second element to be stored in V(Ak) is taken from the address formed by adding the rightmost 32 bits of Xi, shifted left three places with zero fill, to the rightmost 32 bits of Aj. Successive elements in V(Ak) are taken from the address formed by adding the rightmost 32 bits of Xi, shifted left three places with zero fill, to the rightmost 32 bits of the previous address. The Nth (1,2,3,...,n,...) element of V(Ak) is replaced by V(Aj) whose address is (Aj)+8*(n-1)*(Xi). The contents of register Xi are not altered by the execution.

Thus, contiguous vector V(Ak) is formed by gathering elements from V(Aj) at interval Xi.

7.7.10.2 Scatter Vector

a) Scatter vector, V(Ak) replaced by scattered V(Aj) with interval Xi.

SCTV - (Format = jkiD Op Code = 56 Ref# = 193)

label	operation	argument
	SCTV	Ak,Aj,Xi,D
	SCTV	Ak,Xj,Ai,D

This instruction obtains the first element from V(Aj) and stores it as the first element of V(Ak). The second contiguous element from V(Aj) is stored into V(Ak) at the address formed by adding the rightmost 32 bits of Xi, shfited left three places with zero fill, to the rightmost 32 bits of Ak. Successive elements from V(Aj) are stored into the addresses formed by adding the rightmost 32 bits of Xi, shifted left three places with zero fill, to the rightmost 32 bits of the prevous address. The Nth (1,2,3,...,n,...) element of V(Aj) is stored into V(Ak) at (Ak)+8*(n-1)*)Xi).

Thus, the contiguous elements from V(Aj) are scattered in V(Ak) at interval Xi.

7.8 EXTENDED VECTOR INSTRUCTIONS

7.8.1 GENERAL DESCRIPTION

This class of instructions is implemented only on the Cyber 2000V. In all other respects, including broadcasting of Xj, they are similar to the other vector instructions.

7.8.2 FLOATING POINT VECTOR TRIADS

7.8.2.1 TPSFV, TPDFV, TSPFV, TDPFV - Vector Triad Instructions

a) Floating point vector triad, V(Ak) replaced by [V(Ai) times X0] plus V(Aj).

TPSFV - (Format = jkiD Op Code = 58 Ref# = 195)

label	operation	argument
	TPSFV	Ak,Aj,Ai,X0,D
	TPSFV	Ak,Xj,Ai,X0,D

b) Floating point vector triad, V(Ak) replaced by [V(Ai) times (X0)] minus V(Aj).

TPDFV - (Format = jkiD Op Code = 59 Ref# = 196)

label	operation	argument
	TPDFV	Ak,Aj,Ai,X0,D
	TPDFV	Ak,Xj,Ai,X0,D

c) Floating point vector triad, V(Ak) replaced by [V(Aj) plus V(Ai)] times (X0).

TSPFV - (Format = jkiD Op Code = 5A Ref# = 197)

label	operation	argument
	TSPFV	Ak,Aj,Ai,X0,D
	TSPFV	Ak,Xj,Ai,X0,D

d) Floating point vector triad, V(Ak) replaced by [V(Aj) minus V(Ai)] times (X0).

TDPFV - (Format = jkiD Op Code = 5B Ref# = 198)

label	operation	argument
	TDPFV	Ak,Aj,Ai,X0,D
	TDPFV	Ak,Xj,Ai,X0,D

These instructions perform the indicated floating point arithmetic operations on the first element from V(Aj) and the floating point constant contained in X0 and store the result as the first element of V(Ak). The arithmetic operation inside the brackets is done first. The operation is repeated for successive elements until the required number of operations has been performed. The contents of X0 are not altered by the execution of these instructions.

7.8.3 FLOATING POINT VECTOR DOT PRODUCT

7.8.3.1 SUMPFV - Floating Point Vector Dot Product

a) Floating point vector dot product, Xk replaced by summation of [V(Aj) times V(Ai)]

SUMPFV - (Format = jkiD Op Code = 5C Ref# = 199)

label	operation	argument
	SUMPFV	Xk,Aj,Ai,D
	SUMPFV	Xk,Xj,Ai,D

This instruction multiplies each element of V(Aj) times its corresponding element in V(Ai). The resulting products are added together and the sum is stored in Xk. The summation may be done in any order.

7.8.4 GATHER/SCATTER VECTORS - INDEX LIST

7.8.4.1 GTHIV - Gather Vector Per Index List

Gather vector, V(Ak) replaced by gathered V(Aj) per index list V(Ai).

GTHIV - (Format = jkiD Op Code = 5D Ref# = 200)

label	operation	argument
	GTHIV GTHIV	Ak,Aj,Ai,D Ak,Xj,Ai,D

This instruction obtains the first element from V(Ai). The first element to be stored in V(Ak) is taken from the address formed by adding the rightmost 32 bits of the first element from V(Ai), shifted left three places with zero fill, to the rightmost 32 bits of (Aj). Successive elements in V(Ak) are taken from the addressess formed by adding the rightmost 32 bits of the successive elements from V(Ai), shifted left three places with zero fill, to the rightmost 32 bits of (Aj). The Nth (1,2,3,...N,...) element of V(Ak) is replaced by the element of V(Aj) whose address is (Aj) + 8 times the Nth element of V(Ai).

Thus, contiguous vector V(Ak) is formed by gathering elements from V(Aj) at indexes from list V(Ai).

7.8.4.2 SCTIV - Scatter Vector Per Index List

Scatter vector, V(Ak) replaced by scattered V(Aj) per index list V(Ai).

SCTIV - (Format = jkiD Op Code = 5E Ref# = 201)

label	operation	argument
	SCTIV SCTIV	Ak,Aj,Ai,D Ak,Xj,Ai,D

This instruction obtains the first elements from V(Ai) and V(Aj). The first element from V(Aj) is stored into V(Ak) at the address formed by adding the rightmost 32 bits of the first element from V(Ai), shifted left three places with zero fill, to the rightmost 32 bits of (Ak). Successive elements in V(Aj) are stored into V(Ak) at the address formed by adding the rightmost 32 bits of the successive elements from V(Ai), shifted left three places with zero fill, to the rightmost 32 bits of (Ak). The Nth (1,2,3,...N,...) element of V(Aj) is stored into V(Ak) at (Ak) + 8 times the Nth element of V(Ai).

Thus, contiguous elements from V(Aj) are scattered in V(Ak) at indexes from list V(Ai).

APPENDIX A

CALLING THE ASSEMBLER

The Assembler is called on NOS/VE with the command name "ASSEMBLE" followed by parameters in the System Command Language format. All Assembler call parameters are optional. Parameters of the Assembler are:

I INPUT=file

INPUT specifies the file containing source statements that are to be assembled. If this parameter is omitted the value $INPUT will be used.

B BINARY_OBJECT=file

BINARY_OBJECT specifies the file to receive the object text (binary) that is generated bu the assembler. If this parameter is omitted the value LGO will be used.

L LIST=file

LIST specifies the file to receive the assembly listing. If this parameter is omitted the value $LIST will be used.

E ERROR=file

ERROR specifies the file to receive the listing of assembly errors. If this parameter is omitted the value $ERRORS will be used.

LO LIST_OPTIONS=list of A, R, S, NONE

LIST_OPTIONS specifies the content of the listing file. If S is included in the list, the source and generated code are listed. If A is included, the symbol attributes listing is included. If R is specified, the cross-reference is listed. If NONE is specified, only errors will be listed. The default value is S.

C CHECKS=boolean

CHECKS specifies whether assembly checks are to be performed or omitted. Assembly checks are used with the CPU instruction set to validate that the correct register type designators (A-reg or X-reg) are used. If this parameter is omitted a value of TRUE will be used.

STATUS=status variable

STATUS specifies a status variable to receive the command's termination status.

EXAMPLE:

```
ASSEMBLE I=SOURCE B=BIN L=LISTING LO=(S,A,R) C=TRUE
```

APPENDIX B - NOTES AND EXAMPLES

PROGRAMMING NOTES

To fully understand the Cyber 180 Hardware instructions and thier parameters, one must first understand that the Cyber 180 machine is designed to be Stack oriented. Software written for the Cyber 180 will be written in a Stack oriented higher level language (CYBIL). However there will be some code that will have to be written in Assembly language (ie Hardware diagnostics). The following sections contain notes that will hopefully aid in writing Assembly language programs.

REGISTER USAGE

When writing in Assembly language, it is important to understand how the hardware works, especially register usage. The contents of the following registers are assumed to be as described by the hardware, and should not be overwritten.

A0 - Dynamic Space Pointer.
A1 - Current Stack Frame Pointer.
A2 - Previous Save Area.
A3 - Binding Section Pointer.
A4 - Argument Pointer.

GENERAL NOTES

In addition to understanding the hardware, it is also important to understand some things about the Assembler.

SECTIONS-SEGMENTS The relationship between the Assembler concept of Sections and the Hardware concept of Segments is similar, but differs in that two or more sections may be loaded in the same Hardware Segment when they have the same access permissions.

RELOCATABILITY OF CODE Even though code in sections is assembled as absolute, the sections can be loaded as relocatable, and are accessed via pointers.

MONOLITH PROGRAMS When mixing code and data in the same section, it is important to use the ALIGN command when resuming to generate code. This will ensure that the code is generated on the proper boundary.

SAMPLE PROGRAM

The following is a sample program available in the SES catalog. It is intended to aid in the understanding of the CYBER 180 CPU Assembler and the CYBER 180 hardware.

```
test      ident               .sample program
          def    ent1         .defines the entry point
...
.  This program will pick up an entry from the Literal section,
.  and makes a copy of it in the working section.  The program
.  is structured to use the default sections established by
.  the Assembler, and is executed using the C180 defaults.
...
          space  3
          use    working      .The working section will get loaded intoa
.                              segment with read+write permissions.
dum       bss    1            .Put here to show effect of align.
          align  0,8          .Ensures word boundary.
temp      bssz   20(16)       .20(16) bytes(4 words) of temp storage
          space  3
          use    working      .The WORKING section will be loaded
.                              into a segment with read permission.
          align  0,8          .Word boundary.
msg       vfd,8*8  c'EXAMPLE ' .Test data to be moved
          space  3
          use    binding      .The Binding section is used by the
.                              hardware to store pointers which
.                              facilitate the binding of segments.
.                              This section will be loaded into
.                              a segment with read+bind permissions.
msg_pt    address p,msg       .Creates a pointer to MSG.
temp_pt   address p,temp      .Creates a pointer to TEMP.
.                              Pointers are set up with segment number
.                              set to FFF, LINKER fills in this field.
.                              The location field will show an offset of
.                              word boundary + 2, because the 6 byte PVA
.                              is right justified in the 8 byte field.
          space  3
          use    code         .The Code section will be loaded into
.                              a segment with read+execute permissions.
          space  1
          proc                .This proc will count the number of
count     pname               .bytes moved.
num_move  set    num_move+1c:(f:(2,0))  .Add the number of bytes
          pend                .end of procedure
          space  2
```

```
ent1      align  0,8             .Entry point on a word boundary
num_move  set  0                 .Initialize byte counter
          ente   x0,33(16)       .Include X0-X3 and A0-A3 when
.                                 saving the environment.
          callrel move_msg,a3,a4  .move a copy of msg to temp
.
          return                 .End execution.
.
.  Move_msg will move data to working storage
.
move_msg  align  0,8             .Ensure word boundary.
          la     a5,a3,msg_pt        .Load into A5 the pointer to MSG.
          lx     x1,a5,0             .Load data into X1.
          la     a5,a3,temp_pt       .A5 = pointer to storage area.
          sx     x1,a5,0             .Store MSG.
          count  msg                 .Update NUM_MOVE.
          return                     .Return to caller
.
          end    ent1            .Ent1 is transfer label
```

SAMPLE EXECUTION

The sample program in the previous section was executed as shown below:

To be supplied later.

APPENDIX C - RESERVED WORDS

The following words or categories have special meaning and can not be redefined in the user's program.

Register identifiers (A0-AF, X0-XF)

Section identifiers (binding, code, stack, working)

Section types (Code, Binding, Working, Common, Extwork, Extcom)

Attribute identifiers (Bind, Execute, Read, Write)

Machine identifiers (C180CPU, C180IOU)

All pseudo and machine mnemonics.

All symbols starting with the pound-sign character.

Any symbol containing a colon.

Special internal symbols(PADA, PADB, SECT, ASECT, DSECT)

APPENDIX D - ERROR MESSAGES

Error messages may appear either on the listing, and/or on the dayfile, depending on when the error is detected.

LISTING ERRORS

Message

ALIAS NAME INVALID OR DUPLICATE

SIGNIFICANCE
The alias name has been defined as both an internal and external entry point. (ie. appearing on both a DEF or DEFG instruction and a REF instruction).

ACTION
An internal entry point must be unique. However, two external entry points can be aliased to the same linkage symbol.

ALIASED SYMBOL MUST BE REF OR DEF SYMBOL

SIGNIFICANCE
The label field of an alias statement has not been defined in a DEF, DEFG, or REF pseudo instruction.

ACTION
Define the entry point to be aliased in a DEF, DEFG, or REF instruction. Note for a DEF or DEFG symbol, these values must be further defined as a relocatable symbol (symbol category = 6).

ANAME SYMBOL REQUIRED FOR ATTRIBUTES REFERENCING

SIGNIFICANCE
Encountered an ATRIB statement where the user defined attribute name was not previously defined in an ANAME statement.

ACTION
Define attribute name using the ANAME pseudo instruction.

A-REG DESIGNATOR REQUIRED

SIGNIFICANCE
An A register is required in instruction.
ACTION
Check register specifications for instruction in ERS.

ARGUMENT SUBFIELD MUST BE SYMBOLIC NAME

SIGNIFICANCE
The argument field of the following pseudo instructions must be a symbol and cannot be an expression: ADDRESS, ALIAS, END, ERROR, FLAG, LOCAL, OPEN, REF, SECTION, SKIPTO, and TITLE. (An exception is the address type R on the ADDRESS instruction.)
ACTION
Check the ERS for definition of the argument field. Many of these instructions have pre-defined values for use in the argument field.

BDP DESCRIPTOR ERROR

SIGNIFICANCE
There's an error in either the source or destination data descriptor within a BDP instruction.
ACTION
Check register specifications and descriptor limitations for instruction in ERS.

BINDING ATTRIBUTE MUST BE BINDABLE OR NONBINDABLE

SIGNIFICANCE
The 'bind' type in the argument field of the MACHINE pseudo instruction is not one of the pre-defined values BINDABLE or NONBINDABLE.
ACTION
Check value in argument field of the MACHINE pseudo instruction.

CHARACTER STRING TOO LONG

SIGNIFICANCE
A character string cannot exceed one line, therefore is limited to 87 characters.
ACTION
Check for missing quote mark or shorten current string.

CMD STATEMENT ILLEGAL IN PROCEDURE DEFINITION

SIGNIFICANCE

A CMD instruction is equivalent to a one statement procedure definition. Nested procedures are not allowed, therefore a CMD statement cannot be within a procedure definition.

ACTION

Take the CMD statement out of the procedure. Or redefine the CMD statement as a separate procedure and replace the CMD with a 'procedure call'.

DATA GENERATION IN STACK OR BINDING SECTION

SIGNIFICANCE

Data cannot be initialized in the STACK or BINDING sections at assembly time. (An exception is the binding section in which pointers can be initialized with the ADDRESS pseudo instruction.)

ACTION

Check the last USE statement which was encountered.

DISPLACEMENT VALUE IS OUT-OF-RANGE

SIGNIFICANCE

The displacement value on a machine instruction overflows the length of the field designated by the instruction.

ACTION

Check the ERS for the calculation of the address displacement to make sure the value can be represented by the number of bits allotted for the displacement (ie. for a 16 bit Q-field with sign extension the value must be in the range: -7fff(16) < value < 7fff(16)).

DIVISION BY ZERO ATTEMPTED

SIGNIFICANCE

While evaluating an expression, an attempt to divide by zero was made.

ACTION

Check values in the divisor portion.

ERROR STATEMENT = 'character string'

SIGNIFICANCE

The expression in the ERROR statement evaluated to true causing the string or symbol in the argument field to be printed in the object listing. Control is transferred conditionally on the presence of a label in the operation subfield.

ACTION

Check ERS for rules concerning the ERROR statement and the transfer of control.

EXPRESSION EVALUATION ERROR

SIGNIFICANCE

While processing an expression, an arithmetic overflow or underflow has occurred. The following conditions will cause this error:

- exceeding the following limits in integer arithmetic

  32 bit integer - -2(31) <= M <= 2(31) - 1

  64 bit integer - -2(63) <= M <= 2(63) - 1

- exponent overflow and underflow are detected for all single precision, but only for the leftmost part of double precision.

  floating point absolute value - 5.2 * 10**1232

- for general BDP instructions with data descriptors, the source operand fields will be checked for overflow but the destination operand will not.
- in BDP floating point instructions, if the capacity of designated fields are exceeded such that significant digits are lost.
- an exception is the CALDF and EDIT instructions, no overflow conditions detected for these.

ACTION

Check values used in the expression evaluation.

FIELD REFERENCE ERROR

SIGNIFICANCE

This error occured because some field in the source statement requires a symbolic name but an illegal field reference (ie. F: function) or list reference (ie. symbol[X]) was encountered. The value that either of these functions represent is not a symbolic name.

ACTION

Check the fields in the source statement that require symbolic names (ie. label fields, operation subfields as in the SKIPTO statement, etc.). One of the values being referenced is not defined to be a symbolic name.

FIRST STATEMENT IS NOT IDENT

SIGNIFICANCE
The first source statement encountered bv the assembler was not an IDENT instruction. The only permissible source lines before the IDENT are comments. This is also true for multiple assembly modules, the only allowable source lines between the END and the IDENT are comments.

ACTION
Delete those statements before the IDENT instruction.

FLAG STATEMENT ERROR

SIGNIFICANCE
The FLAG statement was processed which conditionally sets an error flag. The two permissible error types are pre-defined as FATAL and WARNING.

ACTION
Processing of this statement does not affect other code.

GENERATED CODE IS NOT "BINDABLE"

SIGNIFICANCE
The relocation information generated with a CMD, VFD, INT, or DINT statement does not correspond to the pre-defined values of the RCT or ADT fields of the Relocation attribute. Both the Relocation Container Type and the Address Displacement Type are pre-defined and the relocation information must agree with these attribute values.

ACTION
Check values on these data generating statements so as to make sure that all relocation information has the correct values, ie. one of those that is pre-defined.

ILLEGAL ATTRIBUTE REFERENCE

SIGNIFICANCE
When evaluating the argument of an attribute, either defined in an ANAME statement or an internal attribute (ie. #REGTYP), an illegal argument was encountered or

the argument was missing. This can also occur if a register specification in a symbolic machine instruction is incorrect.

ACTION

Check argument field of an attribute reference or check the ERS for correct register specifications for machine instructions.

ILLEGAL CONTINUATION

SIGNIFICANCE

The card following a continuation card contained a non-blank character in column 1. This could also be a non-graphic character.

ACTION

Change the card following the continuation character to contain a blank in column 1.

ILLEGAL EXPRESSION

SIGNIFICANCE

While evaluating an expression an illegal reference has been encountered by the assembler. This can be an element number reference, an attribute reference, an intrinsic or user-defined function reference.

ACTION

Check the following conditions:

- element number reference - using parenthesis rather than brackets or trying to access a list value of a symbol that is not a SET/EQU symbol,
- attribute reference - using parenthesis rather than brackets or having more than one argument,
- intrinsic or user-defined function reference - using brackets rather than parenthesis or having no argument or a null argument field.

ILLEGAL OR NON-GRAPHIC CHARACTER DETECTED

SIGNIFICANCE

An illegal or non-graphic character has been detected. Note that a single quote, which is not preceded by a symbolic character, will cause this error.

ACTION

The assembler accepts any graphic ASCII character in a comment or character string. Check the ERS under character set for the ASCII subset which the assembler accepts as input.

ILLEGAL STATEMENT IN FUNCTION EXPANSION

SIGNIFICANCE
A function may not generate code or change location counters, if it is called form a statement which itself, generates code. This condition may occur in any of the following statements: ALIGN, BSS, BSSZ, INT, DINT, FLOAT, DFLOAT, PDEC, VFD or a CMD call statement.
ACTION
Change the function or the source statement from which it is called.

INSUFFICIENT NUMBER OF ARGUMENTS

SIGNIFICANCE
In either a CMD or VFD statement, the number of elements in the value list is less than the number of elements in the length list.
ACTION
Check the elements in the value list. Note that if the number of elements in the value list exceeds the number of elements in the length list no diagnostic occurs and any extra arguments are ignored.

INTEGER OR REAL NUMBER CONVERSION ERROR

SIGNIFICANCE
The floating point number in the argument field of a FLOAT or DFLOAT pseudo instruction is an infinite or indefinite value.
ACTION
Limits on minimum and maximum values and exponents can be found in the CYBER 180 math library documents.

INVALID ELEMENT NUMBER IDENTIFIER

SIGNIFICANCE
The element number being referenced has a value less than 0.
ACTION
Check expression within the brackets which must be greater than or equal to 0.

APPENDIX D - ERROR MESSAGES
LISTING ERRORS

INVALID LOCATION COUNTER DESIGNATOR

SIGNIFICANCE
The value X in $(X) did not evaluate to 0 or 1.
ACTION
The value X can be an expression but this expression must evaluate to 0 for current byte offset or 1 for current bit offset. If no value is given the function defaults to 0.

INVALID MACHINE TYPE

SIGNIFICANCE
The IDENT pseudo instruction is the first statement recognized by the assembler and it pre-defines the processor type due to the argument field. If this value does not correspond with the type on the MACHINE pseudo instruction this error will be produced. Otherwise, the type in the argument field is not one of the following pre-defined values, C180CPU or C180IOU.
ACTION
Check the argument field of the IDENT and MACHINE pseudo instructions to insure they correspond to the same processor type.

INVALID SECTION ATTRIBUTES

SIGNIFICANCE
The attributes defined on a SECTION statement are either not in the set of pre-defined attributes or there's an illegal expression in the definition of these segment access attributes.
ACTION
The pre-defined segment access attributes are: READ, WRITE, EXECUTE and BIND and the only operator permitted is the plus (+) operator.

INVALID SECTION TYPE

SIGNIFICANCE
The section type used in the SECTION statement was not in the set of pre-defined types. Or the section type was CODE, BINDING or STACK and these are already defined by the assembler and cannot be redefined by the user.
ACTION

The section types available to the user are: WORKING, COMMON, EXTWORK and EXTCOM.

INVALID SYMBOL ERROR

SIGNIFICANCE
The symbol encountered was illegal because of one of the following conditions:
- the first character of the symbol does not begin with one of the legal alphabetic characters defined for the assembler.
- there's a colon (:) somewhere in the symbol,
- the symbol is in the list of the assembler's reserved words (see Appendix C of the ERS).

ACTION
Check symbol for illegal character or that it appears on the reserved word list.

INVALID "TYPI" SUBFIELD IN ADDRESS STATEMENT

SIGNIFICANCE
The address type in the argument field of the ADDRESS instruction is not one of the pre-defined types.

ACTION
The address types for the ADDRESS instruction are defined as: P, C, CI, CE, or R.

LABEL NOT SYMBOLIC NAME

SIGNIFICANCE
The label field of one of the following statements does not contain a legal symbol: ALIAS, ANAME, ATRIB, CMD, DO, WHILE, DEND, IDENT, SET or EQU.

ACTION
Check the label field on the source statement.

MACHINE STATEMENT MUST PRECEDE CODE GENERATION

SIGNIFICANCE
The MACHINE pseudo instruction did not precede a data generating statement.

ACTION
The MACHINE pseudo instruction must appear before any statment which generates code. Also there can be only one MACHINE pseudo instruction between an IDENT and an

END assembly unit.

MAXIMUM SEGMENT OFFSET EXCEEDED

SIGNIFICANCE
Code has been generated in a section that overflows the maximum offset allowed by the operation system. This value is 0FFFFFFFF(16).
ACTION
Check the section that currently is being used for code generation.

MISSING CONT STATEMENT

SIGNIFICANCE
While processing a procedure call or a DO/WHILE sequence of statements a SKIPTO was encountered with a name in it's argument field that did not appear before a PEND or DEND statement. This also occurs if the label on the ERROR statement does not appear.
ACTION
Check symbol names in argument field of SKIPTO statement.

MISSING DEND STATEMENT

SIGNIFICANCE
There's no matching DEND statement for a DO directive. An END or a PEND statement was encountered first.
ACTION
Include the DEND statement in assembly module.

MISSING OPERATION FIELD

SIGNIFICANCE
There's a value in the label field of the source statement which has nothing following it.
ACTION
A null operation field is illegal. Check source statement for missing value.

MISSING PEND STATEMENT

SIGNIFICANCE

A PROC directive was encountered but no statement between this and the END statement contained PEND in the operation field.

ACTION

Include the PEND statement in the assembly module.

NESTED PROCEDURE DEFINITION

SIGNIFICANCE

Encountered a PROC psuedo instruction between a PROC-PEND pair.

ACTION

Nested procedures are not allowed by the assembler. A PROC instruction must be followed by a PEND instruction before another PROC instruction can be processed.

OFFSET ARGUMENT NOT ON REQUIRED BOUNDARY

SIGNIFICANCE

While processing one of the offset functions (ie. #WOFF, #HOFF, #POFF, or #BOFF) the address of the argument does not fall on the appropriate boundary (ie. for #WOFF function the argument must be on a word boundary).

ACTION

Check the address of the function argument. Sue the ALIGN statement before the argument definition to assure the correct boundary.

OPERAND MUST BE A REAL NUMBER

SIGNIFICANCE

An operand in the argument field of a FLOAT or DFLOAT pseudo instruction is not a legal floating point number.

ACTION

Check the operands in argument field for legal floating point numbers. Note, all floating point values must be decimal values.

OPERAND TYPE INVALID

SIGNIFICANCE

The following pseudo instructions cause this error if the argument field is incorrect:

- ERROR - argument must be a legal symbol or ascii string,
- FLAG - argument field must be pre-defined symbols FATAL or WARNING,
- INFOMSG - if there is an argument, it must be the symbol LISTON,
- PDEC - the argument must be an ascii string with only the characters 0 - 9 or '+'/'-'. The '+'/'-' must be the last character in the string.

ACTION

Check the argument field for illegal value.

OPERATION SUBFIELD NOT A SYMBOLIC NAME

SIGNIFICANCE

One of the following two conditions has occurred:

- the operation field does not have a legal symbol name in it,
- or one of the following pseudo instructions does not have a legal symbol name in it's argument field: CLOSE, DEF, DEFG, LOCAL, OPEN or REF.

ACTION

Check the operation field or the argument field of the listed pseudo instructions.

PNAME/FNAME STATEMENT MISSING

SIGNIFICANCE

There was no PNAME or FNAME pseudo instruction between a PROC/PEND pair. Or the PNAME/FNAME instruction was not the instruction immediately following the PROC instruction.

ACTION

The PNAME/FNAME statements must be the first instruction after the PROC statement and there must be at least one PNAME/FNAME statement in a procedure defintion.

PNAME/FNAME STATEMENT OUT-OF-SEQUENCE

SIGNIFICANCE

The PNAME/FNAME statement is not immediately following a PROC, FNAME, or another PNAME statement.

ACTION

The PNAME/FNAME pseudo instructions are part of the procedure's heading along with the PROC statement. No other instruction can appear between a PROC and PNAME/FNAME statement.

RELOCATABLE SYMBOL REQUIRED (CATEGORY = 6)

SIGNIFICANCE
An ADDRESS, DEF or END pseudo instruction has a non-relocatable symbol (ie. symbol has a symbol category other than 6) in it's argument field.
ACTION
A relocatable term represents a location of some assembled code. These are defined in the label field of a data generating statement such as VFD, INT, DINT, FLOAT, DFLOAT, PDEC, BSS, BSSZ, ADDRESS, ORG, ALIGN or a call to a CMD instruction. The labels of the symbolic machine instructions will also have a symbol category equal to 6.

REQUIRED OPERAND MISSING

SIGNIFICANCE
The argument field is blank on a pseudo instruction that is required to have an operand.
ACTION
The following pseudo instructions require a value to be present in the argument field: ADDRESS, ALIAS, BSS, BSSZ, CLOSE, DEF, DEFG, FLAG, LOCAL, INT, DINT, FLOAT, DFLOAT, OPEN, ORG, PDEC, POS, REF, SECTION, SKIPTO, TITLE, USE, VFD, and a call to a CMD statement.

SECTION ALIAS NAME INVALID

SIGNIFICANCE
The 'cid' field on the SECTION statement is either not a symbol or has been previously used as a 'cid'.
ACTION
The 'cid' field is optional but if it's not used it must contain a legal null subfield (ie. two commas). If the symbol has already been used, redefine one of the fields.

SPECIFIED SECTION SIZE EXCEEDED

SIGNIFICANCE
The amount of code generated in the section exceeded

the amount given by the 'maxsize' field on the SECTION statement.

ACTION

Check value for 'maxsize' field on the SECTION statement and increase this value as necessary. The maximum segment length is OFFFFFFFF(16).

STATEMENT ILLEGAL IN IOU MODULE

SIGNIFICANCE

The ADDRESS, ALIAS, DEF, DEFG, INFOMSG, PDEC, DINT, FLOAT, DFLOAT, REF, SECTION, and USE pseudo instructions are illegal in an IOU assembly module.

ACTION

Delete these statements from assembly module.

STATEMENT IS VALID ONLY WITHIN A PROCEDURE

SIGNIFICANCE

The LOCAL pseudo instruction can only be used within a procedure definition (ie. between a PROC/PEND pair).

ACTION

The LOCAL pseudo instruction is used to define symbols local to a procedure. A PEND or an END statement terminates the symbols.

STATEMENT LABEL IS NOT UNIQUE

SIGNIFICANCE

The symbol encountered in the label field has already been defined. Note that this can be a directive or procedure/function name.

ACTION

Redefine one of the symbols and change the references to the symbol. Note if a symbol appears in the label field of a pseudo instruction that does not require a label, the symbol is not considered defined.

STATEMENT LABEL REQUIRED

SIGNIFICANCE

The label field of the source statement is blank.

ACTION

The following pseudo instructions require a label field: ANAME, ATRIB, CMD, SET, EQU, PNAME, FNAME, and

SECTION.

SYMBOL CANNOT BE A LOCAL OR OPENED SYMBOL

SIGNIFICANCE
The symbol in the argument field of a REF, DEF, or DEFG pseudo instruction is an OPENed, LOCAL or implied local symbol that has not been closed.

ACTION
The symbol in the argument field of either a DEF, DEFG, or REF statement must be a global symbol and cannot have appeared in a LOCAL or OPEN instruction. It also cannot be an implied local symbol.

SYMBOL MUST BE DECLARED REF OR DEF

SIGNIFICANCE
The symbol in the argument field of the END pseudo instruction has not been declared as an entry point.

ACTION
If the argument field contains a transfer address, the symbol must be declared as an entry point by appearing in either a DEF, DEFG, or REF pseudo instruction in the same assembly module.

SYNTAX ERROR

SIGNIFICANCE
The following conditions will cause this error:
- an illegal character string such as missing or misplaced quote marks,
- an illegal number such as a digit larger than the base allows, a base value other than binary, octal, decimal, or hexadecimal, an illegal character or a missing parenthesis,
- an illegal floating point number which includes any base designator (ie. all floating point numbers are decimal),
- expressions with mismatched parenthesis or illegal or missing operands.

ACTION
Check the ERS for the syntax of self-defining terms (ie. number values or character strings). Or check the expression in the source statement for illegal operands or missing operands. Note tha a blank or comma terminates an expression.

TOO MANY ARGUMENTS

SIGNIFICANCE
The argument field of the ALIAS statement contains more than one symbol.
ACTION
The ALIAS pseudo instruction allows only one symbol in the argument field (ie. there can only be one linkage symbol aliased to an internal entry point).

TOO MANY CHARACTERS IN SYMBOLIC NAME

SIGNIFICANCE
The symbol being processed has more than 31 characters in it.
ACTION
The maximum symbol length is 31 characters, redefine symbol to be less than 31 characters.

TOO MANY STATEMENT LABELS

SIGNIFICANCE
The instruction encountered can have only one symbol in the label field.
ACTION
If the instruction is one of the following statements, only one symbol in the label field is allowed: ALIAS, IDENT, PNAME, FNAME or a code generating statement which has the symbol category 6 (this includes the symbolic machine instructions).

"TRALABEL" FIELD INVALID

SIGNIFICANCE
In an IOU module, the 'tralabel' field of the END pseudo instruction is not blank.
ACTION
The 'tralabel' field of the END instruction is invalid in an IOU module and must be blank.

TRUNCATION ERROR

SIGNIFICANCE
The value that is being put into a field specified by a CMD or VFD statement must be truncated to fit.

ACTION

This message is turned on by the value 1 in the argument field of the TRUNC pseudo instruction. If no TRUNC instruction has been processed in the assembly module the value defaults to zero. Check the TRUNC statement in the ERS to see what constitutes loss of significance.

UNDEFINED OPERATION SUBFIELD

SIGNIFICANCE

The symbol in the operation field is not a pseudo instruction, symbolic machine instruction, an intrinsic or user-defined function (ie. appeared on a FNAME statement), a procedure definition (ie. appeared on a PNAME statement) or appeared on a CMD statement.

ACTION

Check the symbol in the operation field for a valid symbol that is either a pre-defined instruction or function or is a user-defined procedure or function.

UNDEFINED SYMBOLIC NAME "symbolic_name"

SIGNIFICANCE

This error occurs when trying to evaluate an expression or function where one of the operands or argument is undefined. It also occurs when a REF, DEF or DEFG symbol has not appeared as a label for a code generating statement.

ACTION

Symbol defintion occurs when a symbol appears in the label field of a statement (CPU, IOU or pseudo instruction) unless the label field is ignored or used for some other purpose.

VALUE OUT-OF-RANGE

SIGNIFICANCE

The following conditions will cause this error:

- ANAME - argument field < 0
- BSS/BSSZ - argument field < 0
- CMD/VFD - value in the length field < 0
- SET/EQU symbol - element number < 0
- LIST - argument field is incorrect value (check ERS for legal value
- INT/DINT - argument field must be in the following range:

CPU - -7FFFFFFF(16) < M < 7FFFFFFF(16)
IOU - -7FFF(16) < M < 7FFF(16)

- ORG - CPU - argument field < 0 or > 0FFFFFFFF(16)
  - IOU - argument field < load_address (from IDENT statement) or > 0FFF(16)
- DO/WHILE - argument field < 0
- SECTION - the offset, alignment, or maxsize values are < 0 or > 0FFFFFFFF(16)
- SKIPTO - F:(1,1) < 0
- PAGE - argument field < 0
- TRUNC - argument field does not equal 0 or 1.

ACTION

Check ERS for each pseudo instruction for the legal values.

X-REGISTER DESIGNATOR

SIGNIFICANCE

An X-register is required in the instruction.

ACTION

Check register specifications for instruction in ERS.

APPENDIX E

CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTION SUMMARY

REF #	INSTRUCTION	OPERANDS	OPCODE
001	LBYTS,s	Xk,Aj,Xi,D	D0->D7
009	LBYT,X0	Xk,Aj,Xi,D	A4
013	LBYTP,j	Xk,Q*	86
005	LXI	Xk,Aj,Xi,label*	A2
006	LX	Xk,Aj,label*	82
014	LBIT	Xk,Aj,Q,X0	88
020	LMULT	Xk,Aj,Q	80
016	LAI	Ak,Aj,Xi,D	A0
017	LA	Ak,Aj,Q	84
003	SBYTS,s	Xk,Aj,Xi,D	D8->DF
011	SBYT,X0	Xk,Aj,Xi,D	A5
007	SXI	Xk,Aj,Xi,label*	A3
008	SX	Xk,Aj,label*	83
015	SBIT	Xk,Aj,Q,X0	89
021	SMULT	Xk,Aj,Q	81
018	SAI	Ak,Aj,Xi,D	A1
019	SA	Ak,Aj,Q	85
022	ADDX	Xk,Xj	24
027	ADDR	Xk,Xj	20
143	ADDXQ	Xk,Xj,Q	8B
028	ADDRQ	Xk,Xj,Q	8A
166	INCX	Xk,j	10
029	INCR	Xk,j	28
023	SUBX	Xk,Xj	25
030	SUBR	Xk,Xj	21
167	DECX	Xk,j	11
031	DECR	Xk,j	29
024	MULX	Xk,Xj	26
032	MULR	Xk,Xj	22
168	MULXQ	Xk,Xj,Q	B2
033	MULRQ	Xk,Xj,Q	8C

APPENDIX E
CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTION SUMMARY

REF #	INSTRUCTION	OPERANDS	OPCODE
025	DIVX	Xk,Xj	27
034	DIVR	Xk,Xj	23
035	CMPX	X1,Xj,Xk	2D
036	CMPR	X1,Xj,Xk	2C
049	CPYXX	Xk,Xj	0D
053	CPYRR	Xk,Xj	0C
050	CPYAX	Xk,Aj	0B
051	CPYAA	Ak,Aj	09
052	CPYXA	Ak,Xj	1A
054	ADDAQ	Ak,Aj,Q	8E
055	ADDPXQ	Ak,Xj,label*	8F
056	ADDAX	Ak,Xj	2A
161	ADDAD	Ak,Ai,D,j	A7
057	ENTP	Xk,j	3D
058	ENTN	Xk,j	3E
059	ENTE	Xk,Q	8D
060	ENTL	X0,jk	3F
061	ENTZ	Xk	1F
	ENTO		
	ENTS		
164	ENTX	X1,jk	39
165	ENTC	X1,jkQ	87
169	ENTA	X0,jkQ	B3
065	IORX	Xk,Xj	18
066	XORX	Xk,Xj	19
067	ANDX	Xk,Xj	1A
068	NOTX	Xk,Xj	1B
069	INHX	Xk,Xj	1C
070	ISOM	Xk,Xi,D,j**	AC
071	ISOB	Xk,Xj,Xi,D	AD
072	INSB	Xk,Xj,Xi,D	AE
145	MARK	Xk,X1,j	1E
097	CNIF	Xk,Xj	3A
098	CNFI	Xk,Xj	3B
099	ADDF	Xk,Xj	30
100	SUBF	Xk,Xj	31
103	MULF	Xk,Xj	32
104	DIVF	Xk,Xj	33

APPENDIX E
CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTION SUMMARY

REF #	INSTRUCTION	OPERANDS	OPCODE
114	CMPF	X1,Xj,Xk	3C
105	ADDD	Xk,Xj	34
106	SUBD	Xk,Xj	35
107	MULD	Xk,Xj	36
108	DIVD	Xk,Xj	37
062	SHFC	Xk,Xj,Xi,D	A8
063	SHFX	Xk,Xj,Xi,D	A9
064	SHFR	Xk,Xj,Xi,D	AA
037	BRXEQ	Xj,Xk,label*	94
038	BRXNE	Xj,Xk,label*	95
039	BRXGT	Xj,Xk,label*	96
040	BRXGE	Xj,Xk,label*	97
041	BRREQ	Xj,Xk,label*	90
042	BRRNE	Xj,Xk,label*	91
043	BRRGT	Xj,Xk,label*	92
044	BRRGE	Xj,Xk,label*	93
109	BRFEQ	Xj,Xk,label*	98
110	BRFNE	Xj,Xk,label*	99
111	BRFGT	Xj,Xk,label*	9A
112	BRFGE	Xj,Xk,label*	9B
113	BROVR	Xk,label*	9E
	BRUND		
	BRINF		
045	BRINC	Xj,Xk,label*	9C
046	BRSEG	X1,Aj,Ak,label*	9D
047	BRREL	Xk	2E
048	BRDIR	Aj,Xk	2F
134	BRCR	j,k,label*	9F
115	CALLSEG	label*,Aj,Ak	B5
116	CALLREL	label*,Aj,Ak	B0
117	RETURN	jk**	04
118	POP	jk**	06
120	EXCHANGE	jk**	02
121	HALT	jk**	00
122	INTRUPT	Xk,j**	03
203	PSFSA	k	07
124	LBSET	Xk,Aj,X0	14
125	CMPXA	Xk,Aj,X0,label*	B4
126	TPAGE	Xk,Aj	16
127	LPAGE	Xk,Xj,X1	17

APPENDIX E
CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTION SUMMARY

REF #	INSTRUCTION	OPERANDS		OPCODE
130	CPYSX	Xk,Xj		0E
131	CPYXS	Xk,Xj		0F
132	CPYTX	Xk,Xj		08
136	KEYPOINT	j,Xk,Q		B1
138	PURGE	Xj,k		05
139	EXECUTE,s	j,k,i,D		C0->C7
074	ADDN,Aj,X0	Ak,X1	SD DD	70
156	ADDI,Xi,D	Ak,X1,j	DD	FB
096	CALDF,Aj,X0	Xk,Ai,D	SD	F4
084	CMPB,Aj,X0	Ak,X1	SD DD	77
085	CMPC,Aj,X0	Ak,X1,Ai,D	SD DD	E9
083	CMPN,Aj,X0	Ak,X1	SD DD	74
155	CMPI,Xi,D	Ak,X1,j	DD	FA
077	DIVN,Aj,X0	Ak,X1	SD DD	73
091	EDIT,Aj,X0	Ak,X1,Ai,D	SD DD	ED
089	MOVB,Aj,X0	Ak,X1	SD DD	76
154	MOVI,Xi,D	Ak,X1,j	DD	F9
092	MOVN,Aj,X0	Ak,X1	SD DD	75
076	MULN,Aj,X0	Ak,X1	SD DD	72
078	SCLN,Aj,X0	Ak,X1,Xi,D	SD DD	E4
079	SCLR,Aj,X0	Ak,X1,Xi,D	SD DD	E5
086	SCNB,Aj,X0	Ak,X1,Ai,D	DD	F3
075	SUBN,Aj,X0	Ak,X1	SD DD	71
088	TRANB,Aj,X0	Ak,X1,Ai,D	SD DD	EB
172	ADDXV	Ak,Aj,Ai,D		44
		Ak,Xj,Ai,D		
173	SUBXV	Ak,Aj,Ai,D		45
		Ak,Xj,Ai,D		
176	CMPEQV	Ak,Aj,Ai,D		50
		Ak,Xj,Ai,D		
177	CMPLEV	Ak,Aj,Ai,D		51
		Ak,Xj,Ai,D		
178	CMPGEV	Ak,Aj,Ai,D		52
		Ak,Xj,Ai,D		
179	CMPNEV	Ak,Aj,Ai,D		53
		Ak,Xj,Ai,D		
180	SHFV	Ak,Aj,Ai,D		4D
		Ak,Xj,Ai,D		
181	IORV	Ak,Aj,Ai,D		48
		Ak,Xj,Ai,D		

APPENDIX E
CYBER 180 CPU SYMBOLIC MACHINE INSTRUCTION SUMMARY

182	XORV	Ak,Aj,Ai,D	49
		Ak,Xj,Ai,D	
183	ANDV	Ak,Aj,Ai,D	4A
		Ak,Xj,Ai,D	
184	CNIFV	Ak,Aj,D	4B
		Ak,Xj,D	
185	CNFIV	Ak,Aj,D	4C
		Ak,Xj,D	
188	ADDFV	Ak,Aj,Ai,D	40
		Ak,Xj,Ai,D	
187	SUBFV	Ak,Aj,Ai,D	41
		Ak,Xj,Ai,D	
188	MULFV	Ak,Aj,Ai,D	42
		Ak,Xj,Ai,D	
189	DIVFV	Ak,Aj,Ai,D	43
		Ak,Xj,Ai,D	
190	SUMFV	Xk,Ai,D	47
191	MRGV	Ak,Aj,Ai,D	54
		Ak,Xj,Ai,D	
192	GTHV	Ak,Aj,Ai,D	55
		Ak,Xj,Ai,D	
193	SCTV	Ak,Aj,Ai,D	56
		Ak,Xj,Ai,D	
195	TPSFV	Ak,Aj,Ai,X0,D	58
		Ak,Xj,Ai,X0,D	
196	TPDFV	Ak,Aj,Ai,X0,D	59
		Ak,Xj,Ai,X0,D	
197	TSPFV	Ak,Aj,Ai,X0,D	5A
		Ak,Xj,Ai,X0,D	
198	TDPFV	Ak,Aj,Ai,X0,D	5B
		Ak,Xj,Ai,X0,D	
199	SUMPFV	Xk,Aj,Ai,D	5C
		Xk,Xj,Ai,D	
200	GTHIV	Ak,Aj,Ai,D	5D
		Ak,Xj,Ai,D	
201	SCTIV	Ak,Aj,Ai,D	5E
		Ak,Xj,Ai,D	

NOTE 1: *-This field will be modified by the Assembler.

NOTE 1: **-Parameter can optionally be left blank.

NOTE 2: SD and DD are Source Descriptor and Destination Descriptor. They both have the format F,T,L,O.

-end-